ABSTRACT

ARTIFICIAL INTELLIGENCE APPROACHES TO MUSIC COMPOSITION

Music composition using Artificial Intelligence is a well-established area of study with research dating back over six decades. From the time the mathematical model of computation was developed by Alan Turing in the 1940s, the question of whether computers can be built to match human level intelligence has been debated. Creativity is certainly considered to be a sign of intelligence, and many areas of Artificial Intelligence have pursued ways to emulate the creative spark found in humans. Music Composition via Artificial Intelligence falls into this category. This thesis explores the application of Artificial Intelligence approaches towards the goal of composing music by implementing three approaches found in Artificial Intelligence and studying their results.

Adil H. Khan

October 14th , 2013

ARTIFICIAL INTELLIGENCE APPROACHES TO MUSIC COMPOSITION

By

Adil H. Khan

Richard Fox, Ph.D., Committee Chair

Gary Newell, Ph.D., Committee Member

Alina Campan, Ph.D., Committee Member

Renee Human, Ph.D. Candidate, Committee Member

ARTIFICIAL INTELLIGENCE APPROACHES TO MUSIC COMPOSITION

Master of Science Thesis

A Master of Science Thesis submitted in partial fulfillment of the
Requirements for the degree of Master of Science
at Northern Kentucky University

By

Adil H. Khan
Highland Heights, Kentucky

Advisor: Dr. Richard Fox
Professor of Computer Science
Highland Heights, Kentucky

October 14th, 2013

UMI Number: 1549922

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1549922

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

DOCUMENT RELEASE

__X___ I authorize Steely Library to reproduce this document in whole or in part for purposes of research

_____ I do not authorize Steely Library to reproduce this document in whole or in part for purposes of research.

Signed: Adil H. Khan

Date: October 14th, 2013

Acknowledgments

I would like to acknowledge Dr. Richard Fox for his guidance and mentorship in the writing of this thesis along with the development of the accompanying software and his tireless help in editing and reviewing this document. Dr. Alina Campan for her feedback and review of this document. Dr. Gary Newell for his feedback and input. Prof. Renee Human for her feedback and review. To the Northern Kentucky University computer science department for the educational opportunities I have been honored to receive.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1. Introduction

Music composition using computers has been around since the 1950s [3]. Algorithmic music composition dates back centuries [4]. Artificial Intelligence (AI) techniques have been applied in the pursuit of solutions to a vast variety of problems, including emulating human creativity in areas such as music composition. Through the course of this thesis we will explore various applications of AI and implement our version of three of the approaches towards the task of music composition. We also examine the results of these approaches and explore their strengths and weaknesses. The goal of this thesis is to demonstrate AI approaches to the automatic composition of music.

1.1. Bridging the Creativity Gulf

AI approaches have been used to solve problems that do not easily submit to conventional programming methods. Areas with well-defined rules such as games (chess, checkers) have been met with considerable success [1]. Expert systems have met with significant success as theorem solvers and in collaboration with human experts utilizing domain specific knowledge [1].

Many of the problem solving skills that human beings possess fall in the domain of creativity. Creativity in itself is the subject of much debate and is not easily quantified. However, we as human observers do recognize solutions to problems that utilize creativity. Typically, creative solutions tend to employ novel ways of viewing an existing problem; whether we are talking of solving logical problems such as theorem solving, playing chess or more abstract endeavors such as those found in art and design.

Composing and/or performing music is generally agreed to be a creative pursuit. For humans, it takes years of practice and study to become competent at composing or performing music. Even though music can be thought of as inspirational and an art form, there are quite a lot of rules that govern music composition, with the study of music theory being an extensive pursuit in its own right. Computers and the software they run are especially suited for implementing algorithms and rules. Thus the endeavor to create music using computers and AI becomes a process of discerning the rules that make for good music and implementing it as software.

1.1.1. Creativity in Computers

Creativity by computers has been a debate for as long as computing has existed. The detractors argue that computers are incapable of creating anything new, merely of combining the sum of their data in different ways as to provide results which could surprise the user. Can computers be creative? Minsky, one of the pioneers of AI, answers this with a resounding “no”. The caveat being that there really is no such thing as creativity [6]. Furthermore, that which is commonly considered to be creative is instead a lack of understanding of how the mind of an artist works. Mundane everyday thought and creative thought are considered to be no different. Minsky further goes on to say that what in the end is classified as creative is a combination of intense interest, deep knowledge and proficiency in a given domain.

Supporters insist that human creativity itself is the recombination of existing knowledge and no different from what the computer is able to accomplish, albeit at a much slower pace [2]. This debate has not deterred the development of models of creativity and various attempts have been made to simulate creativity, such as systems to produce literature with concepts of betrayal and jealousy [5].

Creativity is also often described as ‘divergent’ thinking, where existing knowledge is applied in new, seemingly random ways. A popular example to illustrate this point is of solving the problem of flight. Instead of figuring out how to make the wings move to generate lift as is done in birds (and for centuries was the focus of this pursuit), the problem was reframed in terms of generating lift over fixed wings [7]. The important lesson here however is that the seemingly random idea that constitutes the creative breakthrough is relevant to the problem at hand and is deeply related to the context of the problem.

If randomness in itself was enough to constitute creativity, computers would be easily employed in creative pursuits, as random number generation is easily accomplished. In fact, most modern operating systems and programming languages provide a random number generator that creates a random floating point number between 0 and 1 which is seeded by some underlying hardware. Typically this is the system clock.

1.1.2. Using Artificial Intelligence

As mentioned earlier, and to be discussed in more detail in chapter 2, Artificial Intelligence has been applied to many areas to produce solutions to problems where the problem domain is too large to be conveniently solved by a simple algorithm. Computers and the software they run are inherently deterministic. Once an algorithm is implemented it will

always produce the same output for a particular input. One way to introduce variations in output is to implement randomness, typically using a random number generator.

One area where this approach has proven useful is in game-playing AI using *Monte-Carlo Tree Search* [8]. As illustrated in Figure 1-1 a tree is built in an asymmetric and incremental fashion.

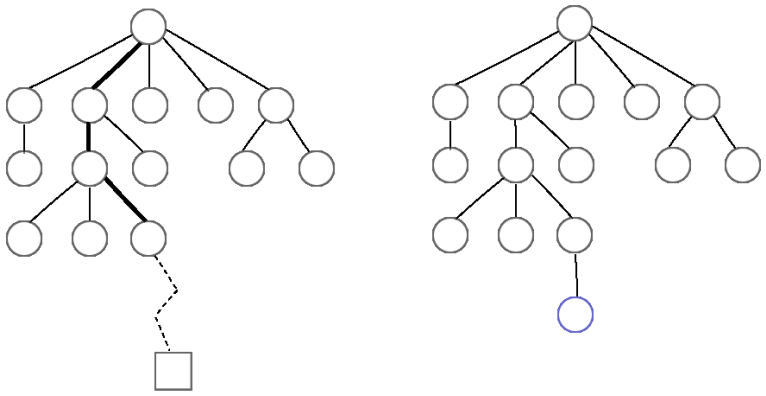

Figure 1-1 Basic Monte Carlo Tree Search Process [59]

Starting at the root node a child node is selected based on values relevant to the context of the application. Next, random moves are tried until some optimal result is achieved. The node representing this result is then added to the tree and the process continues for some specific amount of iterations [38]. In the domain of game AI, by randomly simulating moves, a structure is built that can be evaluated for factors that will make the game both challenging and unpredictable. Without the introduction of randomness the game would become easy to predict and get boring for the player. Even though randomness is useful in building an AI system that keeps gaming interesting for the player, randomness by itself is not sufficient in the larger context of producing intelligent behavior.

Prior knowledge is a vital factor in displaying intelligent behavior. For humans this is usually referred to as experience. For AI systems the logical approach is building a rule or knowledge database that the system can query. JAPE is an example of such a system that produced jokes in the form of riddles (punning riddles) [9]. For example "*What is the difference between leaves and a car? One you brush and rake, the other you rush and brake*".

AARON is an example of a computer program developed over many decades and is capable of creating original works of art. The author of the program, Cohen, insists that these works are original but not creative since the system is merely recombining data from its vast knowledge base [10]. MINSTREL and TALE-SPIN are examples of AI systems designed to

generate original works of literature [11, 12]. BACON is a system developed by Patrick Langley that (re)discovered laws of physics such as Ohms law when analyzing data.

Finally, and of special interest here is the use of AI in generating music. David Cope's EMMY and later EMILY HOWELL systems stand up as distinct examples of music generated using AI systems that were able to fool listeners into believing they were created solely by a human composer. The listeners even claimed to have been moved by the works, that is, until they found out the true nature of the composition [13].

A popular example of creative problem solving in recent years is found in IBM's *Watson* computer which defeated human opponents at the game of *Jeopardy.* Combining a vast knowledge base, along with machine learning, this system was able to provide the correct answers to questions many of which were in the form of puns. This is an achievement formerly considered to be outside the grasp of AI [23].

These examples highlight the success that has been achieved using AI systems that are capable of producing results and solving problems that were previously solely the domain of human creativity. Even though it is arguable that these systems have a long way to go to approach the level of creativity demonstrated by humans; research in these areas continues and improvements continue. Studying music generation through AI within this context is a worthwhile pursuit that on the one hand provides an avenue for better understanding human creativity, and on the other hand provides a framework whose development can be leveraged for all systems that endeavor to emulate it.

1.2. Music and Computers

Computer generated music has a history nearly as long as electronic computers. One of the first instances of computer based music is attributed to the *Ferranti Mark 1* computer at the University of Manchester in 1951[14]. This system had a 'hoot' command which on execution caused a tone to sound from its internal speaker. By providing frequency to this command, various pitches of the western musical scale could be emulated. This allowed the programmer to play 'God save the Queen' and 'Baa Baa Black Sheep' [15]. Five years later the *Illiac Suite* was created and is generally regarded to be the first piece of music composed by electronic computer [3]. The important distinction between these two milestones is that the former was a computer playing music composed by a human, whereas the latter was the computer composing the music itself.

1.2.1 Computer: Tool vs. Composer

Electronic musical instruments were among the first uses of electricity in the late Nineteenth century, with the *Telaharmonium* and the Singing Telegraph considered as the oldest known examples [16]. Electronic instruments proliferated throughout the Twentieth century with the advent of synthesizers in the 1950's and the *Moog* being the first instrument to actually be used by musicians. The arrival of the microprocessor in the 1970s and the MIDI standard in the 1980s led to further proliferation of electronic musical instruments, to the point today where music in nearly all genres has some component of electronic instruments employed. Electronic instruments continue to provide new and interesting ways for musicians to compose their craft. With increases in memory and processing, digital instruments have ironically been sought after as replacements for traditional instruments, even to the point of recreating imperfections found within legacy instruments as a way to create a more natural sound [17]. Throughout this history the key has been the skill and knowledge of the performer who uses these instruments as a tool to perform or compose music.

On the opposite end of the spectrum, the computer and the software it runs are given the role of the composer. Rather than simply being used as a type of musical instrument, the role of the computer is to emulate the processes that human composers apply in creating their works. The role of the computer programmer is to distill the rules of music composition, music theory, and the algorithmic representation of the creativity of the composer into code that the computer can execute to create original works of music. Algorithmic composition has existed in one form or another prior to the development of the modern computer. Indeed, a computer is not considered a pre-requisite for algorithmic composition [13]. From Pythagoras in the time of the ancient Greeks who understood the mathematical nature of music, through Mozart who used his *Mechanical Dice* as a compositional tool and the works of John Cage who used chess moves to dictate the sounds of the composition, give us examples of algorithmic music that predate or circumvent the use of a computer [18]. The development of more sophisticated and powerful computers and the rise of Artificial Intelligence techniques then pave the way for even more extensive use of computers to implement algorithmic approaches to generating music.

1.2.2. Approaches available in AI

Next, we highlight some AI-based approaches to music composition. These algorithms have produced varying rates of success. We will go into more details on these systems in chapter 2, here we highlight the major categories and give some remarks on them.

Papadopoulos and Wiggins categorize AI approaches to music compositions with the following six categories [19]:

i. Mathematical Models
ii. Knowledge Based Systems
iii. Grammars
iv. Evolutionary Methods
v. Learning Systems
vi. Hybrids

Mathematical models were the earliest and least complex method of generating music. Markov chains and other probability based computations are utilized in rendering pitch, duration and other characteristics of the music. The major drawback to this approach is that existing music has to be analyzed in order to build the statistical models. Furthermore, attributes like distinctiveness and style become difficult to capture in statistical models [19].

Knowledge Based Systems (KBS) have been used in diagnostic systems extensively, and the key feature is that a KBS can explain how it arrived at its conclusion to a problem. In the arena of music composition, KBS have the major drawback in that it is difficult to build a Knowledge Base from subjective knowledge domains such as music. Furthermore, 'exceptions' to rules make music interesting and such exceptions can be hard to capture in a KBS [19].

Grammar based systems like EMI, [13] are well suited for composing music that mimics or captures the style of a particular composer. In the case of EMI a minimum of two pieces of music are analyzed and grammar rules extracted from them in order to generate new original music. The drawback of this approach is the complexity involved in parsing the existing music which is not hierarchical. Indeed, Cope, the creator of EMI had to generate a lot of his databases by hand which took months at a time. The output of EMI is familiar to listeners of the original music on which the databases are built, since some sections are produced identical to the original works of the artists [13].

Evolutionary methods including genetic algorithms (GA) and genetic programming are especially suited for providing 'good enough' solutions in large search spaces. A solution to a given problem is evolved over many generations with a starting pool of candidate solutions. The key feature of GA's is the fitness function that is employed at each generation to select the fittest solutions to be used in the production of the subsequent generation [20]. For music composition, designing the fitness function is the biggest challenge. Some systems have

delegated the task of the fitness function to a human listener who picks the best solution after a designated amount of generations. The obvious drawback to this is the bottleneck of the human fitness evaluator.

Learning systems implement artificial neural networks for sub-symbolic, distributed learning whereas machine learning (ML) is implemented at the symbolic level [19]. These systems are trained by exposure to data, music in this case, and the knowledge captured within the system is then used to generate new, original music. For example one system was trained to detect musical tension by having human subjects provide input during a musical performance of a symphony [20]. The power of neural networks is recognized in their ability to learn sub symbolic concepts whereas music is generally focused around rules. This then is the major drawback with learning systems.

Hybrid systems use a combination of the above techniques to their strengths with the goal of achieving better results than those possible with a singular approach. The results with these systems have been promising. The drawback is that they are difficult to implement and it is challenging to validate their results. This however is the most promising avenue for future research [19].

As we have shown above there are a multitude of ways toward solving the problem of generating music with AI. Next, we list the methods that are to be implemented for this thesis. Details of the implementation will be discussed in Chapter 3. What follows serves as a brief introduction.

1.3. A Multi Algorithmic Music Composition System

The realm of music is an infinitely diverse space. Genres and sub genres of music abound. In order to keep the scope of this thesis somewhat limited, the music to be generated is going to be popular (pop) music. This allows us to consider a simpler framework when considering song structure, range, length of song etc. Furthermore, popular music is well studied and resources to understand it are readily available.

As enumerated in the previous section there is a multitude of ways to approach the problem of generating music using AI. For the purposes of this thesis, three approaches are implemented so that their results can be studied and compared.

The first is the stochastic approach using Markov Chains [21]. The second approach is knowledge-based using a routine planning algorithm [22]. The third approach is based on

genetic algorithms [20]. The system built for this purposes has been named MAGMA: the Multi AlGorithmic Music Arranger.

1.3.1 MAGMA Overview

MAGMA is constructed to allow a user to input preferences defining what they would like the song to sound like. Based on these preferences of *Mood, Repetition, Variety, Transition*, *Range* and a choice of one of the three algorithms the system builds a song file in MIDI format.

The stochastic algorithm uses Markov Chains to capture the likelihood of one event following a sequence of prior events. MAGMA's implementation relies on a database of musical components such as pitches, durations, song structures and so on to build the transition matrices that comprise the stochastic algorithm. This database consists of simple text files that are built from musical sources such as song books and popular music websites that have chord progressions and melody listings. Each entry in the database is tagged with its attributes so that MAGMA will pick the entries in the database that most closely match the user preferences to build its stochastic transition matrices.

The routine planning algorithm, similar to the stochastic approach, relies on a database of existing musical attributes to select from for its solution. Unlike the stochastic approach which uses statistical data, the knowledge base in routine planning consists of knowledge accumulated by an expert (in this case, song composer) through experience. This knowledge will include song structures, chord sequences, melody sequences, and rules of when each might be applicable. Once again the entries in the database are tagged with attributes so that the ones matching the user preferences are picked. In the case of multiple matches one is chosen randomly. MAGMA then iterates through its database of chords, melody notes, durations etc. to build the song.

Unlike the stochastic or planning system, the genetic algorithm is implemented with no internal database. However, music theory and attributes used to evaluate pop music are woven into the fitness functions. The fitness functions are applied to all components of the song as it is generated: overall structure of the song, the melody sequence, chord sequence, measure structure. For each component of the song, the genetic algorithm starts with a randomly generated pool of solution candidates. Through each of the iterations of the algorithm a new generation is created based on improvements to the prior generation. Once the predetermined set of iterations is completed the components are combined as in the other two algorithms to produce the completed song.

MAGMA's architecture is designed in a way that the implementation of each algorithm complies to a standard interface. This will make it possible in the future to add other algorithms as well as combine algorithms based on user preferences.

1.3.2 Evaluating the results

To evaluate the quality of the output from MAGMA, a series of tests were run. The first set of tests evaluated the output from the system based on user preferences and how well the output matched the expectations from the user's input. It is understood that this is a highly subjective analysis. The goal for future work is to allow the system to generate songs via a user friendly interface so that end users can evaluate songs and accumulate results for a larger comparison.

For this thesis, the songs were evaluated on several criteria. This includes statistics such as the number of chords and notes in a song and how they varied with user input. How the song structure varied with user input and how listenable the songs were and what user preferences led to the most listenable songs. The results of these experiments will be incorporated in any future work done on MAGMA.

1.4. Thesis Outline

Chapter 2 provides detail on AI and AI algorithms as they are applied to music composition. We will look at algorithms first independent of their application and then specifically how they are utilized in music composition. We will analyze how applicable these algorithms are to music composition and by examining their strengths and weaknesses to this problem.

Chapter 3 describes MAGMA in detail. The chapter will present the overall structure of the system followed by details on the implementation of the three selected algorithms.

Chapter 4 describes the testing and evaluation parameters. Various song excerpts are examined comparing songs generated using the same parameters with the three algorithms. We will also provide the results of the statistical analysis compiled.

Chapter 5 provides the conclusion to the thesis. We examine the strengths and weaknesses of MAGMA and the implementations of the three selected algorithms. We then discuss future work including areas of improvement.

Chapter 2. Literature Review

This chapter provides background knowledge for the work done in this thesis. We first explore areas of artificial intelligence (AI) related to the implementations found in this thesis: stochastic reasoning, planning and genetic algorithms. We then conclude the chapter with a survey of AI approaches in generating music.

2.1. Artificial Intelligence

The term *Artificial Intelligence* was coined by John McCarthy in the 1950s and one of the simplest definitions of AI is 'the branch of computer science concerned with the automation of intelligent behavior' [1]. The field of AI is vast and encompasses many different algorithms and methodologies. Since the software implementation for this thesis relies on Stochastic Reasoning, Planning and Genetic Algorithms we will go into more details on these approaches after covering some background information on AI techniques and the algorithms implemented by them.

A common construct to depict any system with a set of possibilities is a Graph [24]. Graphs are also used to provide an abstract representation of a problem, with each node in the graph representing a state in the problem solving process. For example in a graph representing the path of a salesperson through various cities, each node can represent a city to be visited by the salesperson and the arcs, each with a weight value, can represent the paths between the cities. An algorithm can now be designed to find the least weight path that can be taken by the salesperson in such a way that each city is visited exactly once. This scenario is depicted in Figure 2-1.

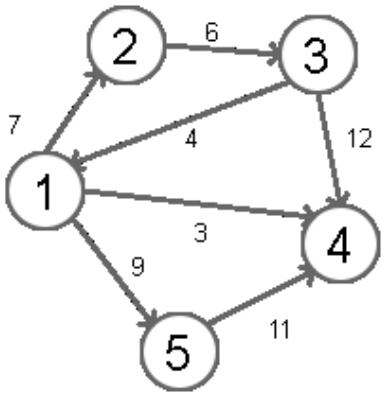

Figure 2-1 Graph with 5 nodes and a few possible paths.

Abstracting the graph representation further and using it to represent a *problem space*, each node in the graph can then be viewed as a partial solution state to the problem represented by the graph. Some of the nodes can represent the initial states and while others can represent the goal states. The arcs that connect these initial states to the goal states constitute the solution path. Search then can be defined as the process of finding paths from the initial states to the goal states.

Any algorithm that implements search must keep track of the operators from the initial nodes to the goal nodes that give us the path through the search space and ultimately lead to the solution. Not all paths through a graph are efficient and one of the challenges of a search algorithm is discerning the most efficient path from the initial state to the goal state. Furthermore, it is also possible to have paths that are cyclic which would leave the algorithm running in an infinite loop. Naturally the algorithm's design has to incorporate the avoidance of such loops [1]. One approach is to incorporate heuristics into the search space.

Heuristics and *heuristic search* are an effort to reduce the size of the search space by using rules to follow those braches in a state space search that have the greatest likelihood of reaching a solution state. Typically heuristics are used when there are ambiguities in which path should be followed through the search space. For example, in medical diagnosis, a set of symptoms could have several causes each represented within the problem domain as separate paths. An additional layer of heuristics can help select the likeliest path. Another scenario where heuristics are employed is when the search space is too complex and any brute force method cannot provide a solution in a reasonable amount of time. In games like chess each move creates a large number of subsequent moves which cannot be evaluated efficiently unless heuristics are used to trim down the possible moves. Another example is the game of tic-tac-toe in which there are 9 possible moves in the first step, followed by 8 for the second step, seven for the third step and so on. This gives an estimate of 9! or 362,880 possible moves in the search space as depicted in Figure 2-2.

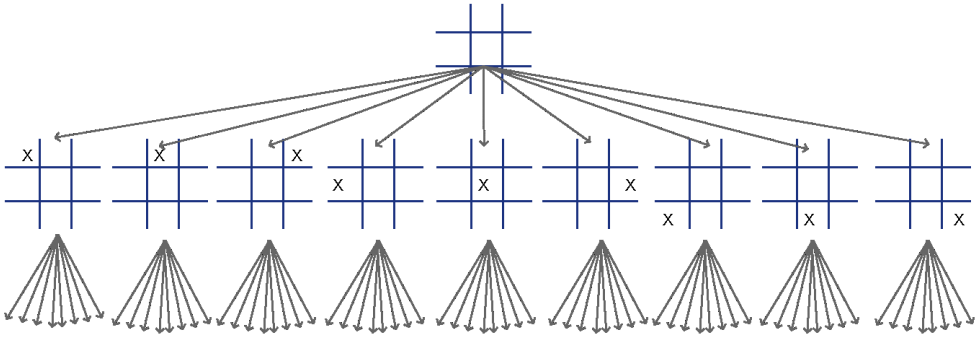

Figure 2-2 Naive search space for tic-tac-toe in first 2 moves

However an observation that the initial move can really only have three possibilities, a corner, center of a side or the center of the grid, can reduce the initial state to just 3 instead of 9. On the second step of the game, applying symmetry again as shown in Figure 2-3 there are 5 moves for a corner position, 5 for a side position and 2 for the center position; giving a total of 12 possible places for the next (3^{rd}) move. The possibilities after the first move is now 12x7!. Giving a total combination of 3 + 12x7! or 60483 moves. This is a 80% reduction of the search space by the application of simple heuristic. Other heuristics can in fact be applied to reduce the search space even more [1].

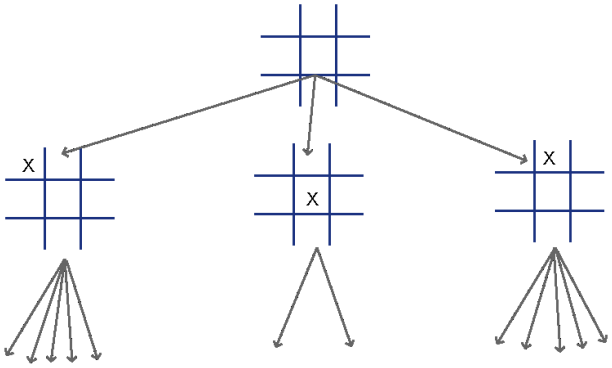

Figure 2-3 Reduced tic-tac-toe search space by applying heuristic

The game of tic-tac-toe while being a good example is not necessarily representative of a real world system. Rarely are the problem domains for which we are seeking solutions so readily defined by rules. Human experts in their respective fields rely on experience and intuition to solve a large variety of problems. Heuristics that are designed for systems meant to replace human experts have the potential to fail where the knowledge is incomplete or a situation is encountered which the designers of the heuristic did not foresee [1]. There are however problems that even with the application of heuristics do not yield solutions in reasonable amount of time for all but the smallest data sets.

Any such problem that results in an exponential growth in computation requirements with an increase in its size falls into the class of *intractable* problems. In order to be effective in solving problems from the real world, any AI methodology that attempts to model a problem domain by reducing it to sub-problems must ensure that the sub-problems are tractable. The travelling salesperson problem described earlier is a classic example of an intractable problem. The size of the search space grows as a factorial of the number of cities in the graph. Since

intractable problems grow exponentially (or worse) it is common to overestimate the efficacy of any algorithm that performs well in smaller data sets. However, without a proper understanding of the growth of the problem space, algorithm design is pointless.

One method to gauge the complexity is the *branching factor* of a space. This is the count of the average number of branches (or children of a given node) from the current state in the space. The number of states at depth n of the search is equal to the branching factor raised to the power of n. So for example, a binary tree where each node has two children has a branching factor of 2. So the search space for a tree of depth 10 is 2^{10} or 1024 sates. This is a number that is easy to contend with even with modest computational resources as long as the depth doesn't increase beyond this number. Compare this to a chess game where there are approximately 35 legal options available at any given state, and we have a branching factor of 35. So if chess only had 10 turns available in a game this would give 35^{10} states. Given that chess can have 50 or more turns in a game the search space can be 35^{50}. Some calculations that include non-legal options put this estimate at 10^{120} which is greater than the number of molecules in the observable universe. These examples show that to solve any non-trivial problems the accurate design of heuristics is vital especially where they can help prune the search space to manageable sizes. A contributing factor to the large search spaces of problems is the variability of the environment. In order to estimate and project every possible scenario within a tree like structure causes the size of the tree and consequently the search space, to grow exponentially. Machine learning is one way of developing systems that are adaptive enough to changing circumstances and thereby eliminate the need to have to search through large spaces for solutions [1].

Knowledge acquisition, or learning is a key feature of any entity that exhibits intelligence. In humans, knowledge is gained over decades through experience, observation and from mentoring by teachers. AI approaches to learning attempt to simulate these aspects by training the agents with minimal amounts of knowledge and examples and use the process of induction to achieve generalization. Heuristics in the form of rules help augment the gaps in knowledge that tend to be found as a result of incomplete data or examples. The learning processes can be viewed as several distinct tasks that have to be accomplished in order for learning to occur. This process consists of the data and goals of the learning task which in turn is composed of domain specific knowledge and categorizations relevant to the learning agent. Furthermore, there must be available a mechanism to represent the learnt knowledge. Typically this is accomplished through a representational language for the domain. For instance, a language to represent the concept of a ball could be written using:

$$size(X,small) \wedge color(X,red) \wedge shape(X,round)$$

which is a generalization of terms like:

$$size(obj1,small) \wedge color(obj1,red) \wedge shape(obj1,round).$$

In these statements the operators of size, color and shape are applied to their arguments. For example, *size (obj1, small)* describes that the *size* of an object *obj1* is *small,* small being described elsewhere in the system and relevant to the context of the problem domain.

Similarly, other operations can be applied by the system to learn generalizations from examples. These operations and the representational language together combine to form the concept space which can then be searched to come up with new generalizations when instances are found that don't correlate with the current representation. The key distinction in this type of learning is the use of symbols to represent concepts. In domains where symbols are not easily defined must utilize the connectionist model of knowledge and are implemented using neural networks. Figure 2-4 shows this general model of the learning process. As depicted in the diagram operations from a representational language are utilized to describe the problem domain. A combination of heuristic search and specific goals are then applied to generate new knowledge which together constitutes the learning process [1].

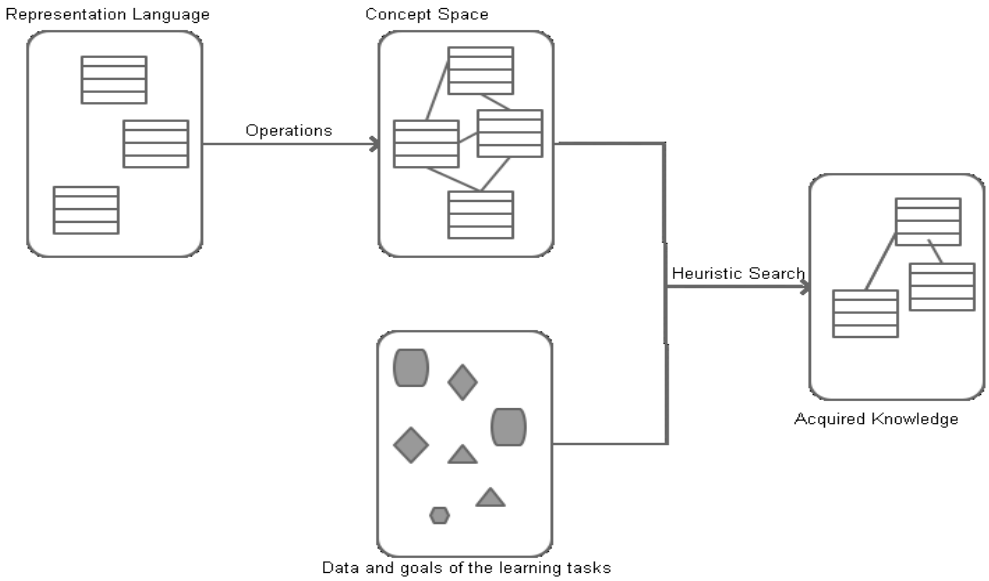

Figure 2-4 General model of learning process [1]

In the previous example of representational language the definition of ‘ball’ includes the color red. If however we wanted to represent various shades of red, for example in an image recognition program, the power of symbolic representation becomes limited. Artificial neural networks operate on the sub-symbolic level to achieve learning not possible on the symbolic level. The neural network is loosely modeled on biological brains where a network of neurons work in concert to propagate or suppress signals based on their internal biases. No single neuron is responsible for learning a concept; rather the knowledge is spread out over the network. This feature of distributed learning is modeled in artificial neural networks (ANN). Unlike the symbolic learning systems, ANNs are trained from examples and feedback by the trainer instead of being explicitly programmed. A network for example may be trained for facial recognition from a training set of a few dozen faces. It can then successfully discern between an image of an apple and a face that it has not experienced before. In fact neural networks are ideally suited for classification and pattern recognition where concepts are not easily defined using symbolic logic. The heart of the neural network is the neuron which is a simplified version of the neurons in biological brains [1].

The neuron is modeled with a set of inputs and an activation threshold. If the input sum exceeds the activation threshold of the neuron then the neuron ‘fires’, or in essence propagates the signal further. This could be to other neurons in the network or could be the final output of the network. The weight of each neuron is the value that is modified by the trainer of the network based on the performance of the network. Figure 2-5 shows an example of an artificial neuron. The threshold function f determines if the neuron will have an output of 0 or 1. Only if the sum of the weights of the inputs exceeds the threshold value then the neuron will fire resulting in an output of 1.

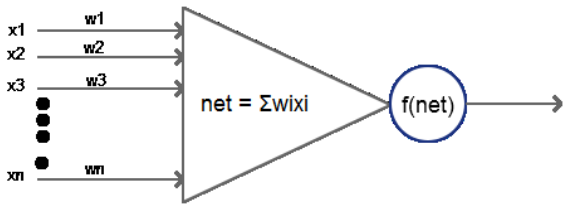

Figure 2-5 Artificial neuron

We have discussed here briefly some of the different areas of artificial intelligence and their applications. In the next sections we discuss more deeply Stochastic, Planning and Genetic Algorithms that are used within implementation of this thesis.

2.1.1. Stochastic Method

In problem domains where the search space is sufficiently large for a heuristic approach to be infeasible, a stochastic approach can help reduce the size of the search space. In compositional systems stochastic algorithms are the simplest of the tools available [33]. In order to capture the probabilities within a dataset a convenient tool to use is the *Markov Chain* [1]. This can be described as a state machine that gives the probability of transitioning from one state to the next. If we have a set of states, $S=\{s_1,s_2,s_3,\ldots..s_n\}$ and if we are in state s_i then there is a probability of p_{ij} to move to state s_j. This probability is not dependent on which state the chain was in before the current state. Since they generate output based on probabilities and yet are random, their output is not easy to predict. Additionally in situations where there is the ability to sample existing data to discern trends, stochastic approaches allow a system to produce new output that correlates with the probabilities existing within the sample data. Where the sample data is generated by human performance such as speech or music, then a Markov Chain built from this data can generate output with the same likelihoods as the human performer. Hence any system that uses human performance data to model its execution will be able to simulate human intelligence to a degree [1].

A common example when describing Markov chains is that of predicting weather. In this simplified version of a weather forecasting engine we assume that the weather for any given day is going to be a combination of the probabilities for *n* previous days. This can be described as:

$$P(w_n \mid w_{n-1}, w_{n-2}, \ldots..w_1)$$

So the observation at w_n depends on the *n* prior states that are included in the chain. Typically just the 1 prior state is used as the predicting factor and this is referred to as a *First Order Markov Chain.* A transition matrix and state diagram for this scenario is illustrated in Table 2-1 and Figure 2-6

Table 2-1 Transition Matrix for weather forecasting

	Tomorrow's Weather			
Today's Weather		Sunny	Rainy	Foggy
	Sunny	0.8	0.05	0.15
	Rainy	0.2	0.6	0.2
	Foggy	0.2	0.3	0.5

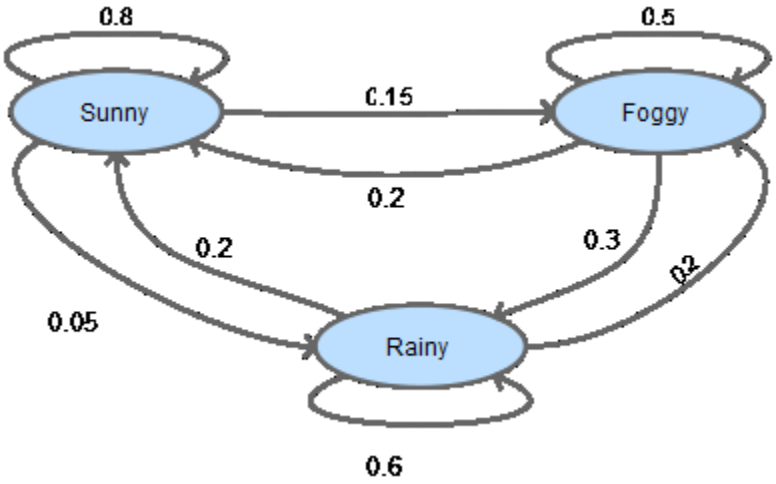

Figure 2-6 Markov state transition diagram for weather forecasting

One area where stochastic approaches have found considerable success is text to speech synthesis. These systems rely on accurate databases of words and the phonemes that represent how a combination of letters is combined to produce the sounds in a language. For the English language there may be multiple ways to pronounce a word. As a result transition matrices come in handy when trying to predict what the sound of the word should be. Figure 2-7 illustrates the probabilities in such a system when attempting to pronounce the word 'tomato' [1].

Whereas a database of lookup values can provide information for matching word segments to phonemes, this type of a data structure is unable to handle ambiguous situations. The state machine model is better at matching the context of word segments. In this way, for example, words like 'read' can be pronounced accurately based on the tense of the sentence and other contextual information. In the absence of complete contextual information the system

can utilize the available probabilities to come up with the 'best guess' which based on probabilities will be accurate in more scenarios than not.

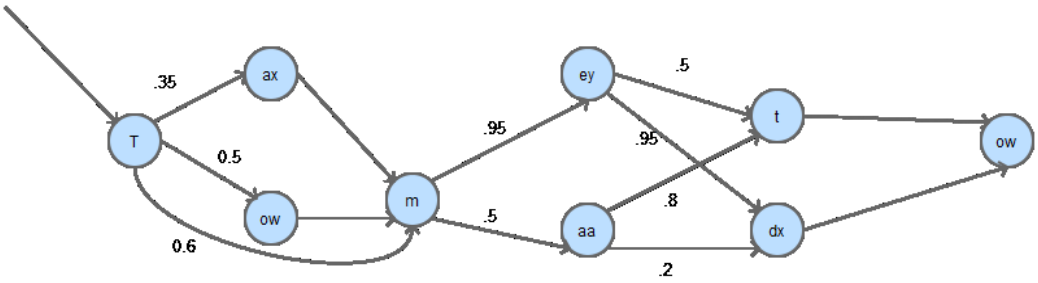

Figure 2-7 Transition diagram for speech synthesis [1]

In the Markov process described above, all the states are observable and provide a discrete sequence of events in time. In examples such as the one on weather prediction, there is a direct correspondence between a state and an observation. In many real world applications however, this direct correspondence between observation and state does not exist. Instead the observation is a probabilistic function of a given state, and more generally, a particular sequence of observations can correspond to different sequences of states. In these cases the states are *hidden* and have to be modeled by an extension of the Markov Model known as the Hidden Markov Model (HMM). For example in the case of speech recognition, the phonemes that make up the words of the language form the hidden layer of the HMM, whereas the noisy acoustic signals that are generated from the user's speech are the observable states. The phonemes that the speaker intended to use are probabilistic functions of the acoustic signals generated by the user and can be classified based on these functions [1]. Using the classification strength of HMMs is also useful in generating representative maps for path planning and obstacle avoidance. For example HMMs have been used to classify terrain into navigable and non-navigable areas which are then in turn used to generate plans for automated robots to navigate a terrain [52].

2.1.2. Planning Systems

The impetus for developing planning systems is to be able to guide automated agents such as problem solving systems and robots through complex tasks in varying environments. The essence of implementing planning systems is to break down a task into a sequence of lesser tasks, and those to even lesser tasks until all the tasks can be solved by applying atomic

operations that are understood by the agent. The agent, be it robot or software, can then traverse through the tree of possible sequences based on the requirements of the problem it is attempting to solve [1].

There are several major types of planning systems, some of which solve the problem linearly (one step at a time), others which follow a non-linear path and solve problems in parallel while yet others that change the paths based on feedback at each stage. However, common to all planning systems are the following steps [53]:

- Choose the best rule to apply next, based on current information.
- Apply the chosen rule and compute the new state of the system.
- Detect when a solution has been found.
- Detect dead ends so they can be abandoned, backtrack to a previous choice and pursue a better path.
- Detect when a close enough solution has been found and apply specific techniques to make it totally correct.

The major hurdle with planning is that the space for the possible sequences can get extremely large for all but the most trivial problems. Furthermore the planning system may be prone to rigidity with an inability to change its actions based on changes in the environment. If flexibility is to be added to the system then operations have to be available to analyze changes to the environment and account for them in planning or re-planning. This can increase the search space exponentially.

Planning is an activity that humans learn by building on prior knowledge, also known as experience. They become more adept at carrying out extremely complex tasks through this experience based on trial and error feedback.

Take for example the task of grocery shopping. The steps involved usually begin with taking stock of inventory and preparing a list. One then drives to the store and based on the list acquires the items in the quantities required, drives back home and restocks the kitchen. Although this seems to be an activity of only a few steps there are a large number of sub steps involved. Just making the list calls upon the ability to navigate the kitchen, determine which items require replenishments (the milk carton maybe full but it may expire soon, whereas the cheese is almost gone but isn't going to be used all week) and which do not. The next step is of building the list based on these observations and perhaps breaking it down by store, if different types of stores are to be visited. Additional complexity in the task is navigating to the store(s),

and then within the store(s) and finally, navigating in an efficient and cost effective manner. The steps to accomplish this task can be seen broken down in Figure 2-8.

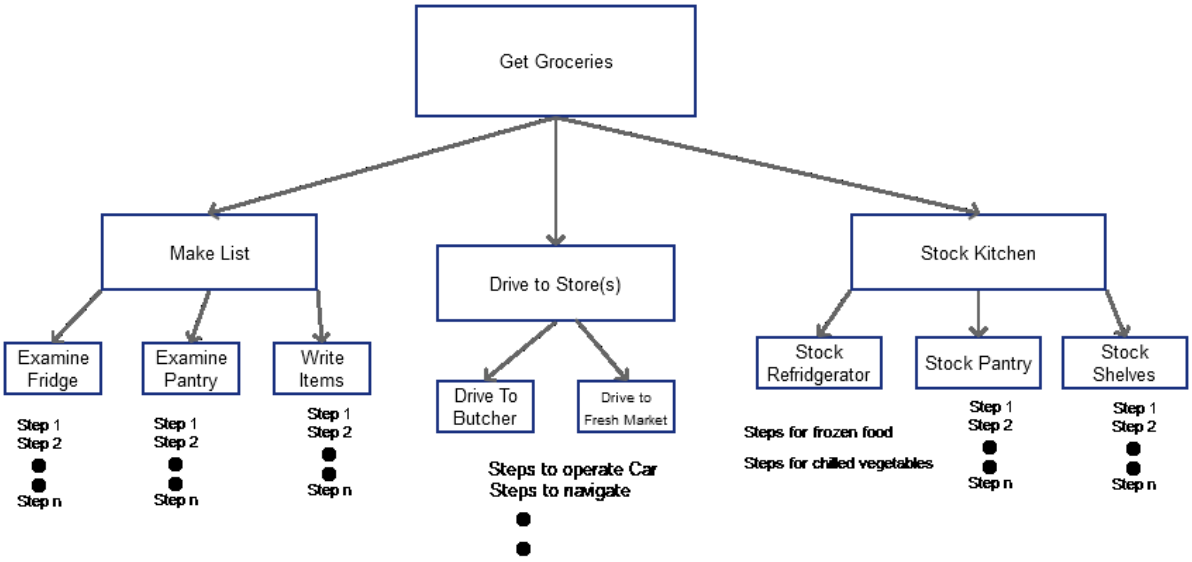

Figure 2-8 Plan decomposition for the task of grocery shopping

Here the main task of getting groceries is broken down into sub tasks, which are then further broken down into smaller tasks. The Goal of any such planning system is to decompose the tasks into atomic units that are well defined. Examining this everyday task of getting groceries in this way makes it clear that there are a large, nearly intractable, number of steps that any system would have to follow in order to be able to accomplish this task.

In order to simulate this aspect of human behavior, a major goal of AI research has been to design systems that can describe a set of plans that an agent (software or robot) can follow in order to accomplish an end goal. The two major components of a planning system is a rich database of operators or plan steps that can be selected from during the decision making process, and awareness within the system of what its current state is, so that the decision mechanism can choose from the relevant facts that apply at a particular moment in time [1].

One of the first systems to implement the Planning approach is *STRIPS* (STanford Research Institute Planning System) [29]. *STRIPS* used a mechanism known as the goal stack, where the states and the operations the system can apply all exist on a stack and have to be linearly evaluated. *STRIPS* is designed to control a robot (*SHAKEY)* that moves from room to room and rearranges blocks based on the goals given to it. Built in the 1960s, this approach has been the basis for many AI planning systems.

STRIPS uses a series of relatively simple atomic operators that are part of its knowledge base: *pickup, putdown, stack* and *unstack.* Each of these operators has a precondition, a delete list and an add list. The precondition is checked first to ensure that the state of the system is such that the requested operation can be performed. If the precondition is satisfied, the system proceeds to the next step which is executing the instruction. Once the instruction is executed the system must now update its internal state. This is done by deleting all of the state items in the delete list and adding all of the state items in the add list. This now becomes the new state of the system. For example we can look at the *Pickup* instruction, which directs the robot to pick up a block, labeled x, from a table as shown in Figure 2-9:

Pickup(X)
Precondition : ONTABLE(X) Λ HANDEMPTY Λ CLEAR(X)
Delete List: ONTABLE(X) Λ HANDEMPTY
Add List: HOLDING(X)

Figure 2-9 STRIPS pickup instruction

In order for the robot to pick up a block the following preconditions have to be true: The block is on the table, the robot's hand is empty and the block is clear (no other block on top of it). If the preconditions are satisfied, the robot can pick up the block. After the block is picked up, *STRIPS* modifies its internal state. The delete list specifies the states that are to be removed, so the HANDEMPTY, and the block being ONTABLE are removed. The state that the robot is HOLDING a block is added. Any future moves that are part of the robots plan will now begin with this state.

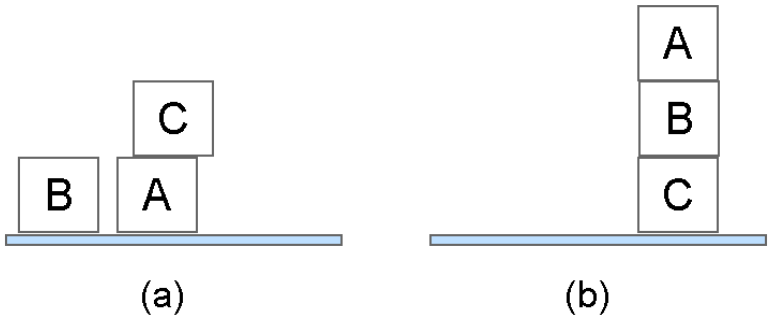

Figure 2-10 Initial and final states

As an example see Figure 2-10 where (a) is the initial state and (b) is the goal state. Using the predicate logic of STRIPS the initial state can be described as:

$$ontable(B) \wedge ontable(A) \wedge on(C,A) \wedge clear(B) \wedge clear(C) \wedge handempty$$

The Goal state is defined as

$$ontable(C) \wedge on(A,B) \wedge on(B,C) \wedge handempty$$

Since STRIPS uses linear planning each conjunctive phrase in the goal state is a sub-goal. If the goal on(A,B) is to be accomplished first then this could be done with the sequence:

pickup(C) -> putdown(C) -> pickup(A) -> stack(A,B)

This gives the state shown in the diagram in Figure 2-11.

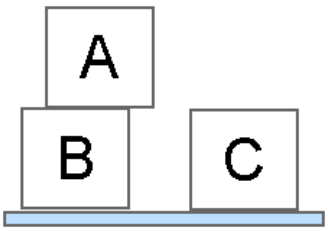

Figure 2-11 State with on(A,B) goal accomplished

However now in order to accomplish the next goal of on(B,C) Strips would follow the path of :

unstack(A,B) ->pickup(B) -> stack(B,C).

This move unfortunately undoes the previous goal of on(B,A) and demonstrates that linear planning though simple to implement has major drawbacks since only one goal is being operated on at a time. This leads to the strong possibility that accomplishing one goal can undo a previously accomplished goal [1, 29].

ABSTRIPS (Abstraction-Based STRIPS), built on top of STRIPS, uses hierarchical planning to overcome the limitations of linear planning systems. This more closely resembles planning done in the real world where plans are usually broken down into major and minor plans. For example, in the groceries scenario introduced earlier, a major step is to drive to the store. Minor steps include starting the car, reversing from the drive way, turning left onto the road etc. The major benefit of hierarchical planning is that it significantly reduces the search space. Instead of having to try out a large number of sequences linearly that fit the path from the initial to the goal state, the higher level goals can be computed first and if there are errors found they are found early. In this way when the time comes to compute the detailed steps they are computed from a much smaller subset of the steps available to the system [54].

GPS (General Problem Solver) employed means-end analysis towards theorem solving [26]. In means-end analysis a system examines the difference between the current state and the goal state and selects an operation that will reduce the difference. For example if the current state contains an 'and' ($\wedge$) operator and the goal does not contain this operator, then means-end analysis would apply a rule such as De-Morgan's Law to replace the $\wedge$ with an 'or' ($\vee$) operator. Utilizing this approach, GPS therefore would calculate the difference between an initial state and a goal state and then move towards reducing that difference recursively. When the difference is zero the goal state has been achieved.

The Figure 2-12 shows the flow of the GPS. There are two main components in GPS. The first is the procedure to compare two states to reduce their difference and the second is the table of connections which gives the links between problem differences and the transformations to reduce them. As can be seen in the figure the first step is to determine the difference D between an object A and B. The next step is to reduce the difference D. This is where the table of connections is used to look up the operator Q that can be applied to reduce this difference. By the recursive application of the operators GPS is able to reduce this difference to zero which indicates the achievement of the goal state. Since this table of connections could be replaced for specific problems outside of theorem solving the problem solver was called 'general'. By employing this methodology GPS was able to solve many logic problems such as the Towers of Hanoi, prove several theorems and even play chess [26].

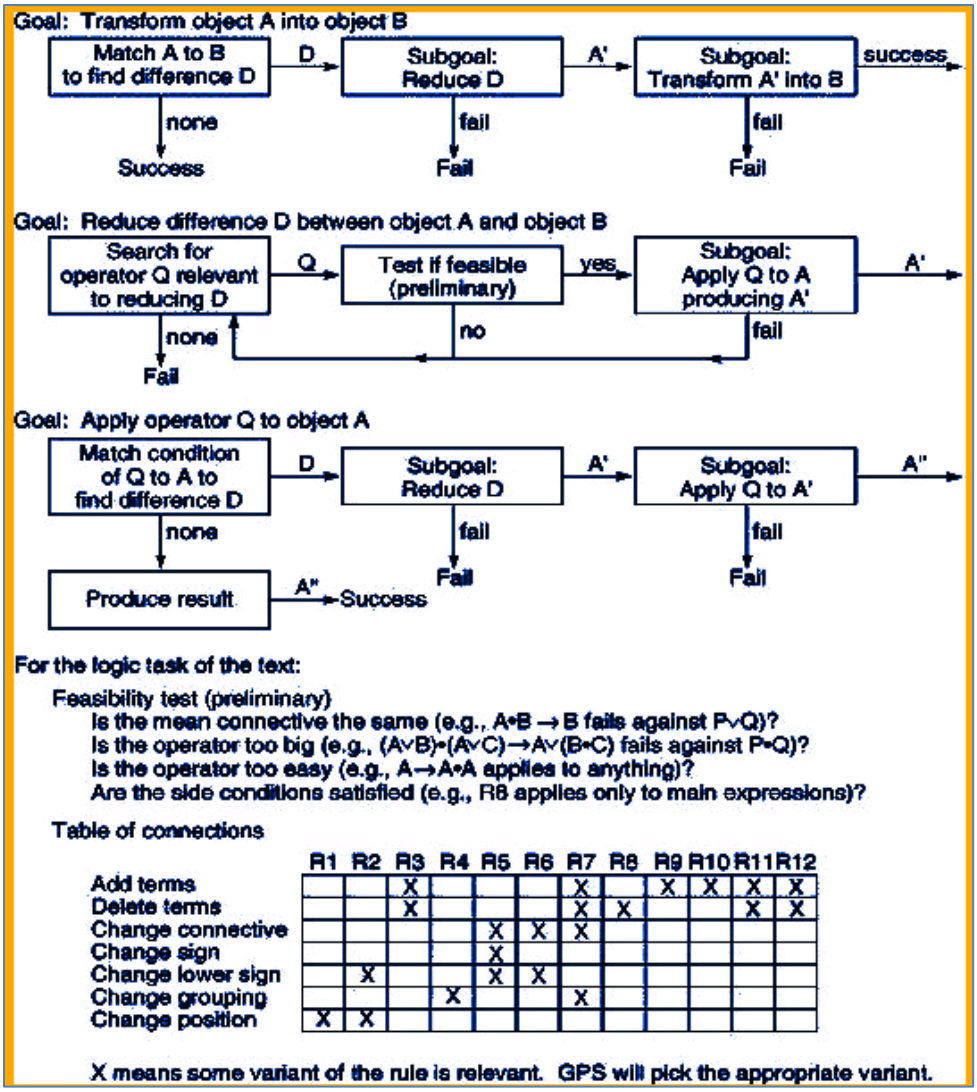

Figure 2-12 General Problem Solver flow chart [1]

In order to deal with real world problems where the amount of data causes the problem to be intractable, systems were developed that rely not on fully defined world states but rather on a space of partial plans. These can be searched concurrently and combined as the process moves towards the goal state. To search the space of partially ordered plans one popular solution is to use locally available options. This is most effective when strong heuristics are available to provide the right choices. In the absence of these heuristics failure is likely, unless backtracking is implemented to try other choices. NONLIN is a program that used this from of backtracking in the case where the choice provided by the heuristic failed. In turn it would be possible to try all of the local choices before backtracking to a higher level and starting on a new branch of the search space, resulting in potentially wasted computation [28].

DENDRAL was a knowledge based system that specialized in the discovery of molecular structure from mass spectrometer data. The operator would key in the chemical formula (for example $C_6H_{13}NO_2$) for a substance and then provide the mass spectrometer data to DENDRAL. Early versions of DENDRAL would examine a spectrometer data peak and then generate all possible structures consistent with the operator provided formula. For large molecular formula this quickly became an intractable problem. In later versions the designers of DENDRAL observed that analytical chemists were using well-known patterns to map peaks in the spectrometer data to molecular fragments. Once these 'cookbook recipes' were added to DENDRAL the search space was dramatically reduced and the system became extremely accurate in predicating molecular structure [25].

In cases where previously solved problems are encountered multiple times, a system can benefit from routine planning and design [58]. In such a system, prior knowledge is maintained and re-applied to new situations where the same or similar problems are encountered. COMIX is a routine design system that can be used by sales people to build industrial mixers based on requirements from the customer [22]. The design process in the case of COMIX is accomplished in two stages. In the first stage the specification of the machine to be built is derived based on one of two types of mixing tasks. Details of the material such as viscosity, density etc. that is to be used by the machine has to be provided by the user. Once this major classification is completed, the detailed mechanical design of the machine to be constructed is determined. This is carried out via the process of constraint propagation. The four main constraints in COMIX are:

1. Directed: The values of all but one parameter have to be known. The values of the known parameters are explicitly stated.
2. Simple: There is only one possible value, and the parameter to be determined is explicitly stated.
3. Functional: The definition of the constraint relation is given by a function and the preconditions that have to be satisfied are explicitly stated.
4. Constructive: Given all but one of the values of the constraint parameters the missing value can be determined and the function to evaluate is provided.

Since some decisions have to be made under uncertainty, and as is the case with constraint propagation, a solution may not be found. In this case backtracking has to occur and new avenues for the solution explored. Furthermore some decisions can only be made after all

the components for the machine to be built are known. One such case is if the revolutions of the industrial mixers are too close to the natural oscillation frequency of the completed machine. In this case knowledge-guided backtracking is employed which is based on the experience of the engineers who had modified properties of the machine to either change the natural oscillation frequency or change the number of revolutions. COMIX provides the user with all the scenarios where success can occur and the choice of which scenario to choose is left up to the user [55].

The key characteristic of routine planning systems is that the structure of the object or goal under design is known. All the components and the sub-goals and intermediate steps to the goal are also known. Furthermore, knowledge about failures and how to handle (back-track from) failures is available.

Another system that benefits from these characteristics of Routine Planning and Design is MPA, which is designed to help the Air Force with tactical mission planning using aircraft for tactical strikes. MPA functions using a high level language called DSPL (Design Specialists and Plans Language) which represents the routine design activity and the types of knowledge available to routine design systems. This includes complete knowledge of the components of the device to be designed along with their attributes. Complete knowledge of the design actions in the form of plan fragments is also available. The approach taken by MPA is to view the planning of the mission as an abstract device to be designed [58].

Routine design tasks are broken down into a hierarchy of planning tasks where the higher level decisions are made early on and the specific details are tackled later on. DSPL uses the concept of a *specialist* to solve the problems in each specific stage of the design. Specialists higher up in the hierarchy deal with the general aspects of the device being designed, whereas specialists lower in the hierarchy design more specific sub-portions of the device. Each specialist has access to local knowledge that it utilizes to accomplish its portion of the design. There is a variety of knowledge represented within each specialist:

i. Explicit design plans that encode sequences of possible actions to complete a specific task.
ii. Each design plan is associated with a *design plan sponsor* which determines the appropriateness of the plan given the current context.
iii. A *design plan selector* which functions to select the most appropriate plan for the current context based on the choices provided by the design plan sponsor.
iv. Constraints, which decide the suitability of incoming requirements and data and capture the knowledge that define what conditions must be true

for the specialist to have completed its task. Others verify the correctness of intermediate design decisions.

As the plan is executed in the DSPL system, control moves from the top-most specialist in the hierarchy to the lowest. Each specialist selects a design plan appropriate to the current context, and executes it. The relevant design actions are performed including any computation and value assignments. Constraints are run to check the progress of the design and if needed sub-specialists are delegated additional tasks and control passed to them. DSPL also includes the ability to handle various types of plan failures in which case control is passed to specialists responsible for those plans. The MPA system described here is designed for one kind of mission: an airstrike against an enemy airbase, known as Offensive Counter-Air (OCA) [58].

This system contains six specialists with the top most being the OCA. Next are the specialists that handle which base to use and which type of aircraft. The bottom tier consists of the specialists that handle configuration of the specific aircraft chosen by the aircraft specialist. Figure 2-13 shows this hierarchy.

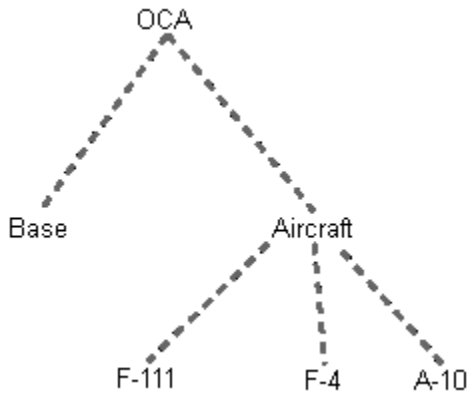

Figure 2-13 The MPA hierarchy of specialists [85]

The mission planning begins when the OCA specialist is requested to plan the mission. The OCA contains a single design plan which requests the base specialist to determine the base and the aircraft specialist to choose an aircraft. The base specialist selects a base from a list of bases known to the system based on location from the target. The aircraft specialist chooses an aircraft based on the threat types of the target and the weather along with time of day considerations. The vast majority of the planning is done within the aircraft specialist. The specialist contains within its context the threat level of the target and environmental conditions such as the weather and time of day. The plan sponsor for the aircraft is then run to assess the suitability of the aircraft. For example one aircraft may be chosen over the other because the mission requires night time flight and only one aircraft of the three has that capability. Once the

sponsors have run and the suitability of each aircraft is known, the plan selector returns the name of the plan to the specialist which then executes the plan. In some cases when no plan is suitable the plan selector can return a NO-PLANS-APPLICABLE primitive to the specialist. Figure 2-14 shows the DSPL code for plan sponsor for an aircraft also the DSPL code for the plan.

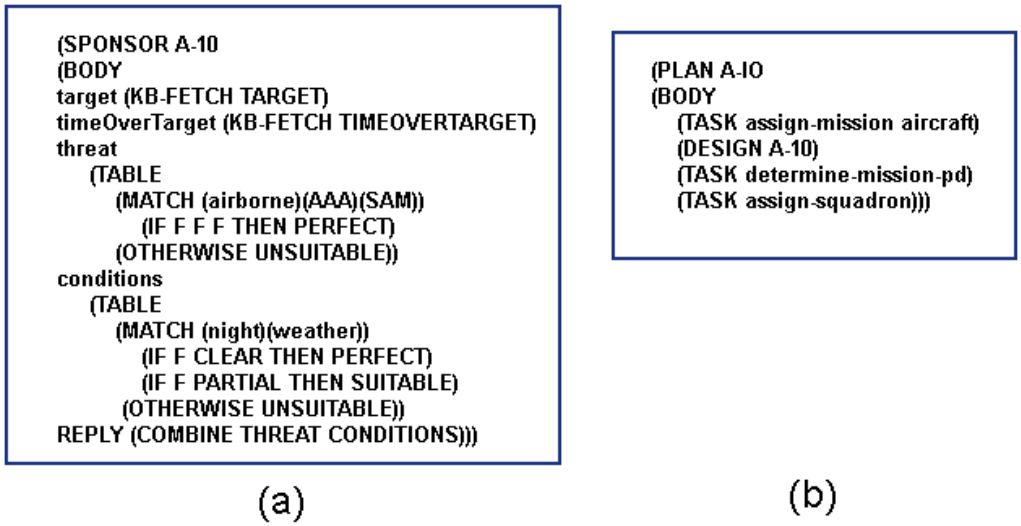

Figure 2-14 DSPL code for (a) Design Plan Sponsor and (b) Design Plan [85]

As can be seen in the figure, the sponsor determines the suitability of the aircraft based on the threat levels, the weather and time of day. In this case the sponsor is for the *A-10* aircraft. The target and the time for the attack are input by the user. The database is then looked up to see if the threat at the target matches the conditions of *airborne, AAA and SAM*. If these conditions are all met then the selection is deemed to be *perfect* otherwise the response is *unsuitable.* Similarly the database is consulted for the conditions of the mission and time of day. As long as all conditions that are suitable for the A-10 aircraft are matched then the aircraft sponsor deems the choice to be suitable, otherwise it is deemed unsuitable. The plan is then executed and the aircraft is assigned to the mission, its payload determined and finally assigned to a squadron. Finally, Figure 2-15 shows the design plan selector for the aircraft with the possibility of a failure resulting in a NO-PLANS-APPLICABLE response.

```
(SELECTOR AircraftSelector
(BODY
REPLY (IF (MEMBER A-IO SUITABLE-PLANS)
  THEN A-10
  ELSE IF (MEMBER F-4 SUITABLE-PLANS)
    THEN F-4
    ELSEIF (MEMBER F-II1 SUITABLE-PLANS)
    THEN F-II1
    ELSE NO-PLANS-APPLICABLE)))
```

Figure 2-15 DSPL code for Design Plan Selector [58]

Attempts have been made to overcome the limitation of planning systems to cope with changing environments. One way to accomplish this is to have a series of agents that are implemented in layers which communicate with each other. Each layer is responsible for handling a specific stimulus from the environment. By the combination of these layers the overall robot is able to accomplish tasks such as wandering a room and building an internal map [27]. Other systems utilize Reactive Planning to alter behavior based on changes in the environment. PRS (Procedural Reasoning System) is a reactive planning system designed to operate a robot assistant. The systems designers envision PRS to serve as a mechanical assistant to astronauts on a space station. PRS is composed of a database containing 'beliefs' or facts about the world in which it operates, a set of 'goals' to be realized and set of procedures, known as Knowledge Areas (KA), that describe sequences on how to achieve goals and react to particular situations. Figure 2-16 shows the schematic of the PRS system. The database consists of pre-programmed beliefs such as the layout of the rooms in which the robot has to operate. Additional beliefs are derived as plans are executed which in turn cause changes in the environment. The goal stack contains current goals of the systems and each goal is limited by time.

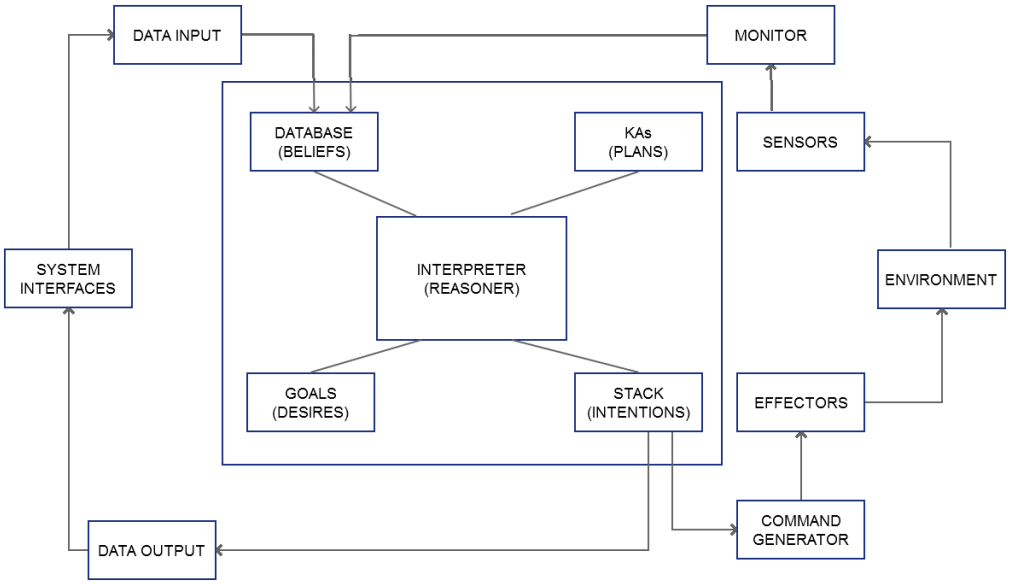

Figure 2-16 PRS system diagram [56]

The knowledge areas specify procedures on how to accomplish a sequence of steps and are paired with an invocation condition which describes under what situations the procedure is useful based on current goals and/or beliefs. The procedure body itself is comprised of sub-goals similar to operators in traditional planning systems. The system interpreter ties all the subsystems together. At any given time a set of goals and a set of beliefs are active in the system. Based on these active goals and beliefs a subset of the KAs will be relevant. One of these KAs is chosen and placed on the process stack. As the KA is executed new beliefs are derived and added to the belief database and new sub-goals added to the goal stack which in turn activates other KAs. At any time during this process if some new fact becomes known or some new input arrives from the sensors, PRS will re-evaluate its goals and hence is able to react to changes quickly. It even has the ability to abandon its planning procedure by deciding there is not enough time to reason and instead it is time to start acting [56].

We have seen in this section various planning systems from simple linear systems to complex hierarchical, partial planning systems and reactive planning systems. The representation of the real world within these systems can only be successful to a point due to the exponential growth in complexity when trying to model every facet of a problem. As a result it is common to deal with search spaces that are too large to provide solutions in polynomial time. Furthermore, planning systems work primarily by following a narrow path through the search space based on knowledge encoded into it and a finite set of rules. These solutions have a rigidity to them that works as long as the search space remains static.

2.1.3. Genetic Algorithms

In dynamic environments, rules (operators) cannot cover every possible situation that may arise, thus planning systems cannot be dependent upon in such cases. Genetic Algorithms (GA) were developed as a result of looking for solutions to very large search spaces that are not necessarily static. Observations of biological entities that evolve over time to cope with changing environments provide a model to build solutions to problem domains that are not precisely defined and are inclined to change [30].

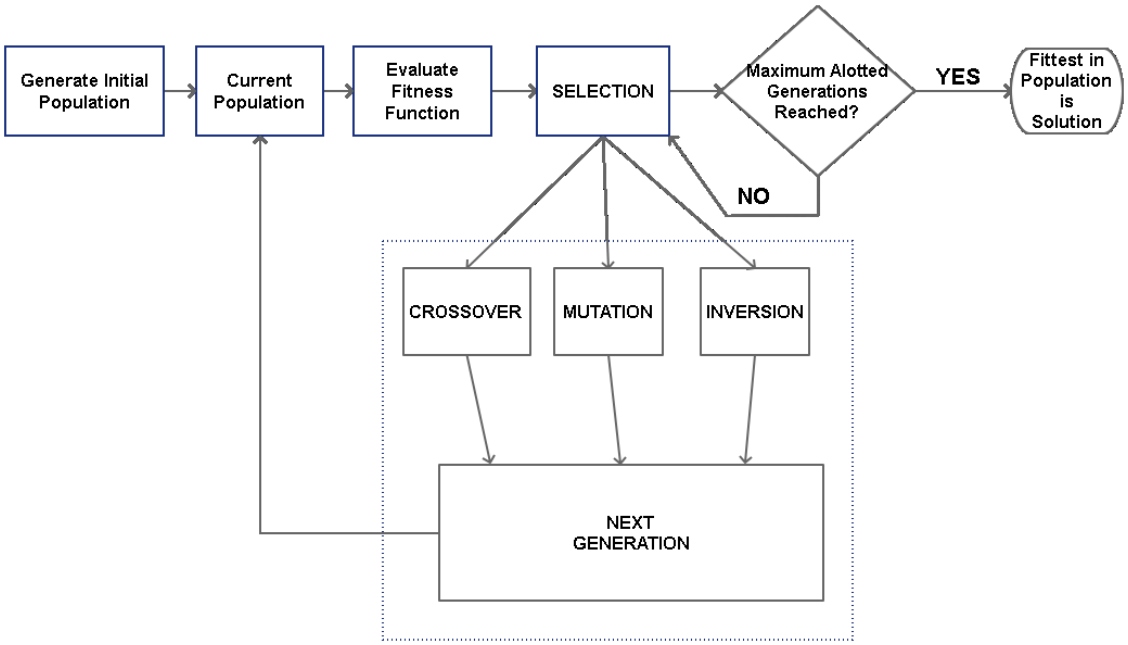

Figure 2-17 Flow of a genetic algorithm

Genetic Algorithms are loosely based on the Darwinian theory of evolution. Organisms evolve as a result of two main processes: natural selection and sexual reproduction. Natural selection determines which members of a population are fit enough to survive and sexually reproduce which in turns allows the mixing and recombination of the genes in the offspring. By having imprecise copying of the genes from one generation to another, mutations are introduced into process. If these mutations are beneficial they are passed onto the newer generations. Emulating this straightforward process within the context of software design provides an avenue for evolving solutions to a variety of problems [31].

In order for Genetic Algorithms to be effective, all of the ingredients found in nature have to be present. An initial population has to be created with members whose encoding represents

the solution they will evolve into. The initial population can be created by random assignment or some other heuristic. There needs to be mechanism to allow the members of this population to combine their solutions to produce new solutions. Finally, and perhaps most critically there is the need for a *Fitness Function* that evaluates each member of the population so that selection can take place to form a new generation.

To utilize the GA approach, one needs to define the representation by which each individual is to be described. This representation will store, for each characteristic or trait of the individual, the value for that characteristic or trait. We call this representation a *chromosome*. Each attribute of the chromosome must have a defined range of permissible values. For instance, a binary attribute would store 0 or 1 for the absence or presence of that attribute. In other cases, the attribute might be defined by integer number, real number or even string. The following represents a binary chromosome.

0	1	1	0	1	0	1	0	1	0

What the binary values represent is defined by the context of the problem being solved. For instance in the case where we are trying to find the largest subset sum of 10 numbers the 1's in the string can represent that the integer numbers at the corresponding indices be added to the sum and the 0's represent that the integer numbers at the corresponding indices be excluded from the sum. Binary encoding however is not always the most convenient representation of the values of the problem domain. Other possibilities include integers or real numbers representing actual solution values. Text encoded values could represent a list of rules for example:

3	1	7	0	11	20	7	15	1	9

0.123	1.33	1.56	0.77	3.1	9.1	11.0	13.0	19.45	7.44

R1	R2	R4	R17	R11	R33	R3	R5	R9

The overall flow of a genetic algorithm is shown in Figure 2-17. The first step is the creation of the initial population of some specific size with each individual of the same length. Each generation that is evolved from this initial population will have the same population size. Individuals in this initial population are combined using the crossover operator. This operator takes two individuals and, at a pre-determined or random locus, splits each of the chromosomes

into two fragments. These fragments of the two individuals are now exchanged creating two children that are a mix of their parents. For example if the two parents are:

A	B	C	D	E

and

P	Q	R	S	T

The crossover operator can result in two children such as:

A	B	R	S	T

and

P	Q	C	D	E

In addition to the crossover operator, operators of mutation and inversion can be applied to create more variety. Mutation operators randomly change a value encoded in the chromosome such as changing a 0 to 1 or a 3 to a 9, whereas Inversion can invert or otherwise rearrange a sequence of values such as changing the sequence 1,1,0,0 to 0,0,1,1. Mutation may not be applicable in some cases where a random change can lead to an illegal entry such as if the possible values for an attribute are 1-4 but a 3 is changed to a 9.

The result of these operators is a new generation. Each member of this new population is evaluated by the Fitness Function to be assigned a score. This provides the worth of each item in the population. That is, the fitness function is an estimate of how likely the given individual in the population might lead to a solution. The fitness function is designed to analyze each chromosome in the population and, based on the values that comprise the chromosome, assign a score to it.

For example consider a simple version of the knapsack problem where we want to find the largest number of items (assuming all have equal weights and values) we can put in the knapsack without exceeding its capacity. A binary encoded string can represent the chromosome. 1's can represent an item in the knapsack and 0's the absence of items. The fitness function in this case would simply count the number of 1's and as long as they don't exceed the capacity for the knapsack, the higher the count the more fit will be the evaluation. In a more complicated version of the knapsack problem, where the items have different values and weights assigned, the chromosomes can be encoded as integer pairs representing the value and the weight of the items placed in the knapsack. The fitness function would now sum up the integer values and weights to get the fitness of the chromosome. The higher values would be

more fit as long as their weights didn't exceed the capacity of the knapsack. For more complex encodings of the chromosome more elaborate fitness functions are likely to be required. In many cases multiple fitness functions with different weights are combined to form the final evaluation score.

Once all of the individuals of a population have been evaluated by the fitness function and are assigned a score, selection of individuals can take place. Typically a mix of highest ranking and most diverse members of the population are selected to be the parents of the next generation.

The combination of these genetic operators is applied in such a way that a new population is created with the same number of individuals as the previous population. The selection of the individuals to reach the next generation will create a parent population equal to the previous number of parents. The evolutionary process continues until either a pre-determined number of generations have occurred, or until some threshold fitness value is achieved in at least one member of the population.

Different problems call for different population sizes and can be influenced by available resources and computational requirements as well [1]. To better illustrate the working of a Genetic Algorithm we briefly describe its implementation when solving a classic computer science problem: the *Travelling Salesman Problem.*

The *Travelling Salesman Problem* (TSP) is described in the following way: A salesperson is required to visit N cities as part of a sales route. There is a cost in fuel, mileage etc. associated with each pair of cities on the route. The goal is to find the least cost path through all the cities that starts at one city, visits all other cities, and returns to the starting city, visiting all the cities exactly once. The full state search space requires evaluation of N! states (where N is the number of cities) and the problem has shown to be NP-hard. Beyond the obvious cases for routing of packages and travel applications such as mapping, TSP solutions minimizing the movements of machinery have obvious cost benefits [1].

The first consideration for using a GA to solve this problem is the method of encoding the path for the cities visited. One method is to encode the cities as integers. So given a set of 9 cities they can be encoded simply as (1,2,3,4,5,6,7,8,9). Figure 2.18 shows the partial collection of edges between the 9 cities. For this problem we assume there is a path from each city to every other city forming a complete graph with N(N-1)/2=36 edges .

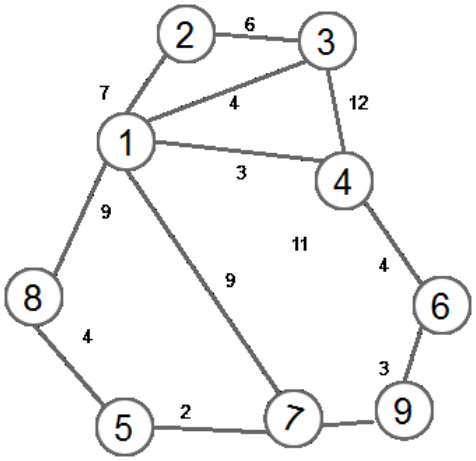

Figure 2-18 Travelling salesperson problem for 9 cities (subset of all edges shown)

The fitness function in this case is relatively simple. It is the sum of the costs of travel between the consecutive cities. In order to preserve the order of cities that produce the lowest cost embedded within the sequence, a modification is performed to the standard crossover operator. This modification involves using the same cut points at random between all parents so that the same size sequences are chosen for crossover, this is referred to as the *order crossover* operator. This is done by choosing a subsequence of cities within the path of one parent. Then the same sized subsequence is used in all parents which are to be involved in the cross over. So for example two parents P1 and P2 will have the same randomly chosen cut points after the third and seventh cities:

P1 = (1,9,2|4,6,5,7|8,3)
P2 = (4,5,9|1,8,7,6|2,3)

The two children are chosen which have the same size subsequence between the first and second cut points:

C1 = (x,x,x|4,6,5,7|x,x)
C2 = (x,x,x|1,8,7,6|x,x)

Next the sequence from the opposite parent is used to fill in the new sequence on the child, ignoring the cities from the opposite parent that are already in the new child. So C1 becomes:

C1 = (2,3,9|4,6,5,7|x,x)

Finally the remaining cities are added so we get the final child C1:

C1 = (2,3,9|4,6,5,7|1,8)

The same procedure applied to C2 gives us:

C2=(3,9,2|1,8,7,6|4,5)

Other operators of inversion can be applied as long as they invert a fixed sequence so for example C2 can be inverted to

C2 = (3,9,2|6,7,8,1|4,5)

Once the algorithm runs through its specified iterations (generations) the final result may be close to the optimal solution with the sequence representing the least costly path through the cities.

The application of GAs to TSP in this solution assumes that all cities have a path connecting them. However if the TSP solution required for a problem were one where all cities were not connected, this approach would not work since the random assignment of cities in the initial population may contain pairs that aren't connected to each other as could children generated by mutation or crossover. Additional heuristics would have to be designed to prevent this from occurring. For example, applying mutation to a chromosome such as (1,2,3,7,8,4,5,9,6) might generation (1,2,3,6,8,4,5,9,6) which results in would cause the same city, 6, to be included twice. One possible way to handle mutation would be to randomly swap the position of two cities, thereby guaranteeing no duplication or loss of city from the solution.

As mentioned earlier, the representation of the problem domain and the design of the fitness function are the major challenges in using GAs. Other challenges include the selection of the size of the population. The population size represents the number of states of the search space examined at a time. If the population is too small, the algorithm may not explore enough of the search space. If the population is too large then efficiency issues arise. Mutation and Inversion operators have to be judiciously applied. If these operators are too aggressive, then the rate of 'genetic change' will be too high and possible solutions will not have a chance to

emerge in the generations. If these operators are applied too infrequently, then the search may not branch out enough to cover a wide range of possible solutions. Finally, problems that are readily solved using traditional analytical methods are poor candidates for genetic algorithms. This is because GAs don't provide exact answers whereas analytical methods do and they do so in a more efficient manner [20].

This concludes our examination of specific AI techniques that will be used in generating music for this thesis. We next examine specific music composition systems that have been used through the history of AI.

2.2. Survey of Music Systems

As mentioned in Chapter 1, music and computers have a long history together. In this section we will highlight the works of many researchers who have approached this challenge from different angles and for different purposes. Computers and AI methodologies have been used to generate music (composition), to play music along with and in response to human performers (improvisational) and to emulate the nuances of human performance (performance systems). Since this thesis implements a music composition system, we discuss similar systems in the following subsections.

2.2.1. Stochastic Systems

No discussion of music generation using computers can begin without mentioning the pioneering work done in the *Illiac Suite for String Quartet* created by Hiller and Isaacson in 1950s [3]. The system generated notes pseudo-randomly through the use of Markov chains. The Markov chains comprised notes which were generated using a series of harmonic and proximity functions. These in turn allowed the operators to control the size of the steps and skips in the notes along with the proportion of the intervals and ultimately the consonant vs. dissonant nature of the music produced. For example the Interval of a major third was assigned the number '3'. The harmonic function then gives the value of '7' for this input and the proximity function gives the value of '10'. The Markov chain was constructed with these values for all the intervals from unison (0) to octave (12) and the likelihood of one note appearing after another was dictated by the Markov chain. The generated work was then tested by applying heuristics and compositional rules of harmony and counter-point. If a generated note or interval was found to violate the rules of counterpoint it was rejected. The printed output was then performed by

human performers. The goal of the system was to prove that music that is listenable can be generated this way without regard to expressiveness or emotional context.

Another important pioneer in music composition via stochastic methods was Iannis Xenakis who built a system to generate data for statistical analysis as described in his book *Formalized Music* [34]. The system used digital signal processing and the manipulation of sound waves to achieve musical notes. All the sounds were computed from wave form samples which were then modified using random walks. The random walks were contained by thresholds hardcoded by Xenakis based on what he felt was reasonable. The culmination of his decades of work resulted in *GENDY3* in 1991. GENDY3 and its forerunners produce sounds that most would not call music but were important steps in proving that stochastic methods via computers could create new and interesting output.

Another system *Stochos* continues upon the work of Xenakis and synthesizes real time music based on several stochastic functions. The music is generated via raw sound waves and signal processing techniques. An event in Stochos is defined by its onset time and its duration. The event contains information on timbre, intensity and pitch and any other attributes assigned by the user. Events can vary in duration from fractions of a second to several seconds long. Multiple events combine to form a cell. The parameters of a cell such as its mean duration and density value are defined by the user. This combination of duration and density value of the cell determines how many events are to be included in the cell.

Stochos can take input from a live source such as a synthesizer or operate on pre-existing wave sound samples. As shown in Figure 2-19, this input is passed to the cross synthesis module where pitch and amplitude envelopes are extracted

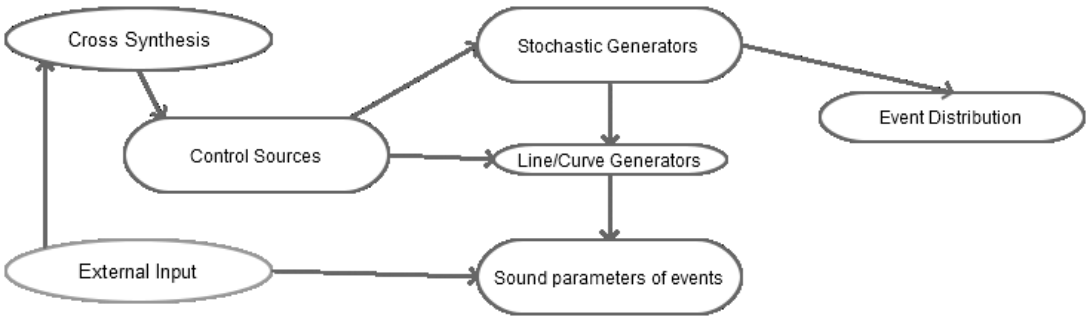

Figure 2-19 Architecture of Stochos [35]

These are then passed to the control sources module where other user specified parameters are applied to the sound samples. These modified sound samples are then passed

to stochastic generators where different stochastic functions are applied to the events within the cell to change their distribution in time by modifying their density and probability distribution. The system implements eight stochastic functions [35]:

1. Exponential: This is an exponential distribution found in statistics.
2. Linear: A simple linear distribution.
3. Uniform: All events have the same probability within the cell.
4. Gauss: Applies Gaussian or Normal distribution to the events.
5. Cauchy: Applies the Cauchy distribution to the events.
6. Weibull: Applies the Weibull distribution to the events.
7. Logistic Map: This is a chaotic generator and is applied to every new event within the cell.
8. Constant: A constant value is applied to each event such as duration, on set time etc.

Closely related to the stochastic approach using Markov chains is the method of using *Cellular Automata.* Cellular Automata are discrete systems where some feature is changed with time. Even though there are rules governing what can change from one time step to the other, the overall direction of the automata is unknown in advance. As a result the phenomena of *Emergent Behavior* is observed, which was first described by Conway in his *Game of Life* [32]. Each square in the grid can be either occupied or empty. The occupied squares show up as black and the empty ones as white. Three simple rules drive the state of each square over time:

1. Any square that has exactly three of its neighbors occupied will be occupied in the next time interval.
2. If a square that is occupied and has two of its neighbors occupied will be occupied in the next time interval.
3. Any other situation the square will not be occupied.

Figure 2-20 shows how the patterns change with the application of these rules in 5 discrete time intervals in a 6x6 grid.

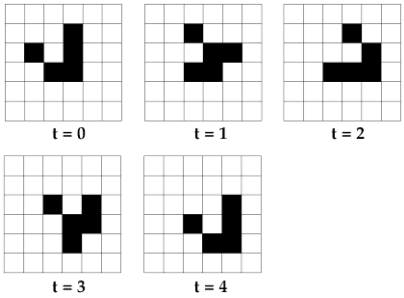

Figure 2-20 Game of life over 5 intervals [57]

On a larger grid over many iterations a pattern shown in Figure 2-21 might be observed [1, 36].

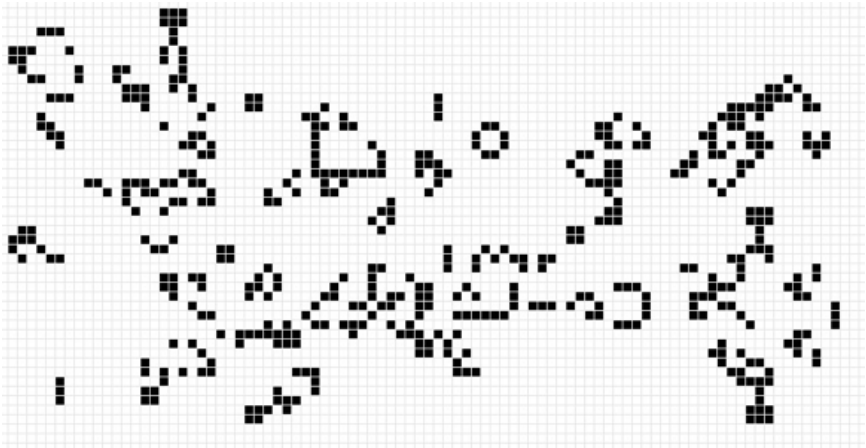

Figure 2-21 Patterns emerging in the Game of Life

Another similar construct using cellular automata is the Demon Cyclic Space where an array of cells can exist in n states. The evolution rule applied to the Demon Cyclic Space is that at each time interval each cell will dominate any neighboring cells whose state is exactly 1 less than the current cell. If the state of the current cell is j and the neighboring cell(s) state is j-1, then in the next time step the state of the neighboring cells will be j. Furthermore the space is cyclic so the cells in state 0 will dominate the cells in state n-1.

Based on these two constructs of the Game of Life and the Demon Cyclic Space, the *CAMUS 3D* (Cellular Automata MUSic in 3 Dimensions) system was developed. Just as patterns propagate through the grid in these cellular automata, so is the composition created in CAMUS 3D. Based on the rules defined by the composer the patterns of repetition, transposition, inversion etc. are generated through the system and the music piece created. Any cell that is not empty is considered ‘live’ and constitutes a musical event. The music generated

from CAMUS 3D centers around a four note musical event which is a result of its algorithm design [33]. Figure 2-22 shows the configuration for a typical time step in *CAMUS 3D* and the mapping that it employs converting cell data to music.

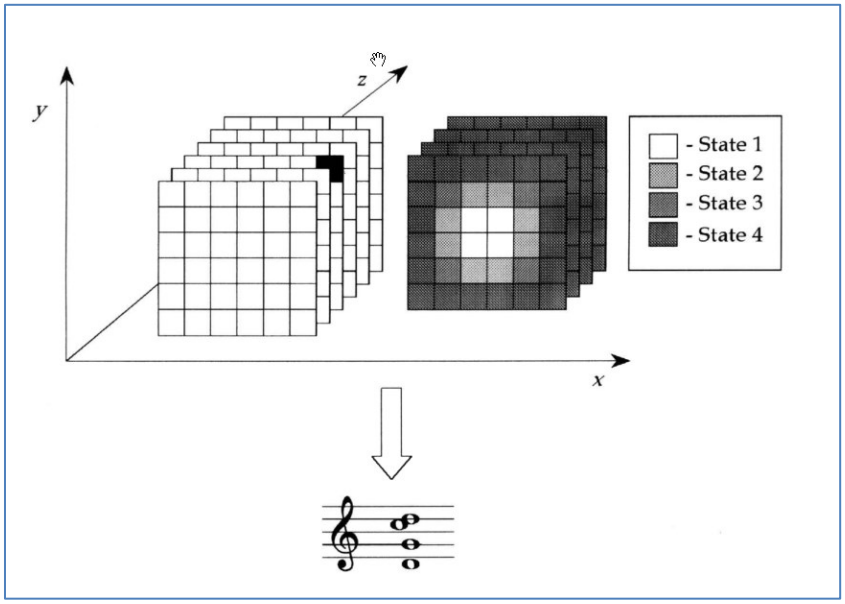

Figure 2-22 CAMUS 3D cell data mapping to music

As can be seen in the figure each live cell represents a coordinate in three dimensional space. These coordinates are in turn mapped to intervals in a four-note chord. The x coordinate represents the semitone interval from the root pitch to the second lowest pitch in the chord. The y coordinate represents the semitone interval from the second lowest to the second highest and finally the z coordinate represents the semitone interval from the second highest pitch to the highest pitch. For example, if the root of the chord is selected to be 'C' and if the coordinates of the cell are (2,2,2). Then the chord played will be C+D+E+F#. Where D,E and F# are 2 semitones or one whole tone apart from their previous note.

The live cell in the figure has the coordinates of (5, 5 ,2) and the corresponding state in the Demon Cyclic space is 4. The root pitch of the chord is mapped to a value using user preferences or randomly selected. Once the root pitch is determined the other keys of the chord are then mapped out from the intervals provided by the coordinates. The Demon Cyclic Space state determines the MIDI instrument used to play this chord. In this example it would use the instrument mapped to MIDI channel 4.

2.2.2. Knowledge Based Systems

As stochastic systems were the easiest to implement [33], it is understandable why they were the first systems developed. With the rise of *rule-based* and *formal-grammar* systems the application of these systems to music composition also increased. Instead of delegating musical events to chance as in stochastic systems, rule-based systems pre enumerate the choices available to the compositional system. As described in Section 2.1.2, planning systems work well when the system they are simulating can be described and limited to specific domain. Even though music in itself is a very large domain, there are plenty of sub genres that can provide small enough knowledge domains that can then be modeled within a planning system. Thus rather than relying on musical models based on statistics, these systems explicitly define music models that their authors desire to implement.

MUSICOMP was one of the first systems built following this approach. Implemented as a system of subroutines it generated the work *Computer Cantata.* The system was implemented as a library of sub routines so that the rules embedded within the subroutines could be combined to generate larger and more varied pieces of music [37].

Moorer's experiment in tonal and atonal music is another example of planning decomposition. In this system the overall form of the music piece is chosen first, then the chord sequence and finally the melody sequence. The probabilities of which sequences are chosen are 'tuned' by the operator. For example, the overall song structure is decided by the system choosing two numbers that are constrained to be a power of two. The first is the number of major groups and the second the number of minor groups. A third number is then chosen to be the number of beats per minor group. This number is constrained to be a power of two times the number of beats per measure. The total piece length is then given by the formula:

Num of Major Groups x Num of Minor Groups x (Num of Beats/Num of Minor Groups)

This value is then compared to the value provided by the user for the limit of the total size of the piece. If this falls within the limit specified by the user it is accepted, otherwise rejected and the process is repeated with a new set of numbers until a result is achieved that matches the user's specification. Similar procedures are carried out for the chord and melody selections using a combination of probabilities and heuristics. The chords for example are chosen by first generating a hypothesis using first-order probabilities. These simply state the probability of one chord occurring after another. Heuristics then eliminate sequence that are too dull or violate music theory rules. The resulting chord sequence is then represented with one letter for each

beat in the piece. Major groups are separated by spaces and minor groups by periods. So a sixteen beat piece can be presented as: CCCC.FFFF CCCC.GGGG. Where C,F,G are the major chords and each instance presents a beat. The representation also shows two major groups and two minor groups within each of the major groups [39].

CHORAL is another expert system developed to harmonize the chorales in the style of J.S.Bach [19]. Given a melody, CHORAL produces harmonization using a heuristic of rules and constraints. It was implemented using *BSL*, a logic programming language that could represent musical knowledge as first order predicate calculus. The system itself is implemented as rules relating to particular views:

- *Chord Skeleton View:* Observes the chorale as a sequence of rhythm-less chords with symbols designating key and degrees within a key.
- *Fill-In View:* Observes the chorale as four interacting automata that changes states and produces notes based on the underlying chord skeleton
- *Time-Slice View:* Observes the chorale as a sequence of vertical time slices each with a duration of eighth notes.
- *Melodic String View:* Observes the sequence of individual notes of the different voices from a melodic point of view.
- *Merged Melodic String View:* Similar to the Melodic String View except that the repeated adjacent pitches are merged into a single note.
- *Schenkerian Analysis View:* Observes the chorale as a sequence of steps of two non-deterministic bottom-up parsers for bass and supporting voices. Based on the theories of musical analysis by Heinrich Schenker.

These views and over 270 production rules based on empirical observations of Bach chorales and other intuitions of the authors of *CHORAL* are implemented using the BSL language built for this system. Figure 2-23 shows a snippet of the BSL code that implements the following rule:

> *"When two notes which have the same pitch name but different accidentals occur in two consecutive chords, but not in the same voice, then the second chord must sound the flattened third of the first chord"* [38].

This example shows the level of sophistication that can be achieved using rule based systems.

```
(A u bass (<= u soprano) (1+ u)
        (A v bass (<= v soprano) (1+ v)
                (imp (and (> n 0)
                        (== (mod (p1 u) 7) (mod (p0 v) 7))
                        (!= (a1 u) (a0 v))
                        (!= u v))
                (and (member chordtype0 (dimseventh domseventh1))
                (or (and (==(a0 v) (1+ (a1 u)))
                        (== v bass)
                        (== (mod (- (p0 v) root1) 7) fifth)
                (and(==(a0 v) (1- (a1 u)))
                        (= = v soprano)
                        (== (mod (- (p0 v) root1) 7)
                        third)))))))
```

Figure 2-23 Snippet of BSL code implementing a rule for notes [38]

2.2.3. Evolutionary and Learning Systems

Beyond stochastic and knowledge based systems, the third major category of systems is designed around machine learning. With the resurgence of artificial neural networks in the 1980s many compositional systems were designed to capture the rules for music embedded within the music itself. One of the driving forces for such systems is the belief that musical taste is largely cultural based, and that preferences for musical styles must therefore have been learnt by the listener over time. This led to a considerable amount of research in neural networks that are trained to deduce musical regularities based on the types of music that are input into such systems [40].

MUSACT is a neural network based music generation system that is able to learn different models of musical harmony. For example, one of the qualities of a dominant chord is that it creates in the listener an expectation that a tonic chord is about to be heard. Based on the musical samples that *MUSACT* is trained on, it is able to generate new music with this type of quality [40]. *HARMONET* approaches the harmonization problem using a combination of neural networks and constraint-satisfaction techniques. The neural network learns the harmonic functions of the chords and the constraints are used to fill the inner voices of the chords [42]. *MELONET* extends the work of *HARMONET* and uses the neural net to learn and produce higher level structures in melodic sequences. Using a melody as input *MELONET* is able to invent a baroque-style harmonization. Of special note is that combined together *MELONET* and

HARMONET are able to produce variations whose quality is similar to that of experienced musicians of that genre.

MELONET II is a system that uses a combination of neural networks operating at different time scales to recognize, propose and generate new music [41]. The system is designed with the observation that even simple music like folk songs are organized in a hierarchical manner and each component of the song operates at different time scales such as measures, phrases, motifs. Several neural networks are combined to classify and learn the attributes of the song at the varying times scales. A combination of supervised and unsupervised learning is used. The supervised learning is used for the prediction in time of the notes whereas the unsupervised learning recognizes and classifies musical structure. This system specializes in composing folk music. By giving it a short beginning of a melody, *MELONET II* invents a folk song type continuation and harmonization. The first step in the system is to make a motif 'proposal' based on the motif context of the input song. The note predication network then makes its proposal based on the input from the motif network and some additional user specifications.

Figure 2-24 shows the overall structure of *MELONET II.* The system was trained using 20 folk songs represented as notes of melodies. The neural networks used were of feed-forward type with the topology of input-hidden-output and two layers comprising the hidden layer. The experiments conducted by the authors claim that the melodies produced by the system have considerably better coherence compared to earlier systems that operate at smaller time scales [41].

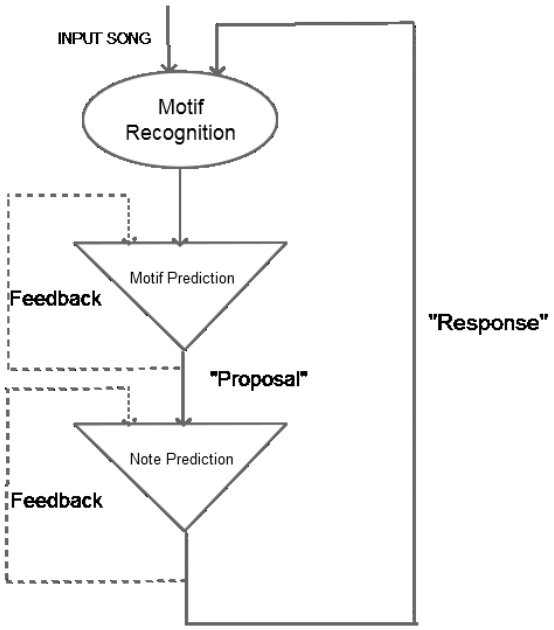

Figure 2-24 Melonet II Structure [41]

So far we have seen examples of systems for composing music. Machine learning and evolutionary AI based systems have been extensively used for improvisation and performance emulation as well. Specifically in the case of performance, the nuances of a musician's technique are difficult to describe in rules. In such cases, machine learning systems such as Neural Nets and *Case Based Reasoning* (CBR) are more effective [1].

SaxEx is a CBR based system whose goal is to add expressiveness to a flat performance. CBR systems use databases of problem solutions to solve new problems. These databases are built with data from human experts. In addition they store the results of previous attempts when they are successful, enabling CBR systems to acquire new knowledge over time. Common to all CBR systems are four steps in solving a problem. The first is the retrieval of appropriate cases from memory. Heuristics are employed in choosing cases similar to the current problem being solved. For example in patient diagnosis systems this would involve determining the level of similarity of symptoms among the stored cases to the current patient. The second step is to modify a retrieved case and apply it to the current situation. Since not all cases match exactly, modification steps are executed in order to alter the parameters of the selected case to conform to the parameters of the current case. In the third step, the transformed case is applied to the current situation which results in either a successful solution or a failure. The Fourth and final step is to save the results of the application of the modified

case as a success or failure so that it becomes a part of the knowledge database of the CBR system [1].

As is key to all CBR systems, in SaxEx there is a memory module that stores the case base. Here, it compares performances of humans that SaxEx is to emulate. The system is also provided the score of the musical piece on which the performance expressiveness is to be applied. The *spectral modeling subsystem* (SMS) is tasked with computing for each of the expressive resources how each note in the performance has been played by a human performer and stores this value with each note in the *cases* database so that it may be called upon when generating the performance. A second layer of analysis based on the *Narmour* music cognition model [44] is applied to determine the role of the note in the musical phrase. This helps the system decide the importance of the note within the phrase, based on its location. For example a note may be the last note in an ascending melodic progression and should be the most prominent note in the measure. SaxEx uses the results of both layers of analysis to annotate the notes and this adds to the richness of the cases database in the system. When generating the expressive performance from the input inexpressive phrase, *SaxEx* searches for notes that have the same roles within the musical phrase and applies these attributes to the inexpressive notes and creating the resulting expressive performance [43]. Figure 2-25 shows the architectural layout of S*axEx.*

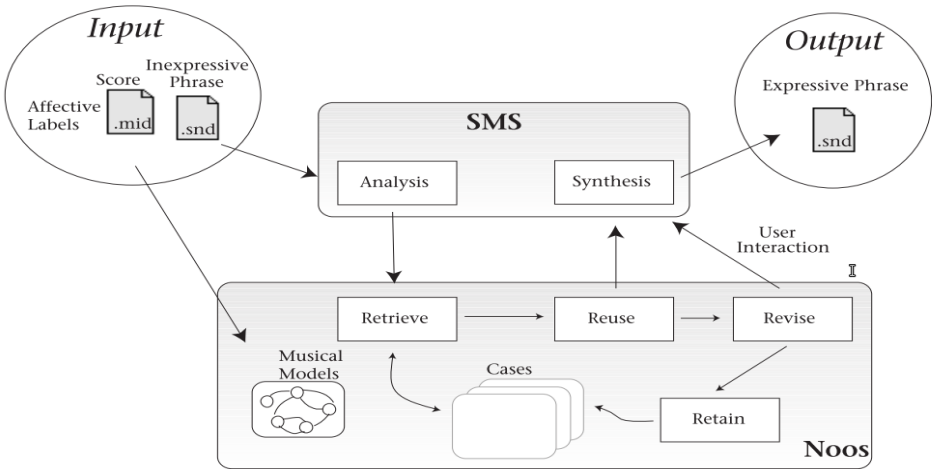

Figure 2-25 SaxEx Architecture [43]

The score of any new music to be performed expressively is input to *SaxEx.* Based on the context of the notes appearing in the score, the case database is looked up to find similar notes, in terms of the context, between the notes of the current score and the notes in the database. The expressive attributes from the case database are then applied to the notes of the score. In circumstances where exact matches aren't found, fuzzy aggregation by means of

weighted average is applied to the notes. The user then listens to the notes and either accepts them or applies modifications. In either case the accepted notes are added to the case database and provide a learning mechanism for the system. Finally the output is generated which contains expressive attributes to the original score [43].

If compositional systems are on one end of the spectrum and performance and performance systems on the opposite end, then we can visualize *Improvisational Systems* as somewhere in the middle. These systems work in real time with musical performers to generate the missing parts of an accompaniment. *GenJam* is a system that improvises jazz solos [45]. Genetic Algorithms are used to model a novice jazz musician learning to improvise. The system maintains hierarchically related populations of melodic ideas that are mapped to specific notes. *GenJam* plays its solos over the accompaniment of a standard rhythm section, while a human mentor gives real time feedback. This feedback is used to derive fitness functions that are used to breed new populations of solos.

The process of improvisation begins with *GenJam* reading a progression file. This provides the system with a tempo and rhythmic style, the number of solo courses it should take and the chord progression. It also inputs piano, bass and drum sequences from pre-generated MIDI files. A tune is improvised by building choruses of MIDI events that are decoded from populations of phrases and measures. The human mentor listens to these solos and on a keyboard and types ‘b’ for bad and ‘g’ for good based on how she judges the quality of the tune. The fitness for the given measure or phrase is accumulated by incrementing counters every time a ‘g’ is passed and decrementing the counters every time a ‘b’ is entered. These fitness values are then written back to the population of measures and phrases.

Three modes are available in GenJam: *Learning, Breeding and Demo.* In *Learning* mode, phrases are selected at random and presented to the mentor for feedback as described above. In *Breeding* mode, genetic operators are applied to the populations and half of them are replaced with the new offspring. *Demo* mode simply picks the most fit phrases and measure and plays them for the listener. No feedback is solicited in this mode.

For genetic algorithms the string representing a gene is of critical importance. In the case of *GenJam*, there are two types of string representations. As mentioned earlier there are populations of phrases and measures. Each element of the string in the measure population represents a midi event. Each of these measures is allocated an index. The elements of the phrase string are made up of the indexes of the measures that constitute the phrase. *GenJam’s* fitness functions are written to not choose a single fittest phrase but rather a collection of phrases that are of suitable fitness so that the system can produce variety when producing

improvisational music in response to the bass, piano and drum sequences that are fed to it. Future work on *GenJam* is planned where the human mentor could be replaced by a neural net, thereby reducing the bottleneck of having a human listen to the solos generated by it. Figure 2-26 and Figure 2-27 show the architecture of *GenJam* and the gene sequences for the phrase and measure populations respectively [45]..

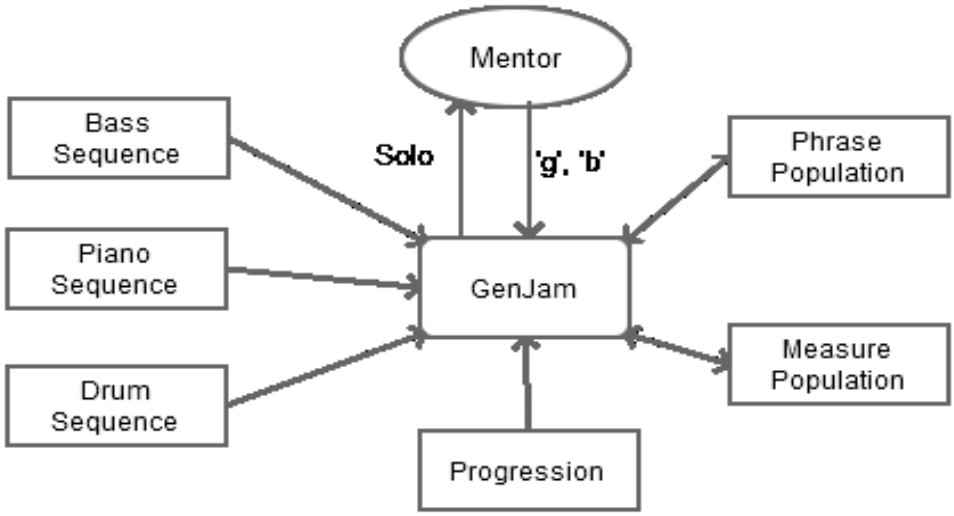

Figure 2-26 GenJam architecture [45]

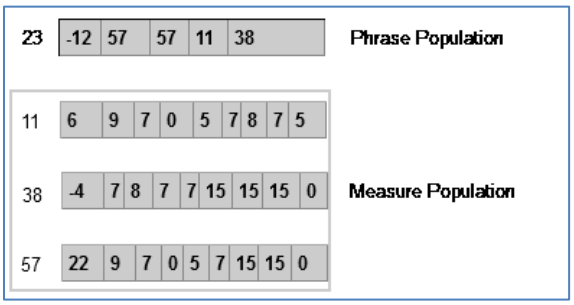

Figure 2-27 GenJam phrase and measure population example

The chromosomes in the Figure 2-27 GenJam phrase and measure population example show how the members of phrase population are indexed to the midi sequences in the measure population. The figure shows the chromosome comprising of index -12, 57,11 and 38, each of which represent a chromosome that is a sequence of midi events and makes up the measure population. The numbers to the left of the chromosomes are the fitness values.

2.3. Summary

Through the course of this chapter we explored three classes of algorithms: *stochastic, planning* and *genetic algorithms* to be applied to music composition. We also covered some background concepts in AI.

We then illustrated some examples of music systems that utilize various AI approaches, concentrating on the strengths of each of these approaches to the task of composing, performing and improvising music. In the next chapter we introduce MAGMA, the Multi AlGorithmic Music Arranger, that builds upon the knowledge of the systems described in this chapter. We will illustrate in detail the various components and algorithms implemented in MAGMA and types of output it generates.

Chapter 3. Multi AlGorithmic Music Arranger - MAGMA

This chapter describes the implementation details of the Multi AlGorithmic Music Arranger (MAGMA). The overall architecture is explained first, along with the common components. Following this, the implementation details of each of the algorithms is outlined.

3.1. MAGMA Architecture

As mentioned in the previous chapters, MAGMA is a system designed to generate music in MIDI format based on user preferences and the incorporation of three different approaches used in the field of Artificial Intelligence (AI): Stochastic Reasoning, Routine Planning and Genetic Algorithms. Based on this requirement to choose an algorithm defined by user preferences, MAGMA is designed and implemented in multiple modules.

Figure 3-1 shows the overall architecture of MAGMA. The *UserPreferences* module serves as the interface with the user in order to gather the preferences chosen by the user, and which are then used in generating the song. The central tier is the *SongBuilder* tier which interfaces with the UserPreferences and the *AlgorithmRunner* tier. SongBuilder takes the preferences obtained by the UserPreferences module and passes them as parameters on to the specific algorithm which is also specified by the user as part of the input. SongBuilder then invokes the execution of the user-specified algorithm. Each algorithm in the AlgorithmRunner tier implements a common interface which allows the SongBuilder module to invoke the construction of a specific part of the song. The output from the algorithm is stored within the data structure of SongBuilder and is combined to build the complete song. SongBuilder is designed to output the song in MIDI as well as ASCII text. This allows us to do some measurements on the output of the system that cannot easily done with the MIDI files.

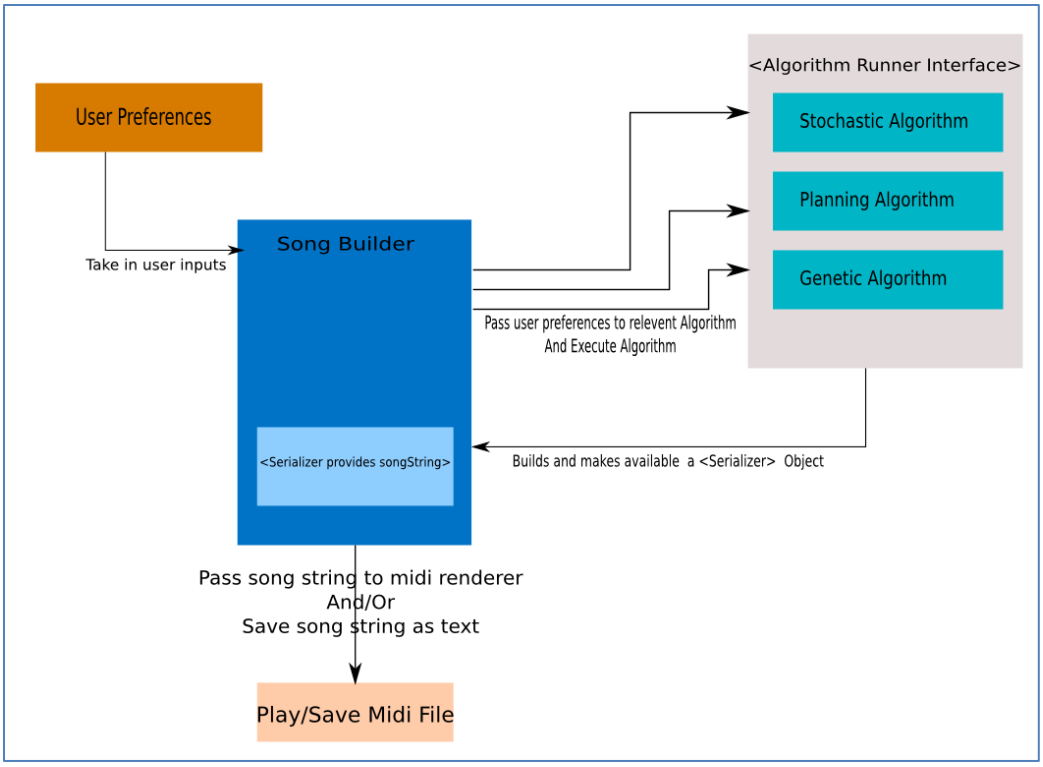

Figure 3-1 MAGMA's architecture

MAGMA is designed to produce songs in the genre of popular (pop) music. Even though pop music means different things to different people, it is widely regarded as the style of music accessible to the widest audience. Furthermore there is a specific structure to pop songs. They are usually between 2 ½ to 5 ½ minutes in length and have specific and repeated sections such as verse and chorus [47].

Following this convention MAGMA internally represents the structure of a song in the hierarchy of song section, measure, chord and melody. This allows the song to be built at the different levels of overall song structure, measure level and finally chord/melody level. Figure 3-2 illustrates this song structure. The overall song structure determines how many sections such as ‘intro’, ‘verse’, ‘chorus’ and how many times they will be repeated within the song. Each of these song sections then contains a sequence of measures which, based on the preferences, can be unique or repeated. Finally, each measure is comprised of a chord and melody sequence.

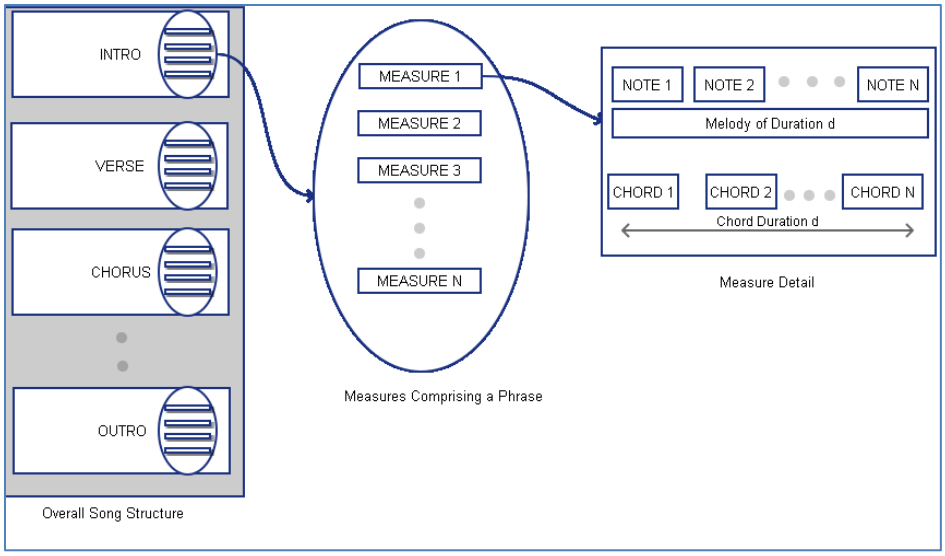

Figure 3-2 Structure of a song in MAGMA

Each algorithm has a mechanism to generate each distinct phrase that makes up the overall song structure, the distinct measures within each phrase and then the chord and melody that comprises each distinct measure. We next describe in detail these common mechanisms that build the overall song from each of these parts starting with the description of the user interface.

3.1.1. User Interface

The user interface in MAGMA is designed to take preferences from a user at a high level and incorporate them into the type of song that is specified. These preferences are designed to make sense to a user who is not necessarily a musician and should make intuitive sense to the casual user. Based on music composition tutorials and music theory concepts, five different preferences are implemented which are designed to capture certain attributes common to all music. These preferences are: *Repetition, Variety, Mood, Transition and Range.*

Repetition describes how often ideas/sections and themes are repeated in a song. Variety, on the other hand, refers to the sense of novelty and uniqueness in different parts of a song [61]. The mood of a song is equated differently to various scales; however there is agreement that the major scales evoke bright, sunny moods whereas the minor ones relate to sadness and melancholy [62]. The size of the intervals or steps from one note to the next or one chord to the next impacts the quality of a song. If the notes are too close together the music can be boring , too far apart and it can end up being erratic and noisy[62,63]. The Transition preference in MAGMA is designed to capture this attribute of music. Finally, range is an

important attribute of music that captures the number of octaves that can be spanned by an instrument or even a human performer [62].

Not all the preferences impact all the layers within the song. Each preference is specified by a value from 1 to 5. The following are the details on each of these preferences.

- **Mood**: This impacts the key and the tempo of the song and is applied at the overall song level. A level of 1 or 2 indicates a happy/upbeat mood. This will result in a faster tempo and a major key for the song. A level of 4 or 5 indicates a somber mood and will result in a slower tempo and minor key. A level of 3 will indicate a mid tempo and random selection of minor or major key.

- **Range**: This impacts the song at the overall song level by specifying the instrument and octave range selections. The higher the range the more varied the octaves when building the chord and melody sections, and the greater the variety of instruments selected.

- **Transition**: This is applied at the chord and melody level. The higher the transition the greater the leaps from note to note and chord to chord whereas a lower transition value leads to more stepwise changes between chords and notes. The former will tend to make the song sound more dissonant, while the latter will make it more predictable and potentially boring.

- **Repetition**: At the overall song level, this will influence how often a phrase such as Verse or Chorus is repeated. Within a phrase, this impacts how many measures are repeated. At the chord and melody level, it dictates how many chords and notes are repeated before transitioning to a different chord and note respectively.

- **Variety**: At the overall song level, this will influence how many different phrases such as Intro, Verse, Chorus, Bridge, Middle Eight and Outro are created. A song with low variety could have no Bridge or Middle Eight represented whereas a song with high variety will likely have them. Within a phrase, a high variety will tend to produce different measures. Similarly, for the chord and melody section, a higher variety will mean more notes and chords used and lower variety will

cause fewer notes and chords to be used.

- **Algorithm:** The user's selection of the Stochastic, Planning or Genetic Algorithm to be used in generating the song.

In addition to the above preferences there is another judging component: *Rhythmic*. This is assigned based on the user preference of Repetition and Variety. It impacts the chord and melody sections and influences the size of the durations of the notes and chords. If the user selected high variety and high repetition then the chords and notes will likely have shorter durations. If the variety and repetition are both low then the chords and notes will likely have longer durations

.In the current implementation of MAGMA the user inputs are defined in a simple text file that the system reads in at execution time. The layout of the file is simple key value pairs such as "T=3" and "M=4" to describe a transition value of 3 and a Mood of 4 respectively. Future implementations will allow this to be input from a graphical user interface (GUI).

3.1.2. The Song Builder

As shown in Figure 3-1 the SongBuilder layer is responsible for taking the user input and passing relevant user preference information to the selected algorithm and building the song from the output of the algorithm. SongBuilder maintains an internal representation of the song as depicted in Figure 3-2. For each song, regardless of the algorithm used, the song is constructed in a top down manner. Following are the steps:

1. Pass the User Preferences to the relevant algorithm. The user preferences specify which algorithm to use. The value for Rhythmic is computed here.
2. SongBuilder invokes the algorithm to generate the song structure. The algorithm generates the overall song structure and returns it to SongBuilder which stores it.
3. For each distinct phrase (intro, verse, chorus etc.) SongBuilder invokes the algorithm to generate the measure structure. This measure structure is then stored in the data structure along with the phrase it belongs to.
4. For each distinct measure the algorithm is invoked to generate the chord sequence and is stored in the data structure along with the measure it belongs to.

5. Finally for each distinct measure the algorithm is invoked to generate the melody sequence and this is stored in the data structure along with the measure it belongs to.

These steps are depicted graphically in Figure 3-3. At the completion of these steps, the entire contents of the song are present within the data structures of SongBuilder. The next step is to generate the output. Rather than utilizing MIDI libraries available in java, we used a third party open source music API library called *JFugue* [46]. The advantage of using JFugue is that it can be given strings that represent music and it in turn generates the MIDI file from these strings. This saves us from having to understand the complexities of generating a MIDI file. Furthermore, the strings that JFugue uses can be parsed to gather statistics on for the experiments performed for this thesis.

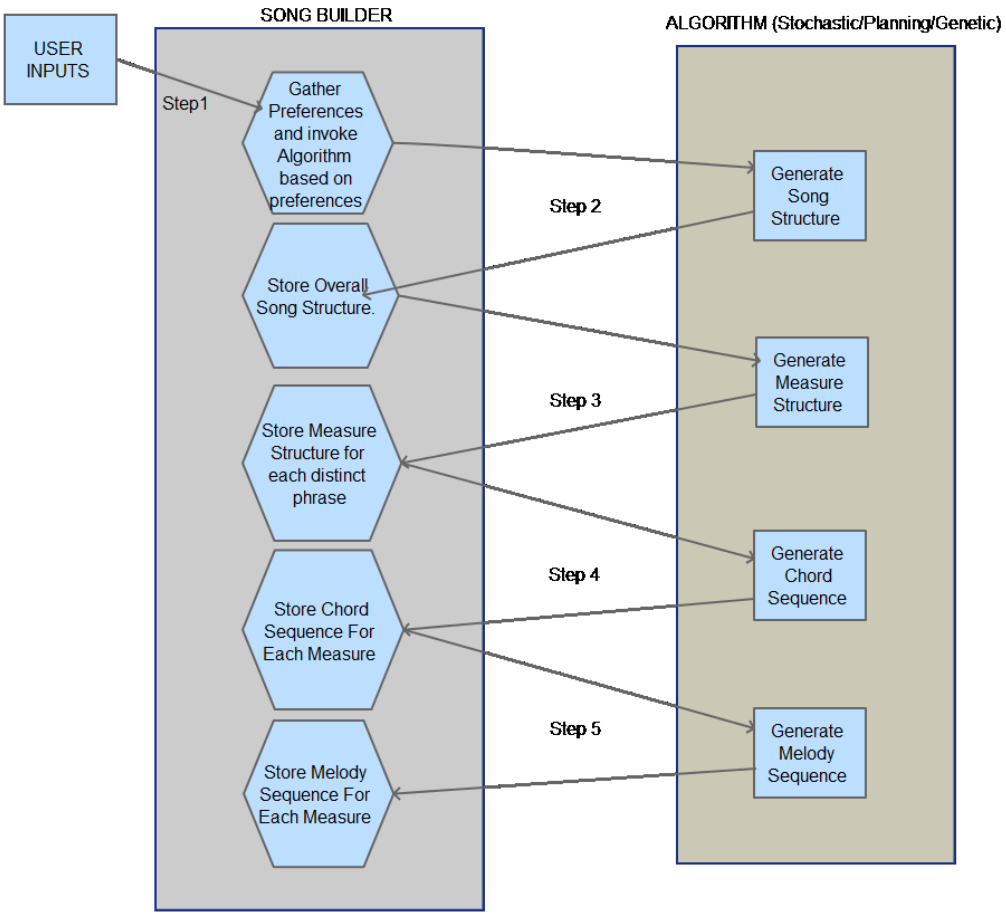

Figure 3-3 SongBuilder flow diagram

3.1.3. Common Interface

Any algorithm that is part of MAGMA is required to implement a public interface. This interface forces the algorithm implementation to provide specific methods that can be used by the SongBuilder module to pass instructions to the specific algorithm depending on which part of the song SongBuilder is building. This interface also provides a pathway for any future algorithms to be included into the system. At present there are 4 methods that the interface enforces and are listed here.

i. *Generate Song Structure*: Instructs the algorithm to generate and return the overall song structure. This will be called once by SongBuilder for each song that is created and will return a value that represents the overall structure of a song, for example Intro-Verse-Chorus-Verse-Chorus-Outro

ii. *Generate Measure Structure*: Instructs the algorithm to generate and return a measure structure. This will be called for each distinct phrase within the song structure. Will return a value such as “1-2-1-2-3”. For the example in (i) this would be called 4 times total. Once each for the Intro, Verse, Chorus and Outro

iii. *Generate Chord and Melody*: Instructs the algorithm to generate and return the chord and melody sequence for each distinct measure. The values generated here conform to the JFugue text strings as described in Section 3.1.4. This method is invoked for each distinct measure in each distinct phrase.

iv. *Set User Preferences*: This method allows the SongBuilder to pass the data structure holding the user preferences to the specific algorithm, which in turns uses those values to generate the values described in (i), (ii) and (iii).

3.1.4. Output

The output of MAGMA is in essence a series of text strings that JFugue can parse to generate MIDI files. These text strings describe all features of the song as determined by the algorithms based on the user preferences. JFugue as an API does not have a notion of a song

or song structure. It simply takes input of notes such as ‘C’, ‘A#’,’Fb’ etc. and chords such as ‘Cmaj’, ‘F#min’, ‘Dbsus7’ etc, and based on the specified instrument converts them into the appropriate MIDI sounds. As a result the strings generated by MAGMA for JFugue are at the chord and melody sequence level. These strings also have attributes for octave and duration. For example, string “C4w Db5q Db5q” instructs *JFugue* to create a MIDI file that plays a whole C note of octave 4 followed by two quarter ‘D flat’ notes one octave higher. Valid octave values are from 1 to 10. Table 3-1 shows all the possible durations produced by MAGMA and consumed by JFugue.

Table 3-1 Available durations in MAGMA/JFugue

Duration	Value
w	whole
h	half
q	quarter
i	eighth
s	sixteenth
t	thirty second
x	sixty fourth

In addition to notes and chords other parameters can be specified for JFugue such as tempo, instruments and which layer the notes or chords should be assigned to. Each parameter follows the convention of having a single letter designated for the parameter, followed by its value in square brackets. ‘T’ specifies tempo, ‘I’ specifies the instrument and ‘V’ specifies the voice or layer. Valid tempo values range from 40 to 220 beats per minute. These parameters are specified before the note or chord sequence and are applied to all the notes and chords in the sequence following the parameters until other such parameters are encountered at which point those new parameters are applied to the following chords and notes. Since the MIDI specification supports 16 simultaneous channels, JFugue voice can have a value from 0 to 15 to correspond to the 16 channels [46]. Figure 3-4 shows the breakdown of chord, note and parameter representations. In this figure the tempo is 80 beats per minute and the instrument is a reed organ and these are applied to the voice channel 0. The various parts of a chord and note are also shown.

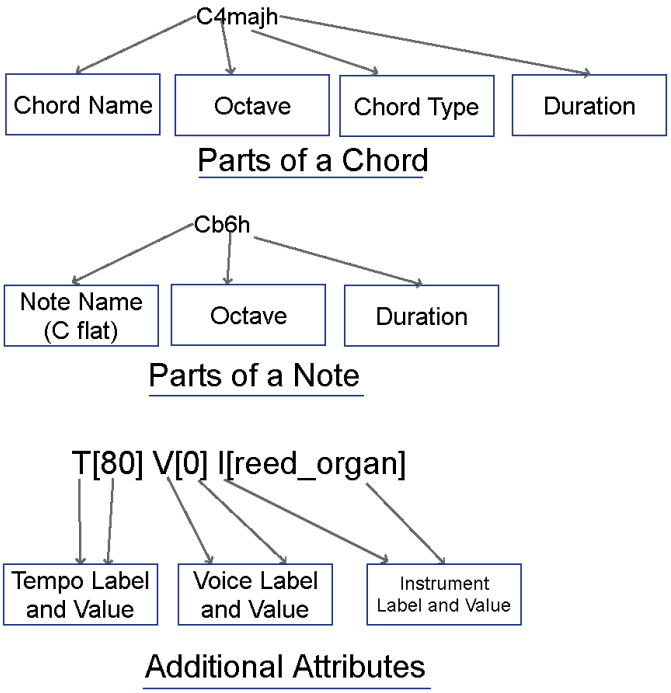

Figure 3-4 Example of text output

MAGMA only uses two voice values of 0 and 1 which result in MIDI files having only two channels or layers. 0 is used for the chords and 1 for notes of the melody. Each measure that is generated is supplied with the parameter of tempo, voice and instrument. The first part of the measure which is the chord layer ends when the 'V[1]' parameter is encountered. This is where the melody section begins right after the instrument parameter is specified. Figure 3-5 shows a partial output of a song generated by MAGMA that represents a song section such as Intro, Verse etc. and in this case comprises of just two measures. The actual output of MAGMA does not have any line breaks but is broken down here into two separate lines for clarity. The first line starts with the Tempo, Voice and Instrument parameters. The Tempo here is 80 beats per minute. The Instrument for the chords (voice 0) is a reed organ, and the instrument for the melody (voice 1) is an electric guitar. The chords add up to 2 whole durations and the melody matches that duration as well. The second line shows the second measure where the instruments are the same as the first measure. The end of the second measure is also the end of the song section.

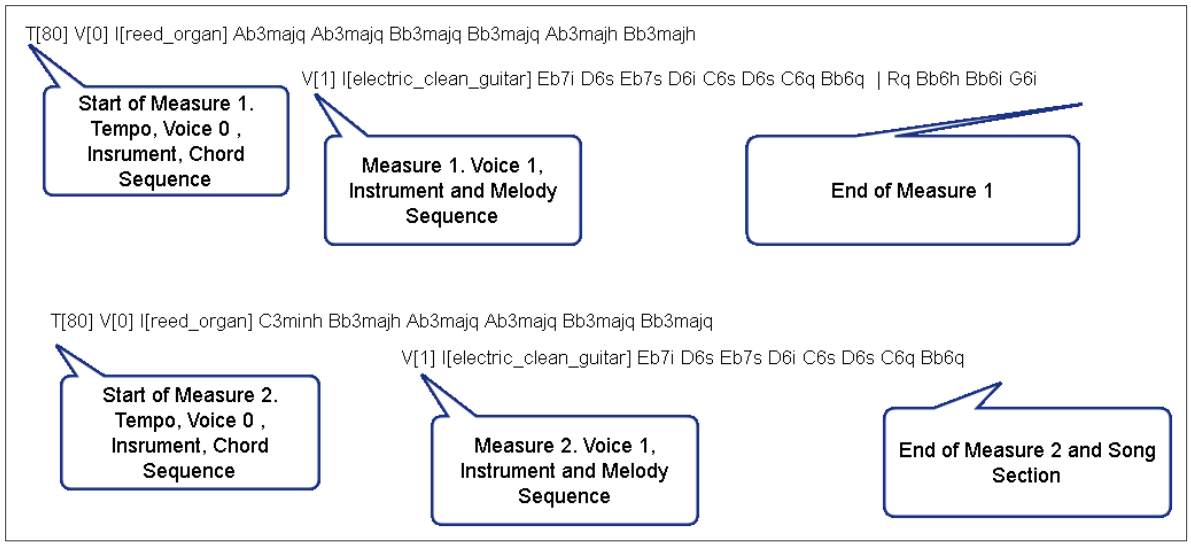

Figure 3-5 Single song section containing two measures

In this section we have seen the overall structure of MAGMA and how the SongBuilder component interacts with the algorithms to build a song. We also showed how the output is generated so that it can be parsed by JFugue to generate a MIDI song. We next describe the implementation of each of the algorithms.

3.2. Algorithm Implementation

In this section we describe the implementation of the Stochastic, Planning and Genetic Algorithm used by MAGMA to generate music. All three algorithms share a common interface with the SongBuilder module in order to conform to the sequence of generating a song described in Section 3.1.3.

3.2.2. Stochastic Approach

Since the stochastic approach relies on existing data to implement a transition matrix, this approach depends on various databases to provide data for the different layers of the song. Each database is implemented as a simple text file. Based on the layer of the song being generated, the appropriate data file is accessed and the relevant data read in to populate a transition matrix.

The data in the files is annotated with preference parameters. For example each chord sequence in the chord sequence data file is annotated by values of Repetition, Variety and Transition. When building the transition matrix for chord sequence generation the lines that most

closely match with the user defined preferences are read to populate the probabilities of the transition matrix

Each step of the process from song structure, through measure structure and down to chord and melody sequence is built in the same way. The database file of the relevant section is read and lines that match the user preference are loaded into the transition matrix. For example in the generation of a chord sequence, if one of the lines from the file that is loaded into the transition matrix is “1|2|3|4|3|2|1” then the transition matrix will have encoded within it the probability of one of the chords following each of the chords encountered here. In this example, a ‘3’ chord is followed by a ‘4’ chord and a ‘3’ chord is followed by a ‘2’ chord. Thus, if a ‘3’ chord is produced then there is a 50% probability that a ‘4’ chord or a ‘2’ chord will be produced afterward. As other lines from the file are loaded, these probability values within the transition matrix change.

Next, based on the preferences, a length is determined via a heuristic for the output of the transition matrix. A random value is chosen initially and the transition matrix is then queried as many times as to satisfy the output length and each value is obtained based on the probabilities within the transition matrix and the sequence built.

When the SongBuilder module calls the ‘Generate Song Structure’ method on the stochastic algorithm, the song structures data file is parsed and the transition matrix built from the lines that match the preferences. Below is a line that would match a high repetition and high variety preference:

{R=5,V=5}=A|B|C|B|C|B|C|B|C|D|D

This and any such lines are loaded in to the transition matrix. The chain is then used to generate a new sequence based on its internal probabilities and sequence length calculated from the preferences. For example a sequence length of 6 would create a new sequence like:

A|B|C|B|C|D

This now is the Phrase sequence of the song, and as can be seen there are four distinct sections (A,B,C,D) that will be generated and populated with measures in the next step. This next step is to generate the measure sequence for each of these sections. The measures database is read and all the lines are loaded into the transition matrix that matches the

preferences. Continuing with the example of high repetition and high variety lines such as the following would be loaded:

{R=4,V=5}=1|2|3|4|1|2|3|4|1|2|3|4|1|2|3|4

OR….

{R=4,V=5}=1|1|1|1|2|2|2|2|3|3|3|3|4|4|4|4

Once again, the probabilities within the transition matrix are the combination of all the lines loaded. The sequence length is calculated via a heuristic, based on the preferences. For a sequence whose length is determined to be 12 a new measure sequence could be built as follows:

1|1|2|3|4|2|2|2|3|4|4|4

This becomes the measure sequence of one of the song section for which it is being built. This step is repeated for each distinct section of the song. Next, for each distinct measure in the sequence a chord and melody is built. Continuing with this example there are 4 (1,2,3,4) distinct measures. In order to build the various distinct measures, 4 files are read. Two for the chord sequence and two for the melody sequence. In both cases of chord and melody, one file describes notes in Nashville notation and the other the durations. Four transition matrices are built and new sequences generated from the transition matrices. Table 3-2 shows examples from the four files. Figure 3-6 shows the building of the song using the database files and the transition matrices.

Table 3-2 Samples from the database

Chord Sequence	{R=2,V=3,T=1}=6|5|4|5|6|5|4|5 {R=2,V=3,T=2}=4|5|2|1|5|4|2|1 {R=2,V=2,T=3}=1|5|4|5|1|5|4|1|5
Chord Duration	{R=3,V=2,H=3}=w|w|w|i|i|i|i|w|w|i|i|w {R=3,V=3,H=3}=w|w|w|w|w|h|h|qqq|q|w {R=5,V=2,H=2}=w|h|h|w|h|h|h|h|w|w|h|h|w|h|h|h|h|w
Melody Sequence	{R=2,V=3,T=4}=1|5|9|5|10|9|5|10 {R=2,V=3,T=4}=1|5|9|5|10|9|5|10 {R=2,V=4,T=4}=-2|3|7|3|8|7|3|7

Melody Duration	{R=5,V=1,H=4}=i|i|i|i|i|i|i|i {R=4,V=2,H=4}=qi|i|q|q {R=5,V=1,H=3}=q|q|q|q

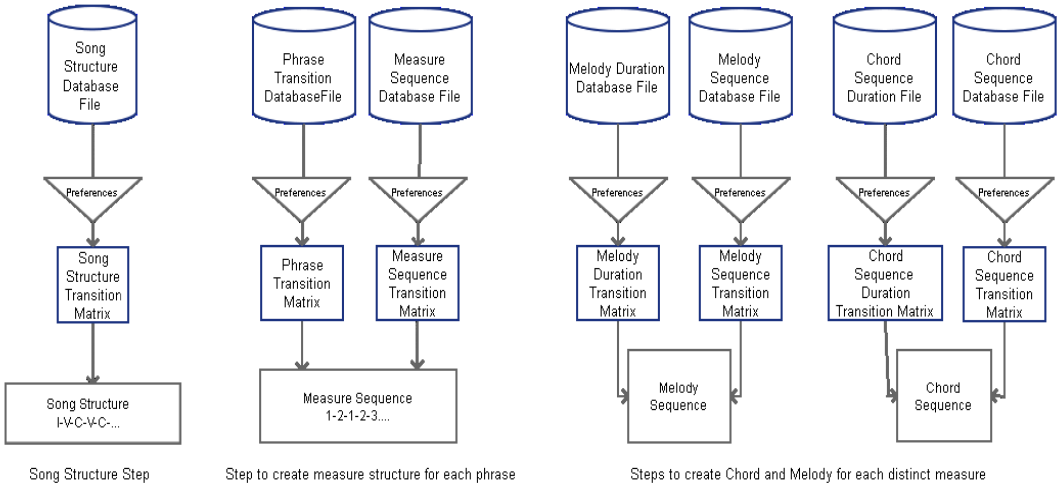

Figure 3-6 Stochastic algorithm process to build a song

The data used to generate the files is collected from various sources. The song structure, measure sequence and chord sequence files were populated by examining song listings on hooktheory.com. This website uses crowd sourcing to provide chord progressions for over 1300 pop songs [48]. By examining a few songs (see Appendix B) we were able to transcribe the data for the song structure and measure sequence files. For the melody sequence and melody duration, we used midi files from musipedia.org [49]. This site has a large database of single note melody MIDI files for a variety of pop songs. These MIDI files were downloaded (see Appendix A) and then converted to *MusicXML* format which is an XML format for describing sheet music [51]. The conversion from MIDI to MusicXML is accomplished using an open source application called *Musescore* [50]. Figure 3-7 shows the process of building these sequences from MIDI files.

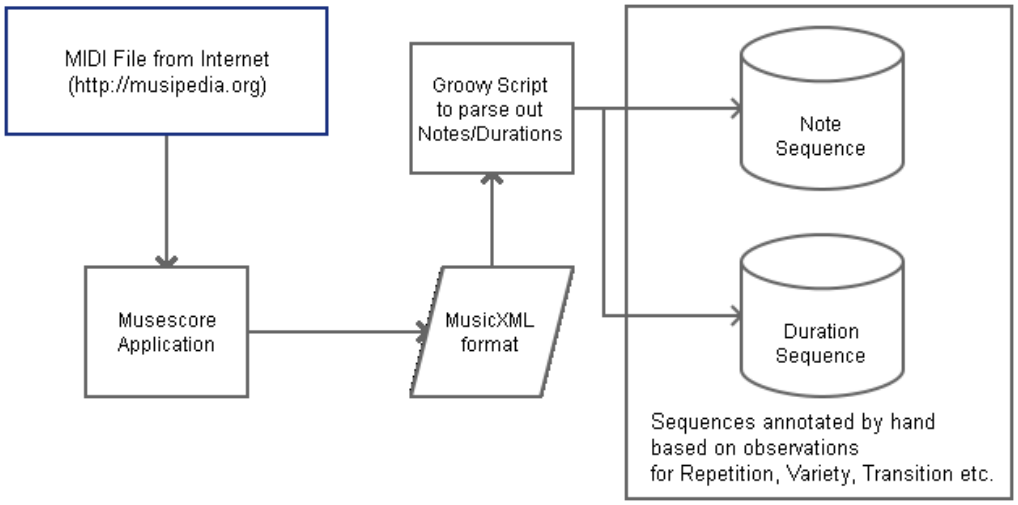

Figure 3-7 Process for extracting data from MIDI file

Once the file was converted to MusicXML, a *Groovy* (programming language) script was written to extract the notes and durations from the file and generate the sequences [60]. Based on the observations of the notes and durations these sequences were then annotated with the preference values of repetition, variety etc. and added to the melody sequence and duration database files. Figure 3-8 shows a three note sample of a MusicXML file and the sequences generated from it.

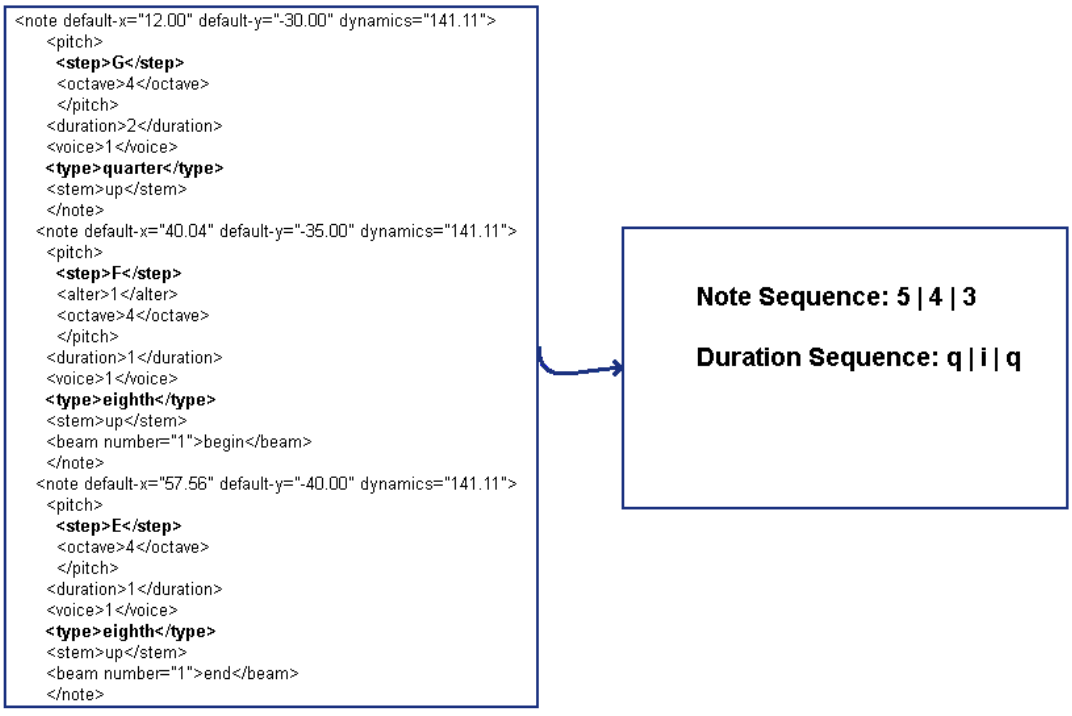

Figure 3-8 MusicXML file to note and duration sequence

3.2.3. Planning Approach

Similar to the stochastic approach, the planning approach relies on database files to provide the data for the various parts of the song. But unlike the stochastic approach, these sequences are used directly instead of being used to populate a transition matrix. The records from the files that represent the relevant level of the song (overall song structure, measure sequence, and chord or melody sequence) are annotated with preference values. Based on the preferences selected by the user a record is chosen from the file. If multiple lines match the user preferences then one is chosen randomly from the matching set.

In Table 3-3, Table 3-4

Table 3-5 and Table 3-6 samples from the planning database files can be seen. In the case of the Planning algorithm there are 4 files. These four files correspond to the four layers of the song and provide data for the song structure, measure sequences and one each for chord and melody sequences.

Table 3-3 Examples of song sequences and their preference

Preference	Sequence
R=1,V=1	I
R=1,V=3	I|V|C
R=1,V=5	I|V|C|B|O
R=2,V=1	V|V|V|V|V|V
R=2,V=1	V|V|V|V|V|V
R=2,V=2	V|V|C|C
R=2,V=3	I|V|C|I|V|C
R=3,V=1	I|I|I|I
R=3,V=3	I|I|I|V|V|V|C|C|C
R=3,V=3	I|V|C|I|V|C|I|V|C
R=3,V=4	I|V|C|V|C|B|C|O
R=5,V=1	V|V|V|V|V|V
R=5,V=2	V|C|V|C|V|C|V|C|V|C
R=5,V=5	I|V|C|B|O|I|V|C|B|O|I|V|C|B|O|I|V|C|B|O

Table 3-4 Measure sequences

Preference	Sequence
R=1,V=1	1
R=1,V=2	1|2
R=1,V=3	1|2|3
R=1,V=5	1|2|3|4|5
R=1,V=5	1|2|3|4|5
R=2,V=3	1|2|3|1|1|2|3
R=2,V=3	1|2|3|1|2|3
R=2,V=3	1|2|3|4|1|2|3|4
R=3,V=2	1|2|3|3|1|2|3|3
R=3,V=3	1|1|1|2|2|2|3|3|3|3
R=3,V=4	1|2|3|4|1|2|3|4|1|2|3|4|
R=4,V=4	1|2|1|2|1|2|3|4
R=5,V=1	1|1|1|1|1|1|1|1|1|1|1|1|1|1|1
R=5,V=5	1|1|1|1|1|2|2|2|2|2|3|3|3|3|3|4|4|4|4|4|5|5|5|5|5

Table 3-5 Chord sequences

Preference	Sequence
R=1,T=1,V=1,H=3	5w
R=1,T=1,V=1,H=3	6w
R=1,T=1,V=2,H=3	4h|5h
R=1,T=1,V=2,H=3	4q|4q|5q|5q
R=1,T=1,V=2,H=3	5h|6h
R=1,T=1,V=2,H=3	6h|5h
R=1,T=5,V=4,H=5	5q|2q|6q|1q
R=2,T=4,V=2,H=3	1q|1q|1q|5q
R=2,T=5,V=4,H=5	5i|5i|2i|2i|6i||6i|1i|1i
R=3,T=3,V=2,H=3	5q|5q|5q|2q
R=3,T=4,V=2,H=3	1q|7q|7q|7q
R=3,T=4,V=2,H=3	1q|7q|7q|7q
R=4,T=4,V=4,H=4	1q|1q|1q|5q|5q|5q|6q|6q|6q|6q|3q|3q|3q|3q|1q|2q|3q

R=5,T=5,V=5,H=5	1i|1i|1i|5i|5i|5i|6i|6i|6i|6i|3i|3i|3i|3i|1i|2i|3i|8i|8i|8i|8i

Table 3-6 Melody sequences

Preference	Sequence
R=1,V=2,T=3,H=3	8i|7s|8s|7i|6s|7s|6q|5q
R=1,V=2,T=4,H=3	0q|5q|1i|3i|1i|5i|5q|1q
R=1,V=3,T=1,H=4	6i|5i|4i|3i|4h|6i|5i|4i|3i
R=1,V=3,T=2,H=2	5q|2q|1q|-1i|-1i
R=2,V=1,T=3,H=2	6i|7q|8q|7q|0i|6q|7i|8q
R=2,V=2,T=2,H=4	3i|3i|5i|5i|5i
R=2,V=2,T=2,H=4	6q|6i|5i|5q|3i
R=2,V=2,T=3,H=3	0i|1i|6i|5i|6q|6i|5i|6i
R=2,V=2,T=4,H=3	1q|1i|1i|3q|5q|9s|8s
R=4,V=3,T=1,H=3	5h|5q|5i|5i|5q|4i|3i|3q
R=4,V=5,T=5,H=3	3q|3q|3q|1i|1s|5s|3q|1i'
R=5,V=3,T=1,H=3	0q|5i|5i|5i|5i|4i|4i|4i
R=5,V=3,T=1,H=3	8i|8i|8i|8i|8i|8i|7q|5i
R=5,T=5,V=5,H=5	1i|1i|1i|5i|5i|5i|6i|6i|6i|6i|3i|3i|3i|3i|1i|2i|3i|8i|8i|8i|8i

The procedure to generate the song follows the same pattern of generating the song structure first. Based on the user preferences the line most closely matching the preferences is selected. If more than one matches exactly then one is chosen randomly from the candidates.

An example line from the song structure database looks like:

{R=1,V=3}=I|V|C|V|C|O

If this is the record selected based on preferences then the song structure is created with 4 distinct phrases: I, V, C and O. For each of these distinct phrases in the song structure, a measure sequence is built by selecting measures sequences from the measure sequence database file. Preference values dictate the candidate lines to be selected and on multiple matches one is randomly chosen.

An example of a measure sequence from the database is:

{R=3,V=4}=1|2|3|4|1|2|3|4|1|2|3|4|

Each number in the sequence represents a measure and the distinct list of numbers is the amount of measures for which a chord and melody is built. In the example above this would be 4 unique measures (1, 2, 3 and 4). The chord and melody sequences for each measure are selected based on preferences as well.

Following is an example of chord sequence:

{R=1,T=1,V=2,H=3}=4q|4q|5q|5q

As can be seen from the example there are four properties that annotate the sequence. Repetition for how often a chord is repeated consecutively, transition for how big the jump is from one chord to the next, variety for how many different chords are used and finally the rhythmic value on how short or long the durations are for the chords. Also of note is that the chord sequence is depicted in Nashville notation which is key agnostic. The preference values of mood affects what key the chords will be converted to. In the above example if the key is C major then the sequence will be converted to two F quarter notes and two G quarter notes.

The melody sequences in the database are similar to the chord sequence records. The only difference being that the numbers in Nashville notation represent single notes instead of chords. And as is the case with the chords the key is applied to the notes based on the mood preferences. The key that is applied to the chords is also applied to the melody within the measure to conform to music theory conventions. When applying the chord and melody sequence to a measure, there is a probability that the durations between the chord and melody will not match. If the melody sequence is longer than the chord sequence then it is truncated to match the total length of the chord sequence. If the melody sequence is shorter than the last note's duration is increased to match the length of the chord sequence. The tables Table 3-3,Table 3-4Table 3-5 and Table 3-6 show song, measure, chord and melody sequences respectively along with the preferences that are associated with those sequences.

Figure 3-9 Planning algorithm flow shows the flow of the planning approach as the song structure, measure sequence, chord sequence and melody sequences are built. It is similar to the stochastic approach with the difference being that instead of creating transition matrices of probabilities, the sequences are used from the database directly.

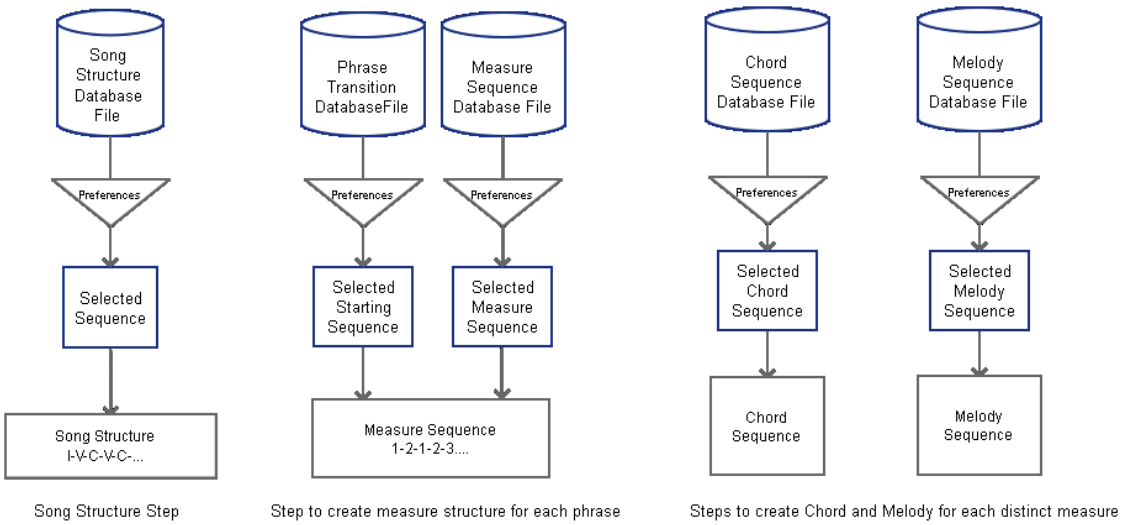

Figure 3-9 Planning algorithm flow

All the data for the database files is taken from song analysis done on hooktheory.com [48]. By examining the songs on this site (see Appendix C) it was possible to get a variety of songs for structure measure sequences and chord and melody sequences. In addition some sequences were added to the database that we thought might sound good or simply to add variety.

3.2.4. Genetic Algorithm Approach

In contrast to the Stochastic and Planning approach, the Genetic Algorithm approach does not rely on any database of files. Instead four different chromosomes are used to represent the four parts of the song: the overall song structure, a measure sequence, a chord sequence and a melody sequence. Each of these chromosomes has a different encoding to facilitate the generation of the structure it is responsible for. Table 3-7 lists the examples of the encodings.

Table 3-7 Encodings of various elements

Song Part	Example Encoding (comma separated)
Overall Song Structure	A, B, C, B, C, D
Measure Sequence	1, 1, 2, 3, 2, 3, 4, 3
Chord Sequence	1, 4, 3, 4, 7, 4, 3
Melody Sequence	1, -1, 3, 4, 7, 15, 11

A fitness function exists for the user preferences of Variety, Repetition, Transition and Rhythmic. Since not all preferences apply to all parts of the song, not all fitness functions are applied to all of the chromosomes that are evolved for their respective song part.

The overall song structure is evolved first. The fitness functions of Repetition and Variety are applied to evolve the song structure. Encoded within this chromosome is the number of distinct phrases of the song. The preferences of Variety and Repetition dictate, via a heuristic, how long the song structure sequence is going to be. A choice of low variety and low repetition will have a shorter song structure than a preference for high variety and high repetition. For example the string “A, B, C, B, C, D” represents a song structure, where there are 4 distinct sections to the song and may be viewed as I-V-C-V-C-O structure.

The next step is to evolve the measure sequence for each distinct song section. As in the case of the overall song structure the preference of repetition and variety impact the size of the chromosome. Higher repetition and variety will result in a longer chromosome and a lower value for these two preferences will result in a shorter chromosome. The chromosome is represented as a number value from 1 to 9:

1,2,3,2,4,3,1

In this example there are 7 total measures while 4 are distinct. For each of the distinct measures, MAGMA will next evolve a chord and melody sequence. The chord and melody sequence are represented as numbers between -7 and 14 which in turn are the notes in Nashville notation. Negative numbers indicated lower octaves and numbers greater than 7 indicated higher octaves. An example of a chromosome representing a melody sequence is:

1, 2,1,-1,-2,-3,4,8,12

Chord and melody sequences are influenced by preferences of repetition, variety, transition. The example sequence above has low repetition, high variety (5 different notes) and high transition since the notes from -3 to 4 , 4 to 8 and 8 to 12 have significant jumps between them.

The genetic algorithm for all song parts is implemented using an initial population size of 10. As mentioned earlier the actual size of the chromosome varies based on the user preferences. However all the chromosomes of a population are built to the same size and initialized randomly. The algorithm steps are as follows:

```
Initialize Population
Current population = Initial Population
    While ( Generation limit not reached)
        Select 2 Parents based on fitness
        Select 2 that are most diverse to the selected parents
        Cross over the 4 selected parents to produce 6 children
        Mutate the 4 parents to produce 4 more children
        Assign all children to current population
        Evaluate all children using the fitness functions
     Done
```

The measure and song structure populations are evolved over 20 generations since they are simpler and there is a limited number of combinations possible. In other words, the search space of possible combinations for song structure and measure sequences is limited. The chord and melody populations are evolved over 100 generations each since there are a lot more possibilities and much larger search space. A higher number of generations could potentially be beneficial, however, the fitness functions evaluate the chord and melody sequence for repetition, variety, transition and music theory which are computationally intensive and the time frame to generate a song becomes impractical. In all cases of the song parts, the fitness functions are weighted based on the user preferences provided. So if the user selects high repetition then the repetition fitness function will have a larger weight.

The crossover point is chosen at random from one of the parents since all parents are of equal length. Mutation is done by swapping out one value with another randomly chosen value. So for example an ‘A’ can swapped out with a ‘B’ in the overall song structure chromosome, or a ‘2’ with a ‘4’ in the measure chromosome. In the chord and note chromosome the mutation operator replaces the current note or chord with random chord or note from -7 to 14 and replaces the duration with a random duration from the list of durations specified in Table 3-1. We next describe the details of the fitness functions used in the implementation of the genetic algorithm.

3.2.4.1. Fitness Functions

For each of the fitness functions, the value returned is applied to a weight based on the user preferences. Thus, the fitness function’s impact can increase or decrease its influence on any chromosome. For example, if the user prefers high repetition then the repetition fitness

function will return a higher score if the evaluated chromosome has high repetition. However, if the user prefers low repetition then the same chromosome would get a lower score when evaluated for repetition. Following are the functions used:

1. *Diversity Fitness Function*: Gives a value for the variety within a chromosome. The more varied the values the higher the value returned by the fitness function. For example in the case of a melody represented by the string 1,2,3,2,3,4 the value returned would be 4. This fitness function is used in all layers of the song.[1]

2. *Repetition Fitness Function*: For any sequence represented by the chromosome counts the consecutively repeated elements of the chromosome and returns the value for the highest repeated sequence. For example if the sequence is 1,2,2,2,3,4,5,5,7 then the fitness value returned by the function will be 3. This fitness function is used in all layers of the song.

3. *Transition Fitness Function*: Measures the difference between the values of a sequence. Returns an average of all the jumps between elements of the sequence. For example if the sequence is 1,2,3,10,7, then the value returned will be the average of the step sizes, which is (1+1+7+3)/4 = 3. This fitness function is used in the chord and melody sequence generation of the song.

4. *Rhythmic Fitness Function:* Used in the chord and melody chromosome, evaluates the size of the durations in the chord and melody and based on the preference returns appropriate value. The fitness function employs a heuristic to achieve the score. If the Rhythmic preference is low then durations of whole and half notes will have a higher value. If the Rhythmic preference is high then the shorter durations such as 32nd and 64th notes will produce higher values. Since the preferences are taken into account

[1] Diversity here is not diversity between children that will become the parents of the next generation, but rather the diversity of the song being produced.

in the value returned by this fitness function no additional weight is applied to it.

5. *Music Theory Fitness Function:* Used only in the chord and melody chromosomes. The preferences do not impact the output of this fitness function. The function measures how well the sequence fits certain music theory recommendations such as:
    i. Start on a 1 note.
    ii. End on a 4 or 5 note.
    iii. Use a 6 note in middle or after a 1 note or before a 4 or 5 note.
    iv. Use a 2 note before a 4 or 5.
    v. Do not end on a 2 note.

All the values provided by the above fitness functions, unless noted otherwise, are then factored in with the preferences selected and weighted accordingly. Figure 3-10 shows the implementation of the genetic algorithm in MAGMA.

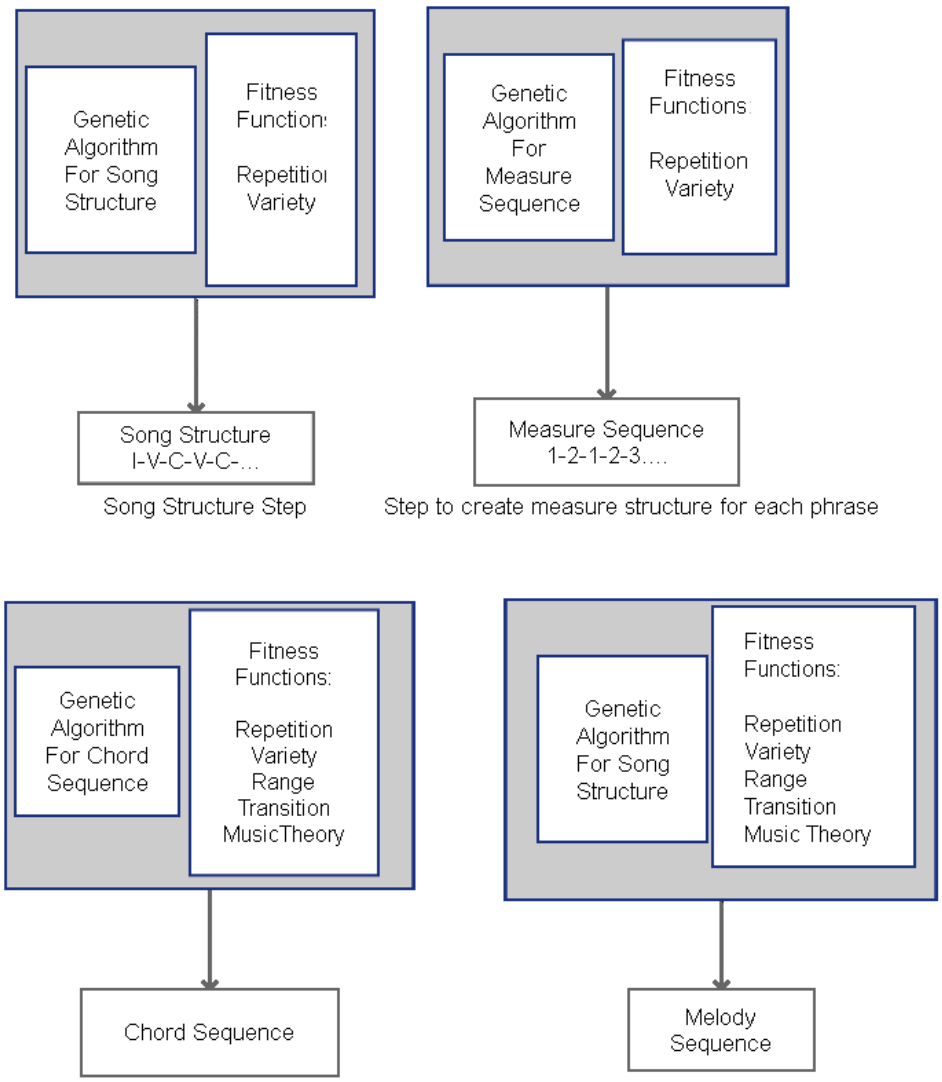

Figure 3-10 Genetic Algorithm in MAGMA

3.3. Summary

This chapter provided the details on the implementation of MAGMA. The overall architecture of the system was described along with the internal data representation of a song. Common modules such as the SongBuilder and the algorithm interface were explained. The overall process flow within MAGMA from the input of user preferences to song output was

illustrated including the format of the text output. For the stochastic algorithm details were provided on how existing MIDI files downloaded from a website were parsed to get note and duration data. Finally, the details on all the three algorithms were described including samples from relevant data files.

The next chapter details the testing that was done with MAGMA. This includes statistics on the output generated as compared to user preferences and overall quality of the songs from a subjective point of view.

Chapter 4. MAGMA Evaluation

In this chapter we evaluate the music composed by MAGMA. The evaluation is conducted in two parts. In the first part over a hundred songs were generated and analyzed to ascertain how well they fit the user preferences. The second part is a subjective evaluation where different parameters are tried to see if certain combinations produce more 'listenable' music than others.

4.1. Evaluation Setup

As described in chapter 3 MAGMA produces output in both MIDI as well as text. The text output consists of markers that describe the start of sections and measures as well as contain chord and note data. Figure 4-1 shows a sample of the output.

```
|Phrase| T[260] V[0] I[piano] E1majq E1majq D#1mins D#1mins V[1] I[piano] F#4s G#4q F#4s G#4i | C#4i |
T[260] V[0] I[piano] E1majh E1majh D#1minh D#1minh V[1] I[piano] G#4i F#4q E4i B4h | F#4q D#4q B4h |
T[260] V[0] I[piano] B1majs B1majs A#1dimq A#1dimq V[1] I[piano] F#4i | G#4i B4s E4i F#4s A#4i | T[260]
V[0] I[piano] E1majh E1majh D#1minh D#1minh V[1] I[piano] G#4i F#4q E4i B4h | F#4q D#4q B4h | T[260]
V[0] I[piano] D#1mins D#1mins G#1mini G#1mini V[1] I[piano] C#4i | D#4q | |Phrase| T[260] V[0] I[piano]
A#1dimq A#1dimq E1majh E1majh V[1] I[piano] E4q F#4s F#4s G#4i | F#4s G#4i G#4i C#4i A#4i F#4qttttttt |
T[260] V[0] I[piano] C#1mini C#1mini G#1mini G#1mini V[1] I[piano] C#4q | B4i C#4i | T[260] V[0] I[piano]
A#1dimq A#1dimq E1majh E1majh V[1] I[piano] E4q F#4s F#4s G#4i | F#4s G#4i G#4i C#4i A#4i F#4qttttttt |
T[260] V[0] I[piano] C#1mini C#1mini G#1mini G#1mini V[1] I[piano] C#4q | B4i C#4i | T[260] V[0] I[piano]
A#1dimq A#1dimq E1majh E1majh V[1] I[piano] E4q F#4s F#4s G#4i | F#4s G#4i G#4i C#4i A#4i F#4qttttttt |
```

Figure 4-1 Snippet of text output generated by MAGMA

The text in Figure 4-1 shows the contents of two consecutive phrases. A phrase is a section of a song such as the 'intro', 'verse' or 'chorus' etc. Within the phrase are attributes that give the tempo of the phrase identified here by 'T[260]' (260 beats per min in this case). The 'V[0]' term assigns a layer or track to the song. All songs in MAGMA are limited to two layers. Layers play music simultaneously. In all songs generated by MAGMA, layer 0 is the chords layer and layer 1 is the melody layer. Each chord in the chord layer has a number to denote its octave and a letter at the end that gives its duration. For example the chord 'E1majq', is an E major chord played in octave 1 for a quarter note duration. Similarly each note in the melody layer has a number to denote its octave and a letter to denote its duration. For example the note 'C#4i' is a C sharp note played in octave 4 for an eighth duration. By parsing this text output it is

possible to obtain information about the song such as the number of chords and notes used, the counts of particular types of chords (major or minor), the types of durations used and so forth. Other statistics such as step size that are more easily computable in Nashville (numeric) notation are generated as the song is being built and output to the console. Further details on the text output have been described in chapter 3.

As mentioned previously, MAGMA songs are influenced by 5 primary preference parameters of Transition, Repetition, Variety, Range and Mood. Each of these preferences accepts a value from 1 to 5. For example a value of 5 for Repetition would generate songs with more consecutive sections (Verse-Verse) as well as repeated measures(1-1-2-2) etc. and repetition in the chords and notes as well. A value of 1 for repetition will generate songs with very few repeated sections, measures etc.

In order to test how well the system handles preferences a variety of combinations are used. Figure 4-2 shows these combinations. The first five combinations simply apply the same value to all five preferences from 1 to 5. This shows how the impact of the preferences scale when compared to each other. The next five combinations take one preference and assign it the maximum value of 5 and keep the other preferences with a value of 1. This is to see the impact of the preference with an extremely high preference value while potentially muting the impact of the others. The final five combinations reverse this approach and give one preference a value of 1 while giving the others the maximum value of 5. This is done to see the impact of the preference at the lowest extreme.

Transition=1 Repetition=1 Variety=1 Range=1 Mood=1
Transition=2 Repetition=2 Variety=2 Range=2 Mood=2
Transition=3 Repetition=3 Variety=3 Range=3 Mood=3
Transition=4 Repetition=4 Variety=4 Range=4 Mood=4
Transition=5 Repetition=5 Variety=5 Range=5 Mood=5
Transition=5 Repetition=1 Variety=1 Range=1 Mood=1
Transition=1 Repetition=5 Variety=1 Range=1 Mood=1
Transition=1 Repetition=1 Variety=5 Range=1 Mood=1
Transition=1 Repetition=1 Variety=1 Range=5 Mood=1
Transition=1 Repetition=1 Variety=1 Range=1 Mood=5
Transition=1 Repetition=5 Variety=5 Range=5 Mood=5
Transition=5 Repetition=1 Variety=5 Range=5 Mood=5
Transition=5 Repetition=5 Variety=1 Range=5 Mood=5
Transition=5 Repetition=5 Variety=5 Range=1 Mood=5
Transition=5 Repetition=5 Variety=5 Range=5 Mood=1

Figure 4-2 Preference variations used to generate data

These 15 combinations create the most distinct songs possible in the system. This gives a total of 15 songs for each of the 3 algorithms or a total of 45 songs. In order to add more songs and account for variations produced within consecutive runs with the same preference, this process is repeated 3 times to generate 3 songs for each combination of algorithm and preference giving a total of 135 songs. Since there is no reasonable way to combine the data of the three songs produced for each identical preference set, one song is chosen randomly to be part of the data set of 45 songs.

4.2. Evaluation Data

For each of the 15 combinations of preferences (for a single algorithm) data is chosen that demonstrates the impact of each of the preferences or their combinations. For example Chord and Note counts are impacted by Repetition and Variety, whereas the octaves used is impacted by Range. Table 4-1 shows all the relevant data that is influenced by the corresponding attribute values.

Table 4-1 Song data and influencing attributes

Data Chart	Impacted by Attributes
Overall Song Structure, Section/Measure Counts	Repetition , Variety
Chord Counts / Note Counts	Repetition , Variety
Duration Counts for Chord, Notes	Repetition, Variety
Major vs. Minor Chord Counts	Mood
Chord/Note Step Size	Transition
Octaves Used	Range

For this analysis the same set of data is examined for each of the algorithms in order to compare the results directly with each other. As a consequence the same data charts are generated for each of the algorithms and are presented in the coming subsections. The methodology employed to generate this data was done in two steps. First a script was run to iterate over the fifteen preference permutations listed in Figure 4-2. This script generates a text file containing calculated values such as total chord counts, octave counts etc. A separate file is

generated for each song. In the second step, a different script is used to aggregate and combine the relevant data into a single comma separated (csv) file for each feature under consideration. This file is then used in a spreadsheet program to generate a chart. For example Figure 4-3 shows the contents of the csv file that provides overall song stats such as number of sections and number of measures:

Preference	Number of Sections	Number of Measures
T=1 R=1 V=1 G=1 M=1	4	20
T=1 R=1 V=1 G=1 M=5	4	20
T=1 R=1 V=1 G=5 M=1	4	20
T=1 R=1 V=5 G=1 M=1	8	72
T=1 R=5 V=1 G=1 M=1	8	72
T=1 R=5 V=5 G=5 M=5	10	160
T=2 R=2 V=2 G=2 M=2	7	56
T=3 R=3 V=3 G=3 M=3	10	130
T=4 R=4 V=4 G=4 M=4	10	160
T=5 R=1 V=1 G=1 M=1	4	20
T=5 R=1 V=5 G=5 M=5	8	72
T=5 R=5 V=1 G=5 M=5	8	72
T=5 R=5 V=5 G=1 M=5	10	160
T=5 R=5 V=5 G=5 M=1	10	160
T=5 R=5 V=5 G=5 M=5	10	160

Figure 4-3 Example of data generated for each preference set

The same process is repeated for each of the algorithms from the subset of the 135 songs. All the charts are presented as bar graphs and clustered bar graphs where appropriate, with the legends provided. The y axis in all cases is the count number. The x-axis in all cases is the values for the preferences which have been abbreviated as T for Transition, R for Repetition, V for Variety, G for Range and M for Mood. We begin with the details on the stochastic algorithm.

4.2.1. Stochastic Algorithm

The data presented here begins with the overall song structure, followed by the details on the chord and note counts. Next the data on chord and note durations is presented. Major and Minor chord distribution follows and finally step size and octaves round off the analysis.

4.2.1.1. Overall Song Structure

This aspect of a song is impacted by Repetition and Variety. The chart in Figure 4-4 illustrates how the number of sections and the total number of measures in the song increases with increase in Repetition and Variety. The data shown reflects the total number of sections and measures which would include all repeated sections and measures. The total of the durations the songs increases with the increase in the number of sections and measures

.

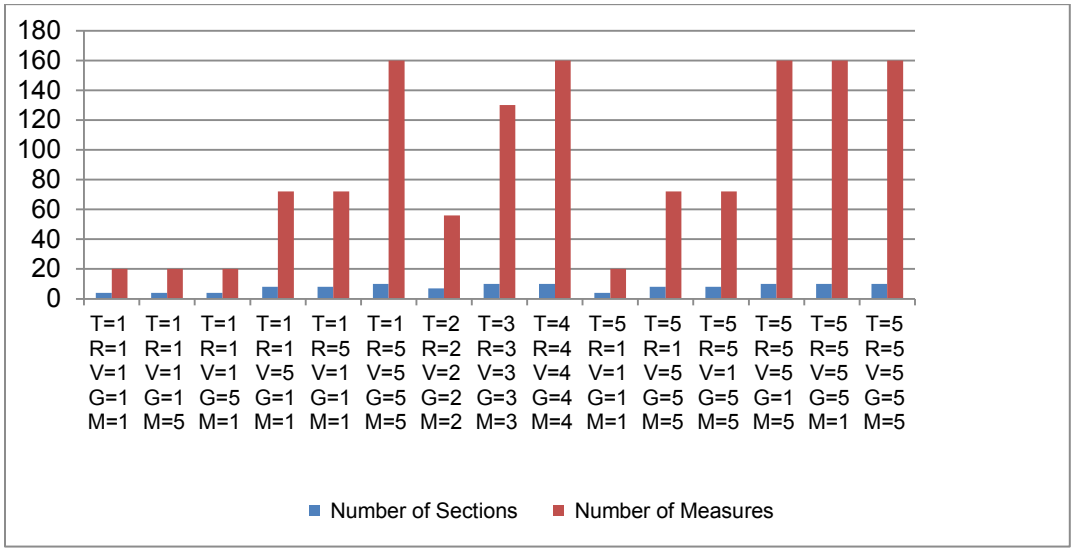

Figure 4-4 Overall structure of stochastic songs

4.2.1.2. Chord and Note Counts

Going one step further into the details we can observe the number of chords and notes for each of the fifteen preferences in Figure 4-5. Once again as the repetition and variety increases the count increases. This count takes into account all the chords and notes regardless of how often they are repeated.

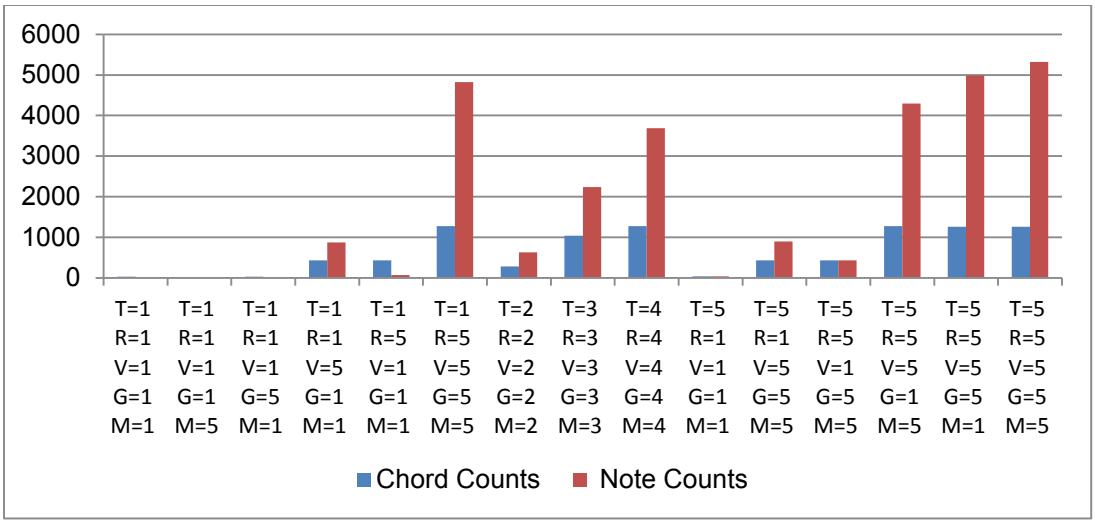

Figure 4-5 Chord and Note Counts

4.2.1.3. Chord and Note Durations

Another factor to consider is the durations of the notes and how they vary with the changes in the preferences. The chart in Figure 4-6 shows the duration distribution between whole (w), half(h), quarter(q), eighth(i),sixteenth(s), thirty-second(t) and sixty-fourth(x) chords.

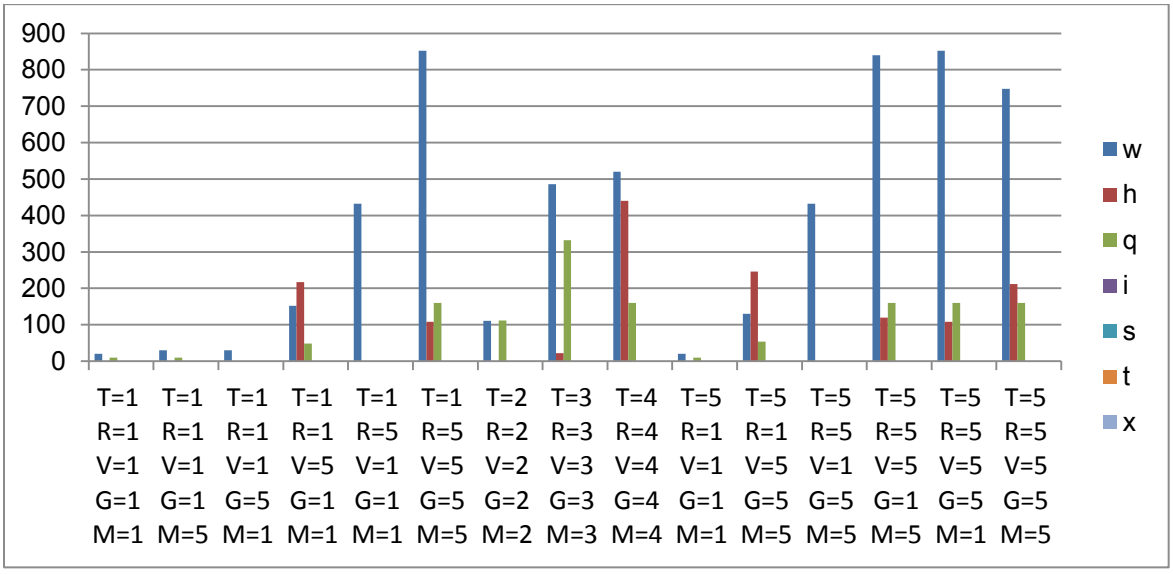

Figure 4-6 Various durations used among the chords

The chord durations in the above figure varies with the preferences. Chords have a heavy bias towards whole and half notes and hence we can see that they remain predominantly

the most frequently used chords. Increase in variety values along with different pitches increase the variety in durations. This is reflected in both Figure 4-6 for chords and **Error! Reference ource not found.** for notes. For high repetition and variety the number of notes increases and the shorter duration notes occur more frequently than the longer duration notes.

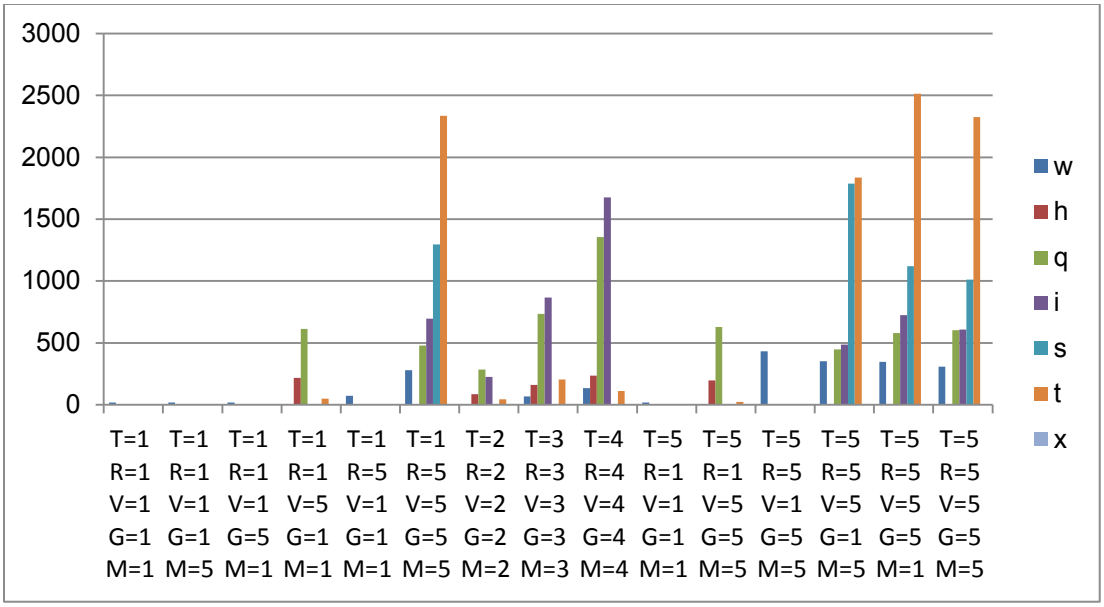

Figure 4-7 Note Durations

4.2.1.4. Major and Minor Chords

The preference of Mood impacts the frequency with which major and minor chords are used. A value closer to 1 is designed to give more major chords whereas a value closer to 5 will give more minor chords. A value in the middle at 3 should give an even distribution among major and minor chords. The chart in Figure 4-8 shows the distribution between major and minor chords. For higher values of Mood (which correspond to somber mood) the minor chord count increases, while the opposite happens for lower (happy) values of Mood. The middle range results in considerably more major chords being used, indicating that the distribution is not as even for this value.

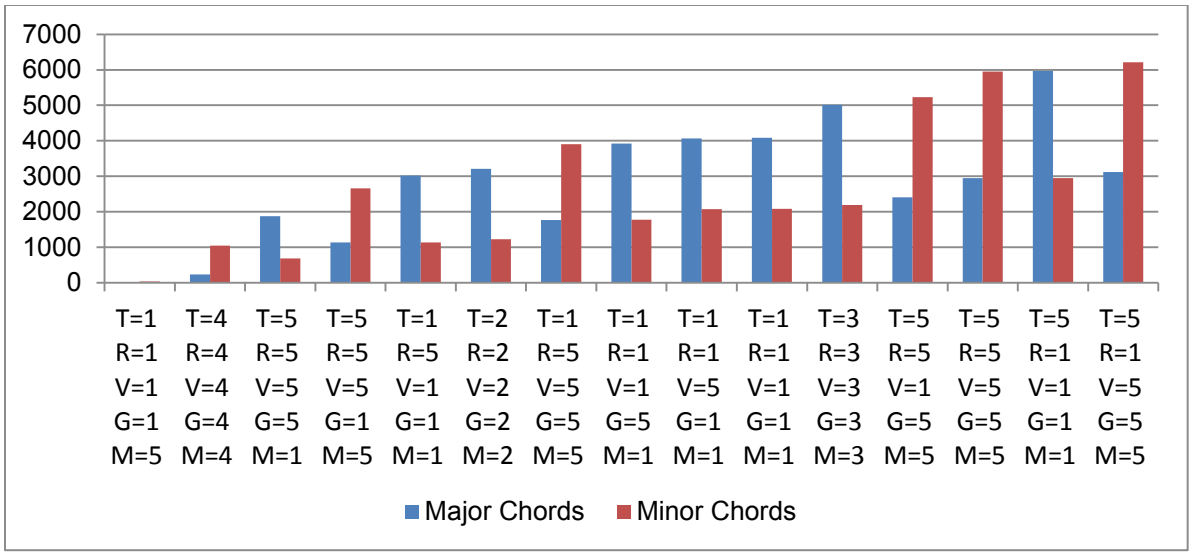

Figure 4-8 Distribution of major and minor chords

4.2.1.5. Step Size

To gauge the impact of the Transition preference we measure the average step size between chords and notes. This is show in Figure 4-9 where the step sizes increase with increase in Transition value.

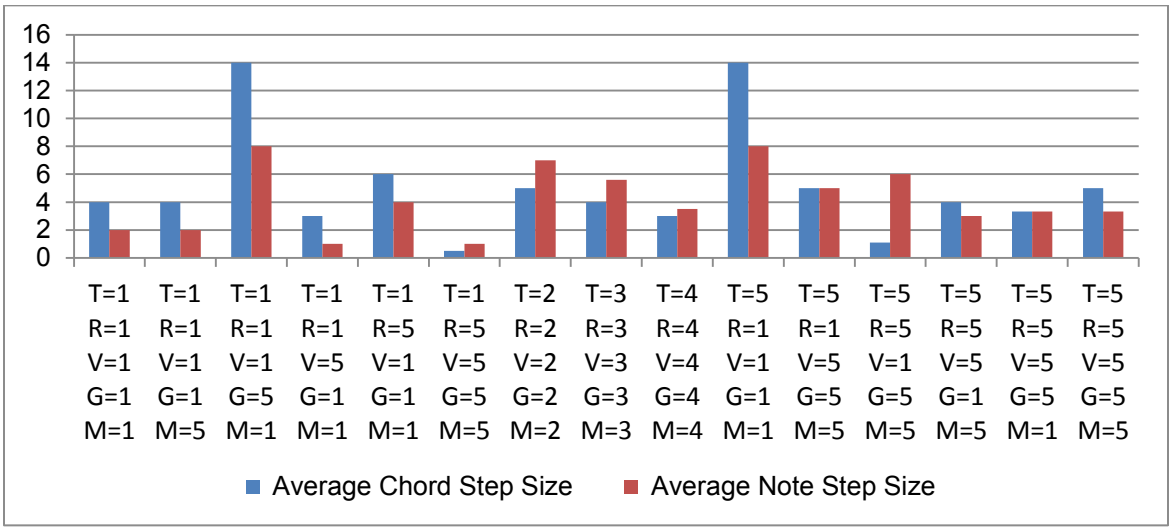

Figure 4-9 Step sizes in chords and notes

The average transition size increases for songs that have low repetition and low variety and hence have fewer overall chords and notes. Interestingly, songs which have a higher value of

Range where the notes and chords are spread out over various octaves we observe an increase in the average transition size since some transitions are ‘jumps’ across an octave.

4.2.1.6. Octaves Used

In Figure 4-10 and Figure 4-11 we can see the number of chords and notes respectively and the octaves used by them. Higher Range values show the use of multiple octaves in the generated songs.

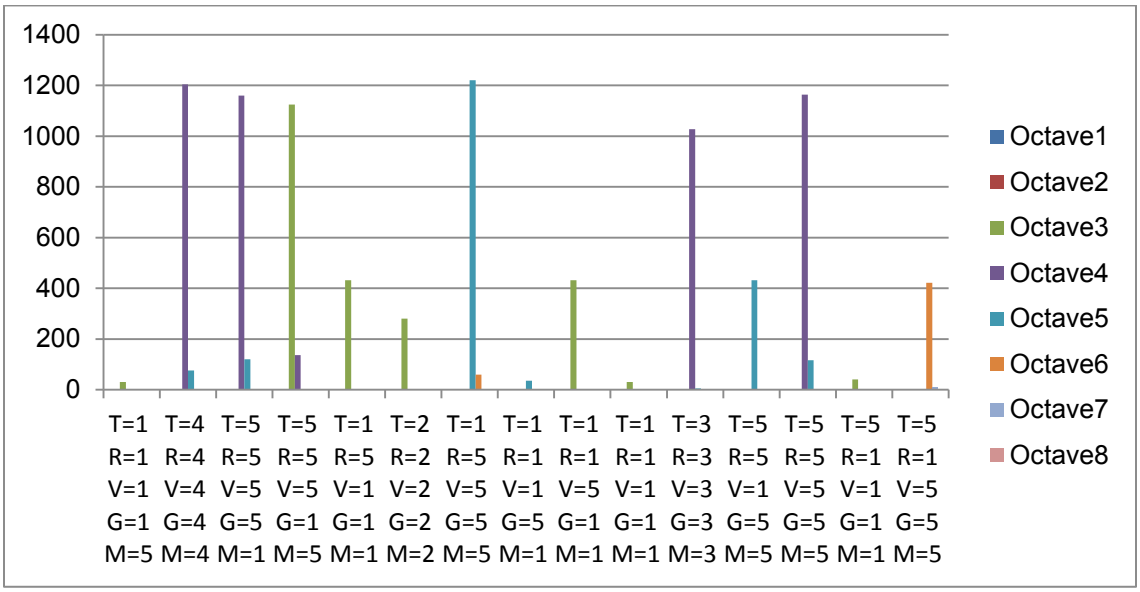

Figure 4-10 Octaves used in chords

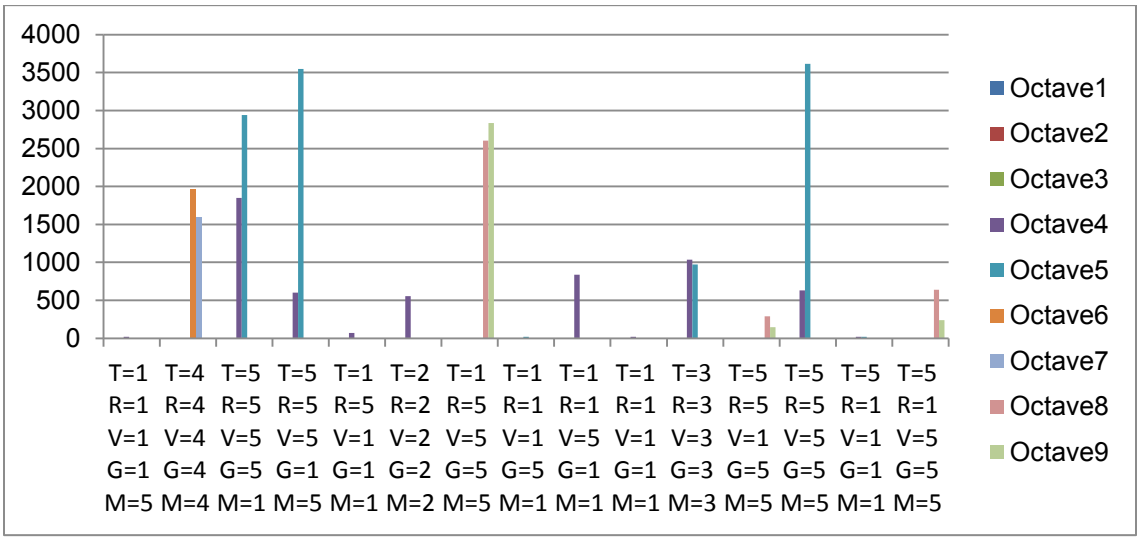

Figure 4-11 Octaves used in the notes

4.2.2. Planning Algorithm

We next display the data for the planning algorithm. The data follows the same sequence as that for the stochastic algorithm discussion.

4.2.2.1. Overall Song Structure

The counts in the planning algorithm are more precise when the same values of Variety and Repetition are used. The distribution of the section and measure counts as seen in Figure 4-12 reflects the preference values.

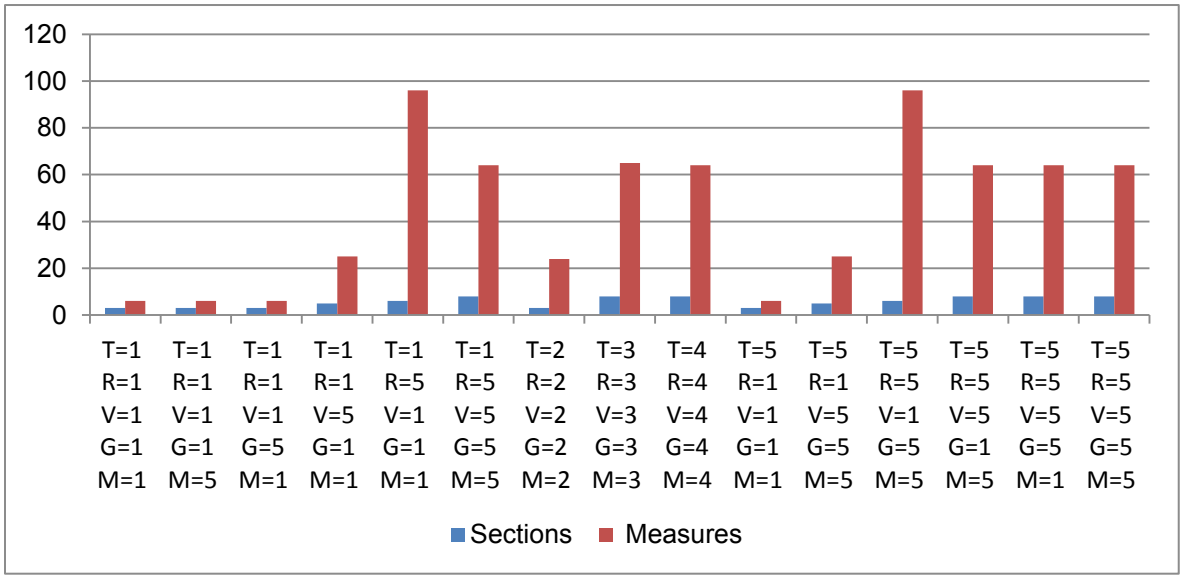

Figure 4-12 Song sections and measures count

4.2.2.2. Chord and Note Counts

The chord and note counts increase and decrease with the increase and decrease of the Repetition and Variety preferences as shown in Figure 4-13. The difference in the counts of matching Variety and Repetition numbers is a result of the variations caused by different sequences in the database with the same preference values.

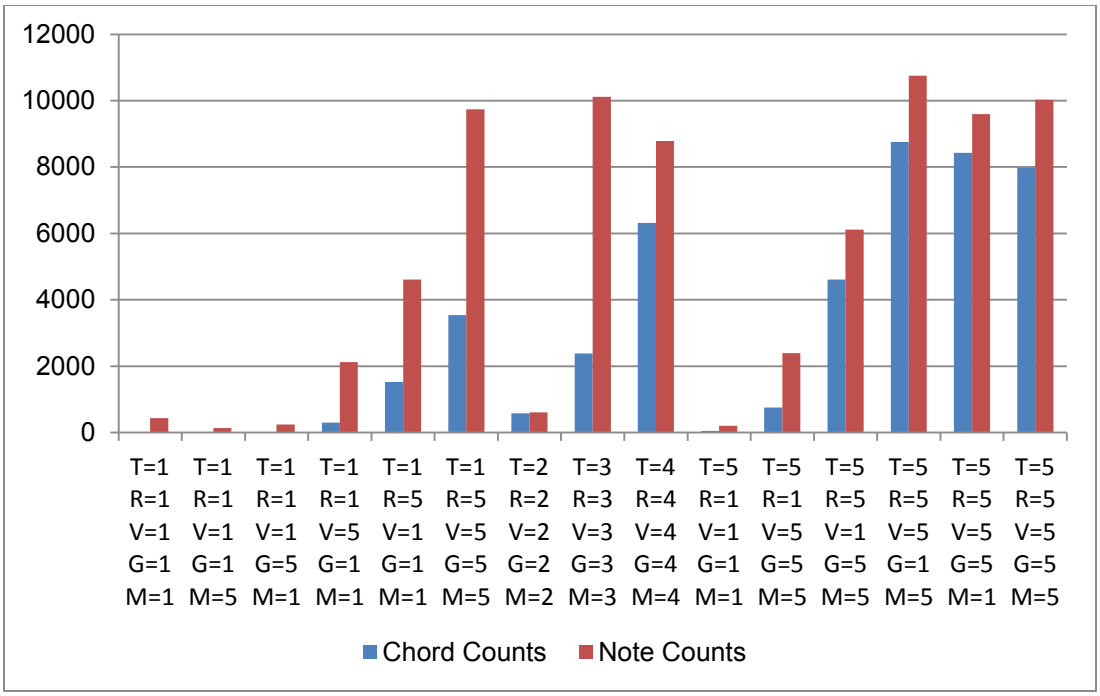

Figure 4-13 Chord and note counts

4.2.2.3. Chord and Note Durations

Chord and Note durations as shown in Figure 4-14 and Figure 4-15 respectively increase with higher Variety settings. Higher Repetition values along with higher Variety values increase the use of shorter durations such as the Eighth chord/Notes. Adding more sequences to the database would increase the different durations used when generating the songs.

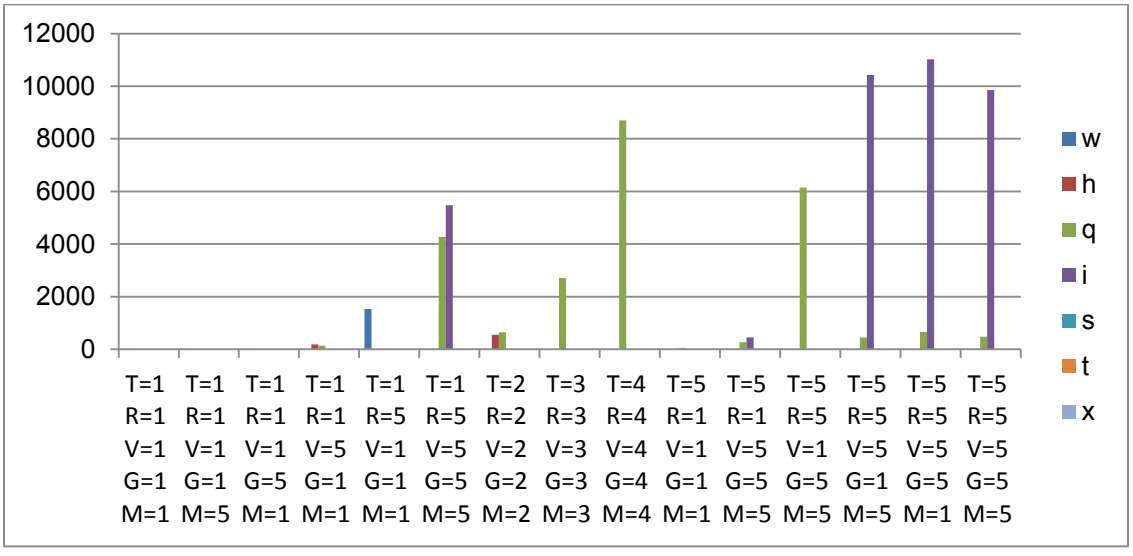

Figure 4-14 Chord and note durations

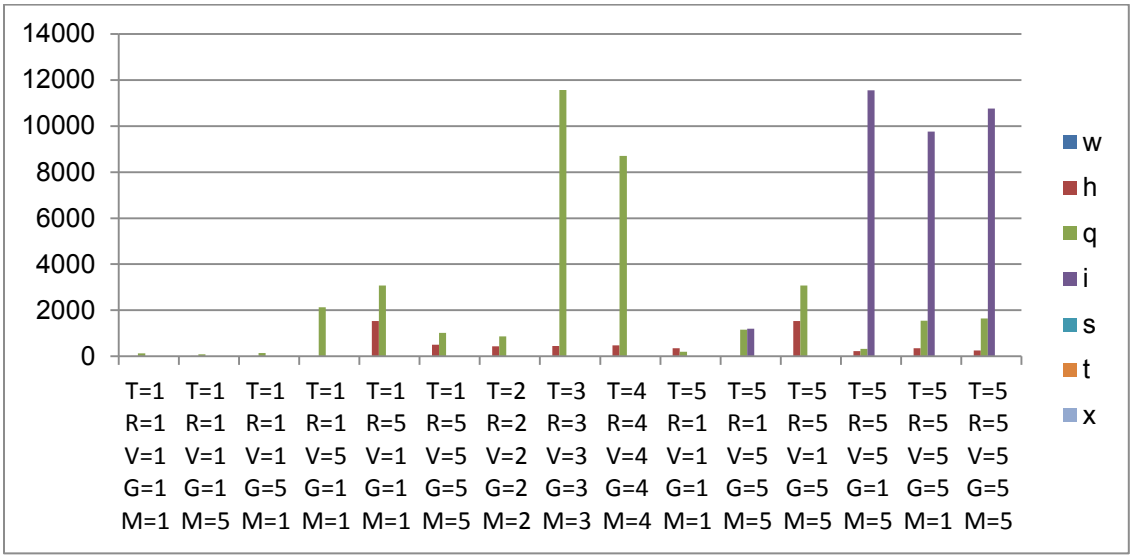

Figure 4-15 Note durations used

4.2.2.4. Major and Minor Chords

The major and minor chord distribution follows the expected behavior of the Mood preference. Higher values result in more minor chords and lower values in more major chords as shown in Figure 4-16.

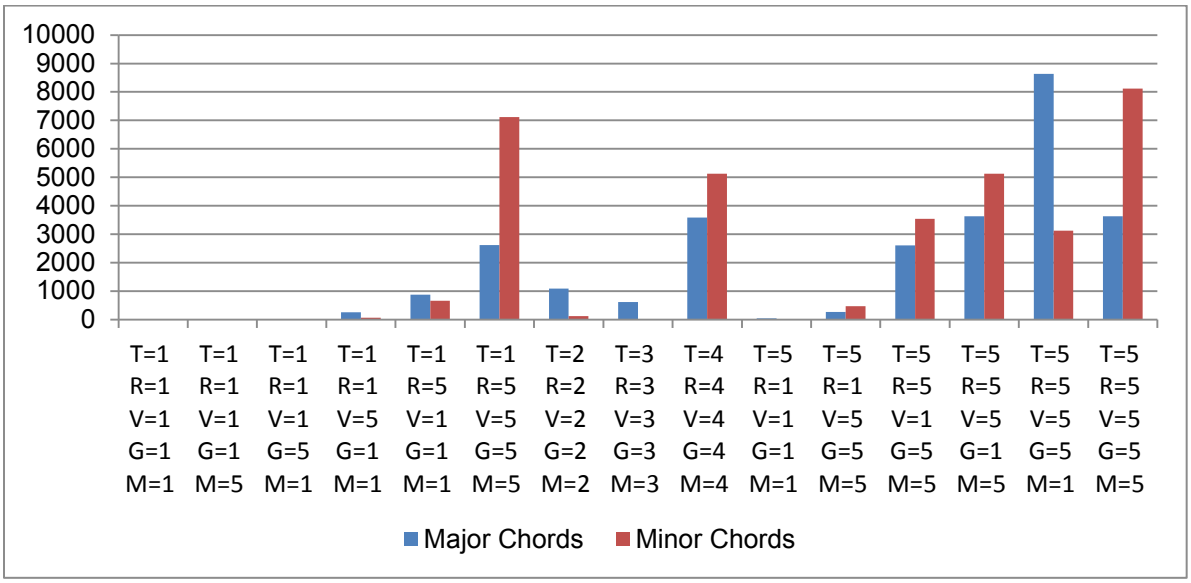

Figure 4-16 Major and minor chord distribution

4.2.2.5. Chord and Note Step Sizes

The step sizes with the planning algorithm follow the expected behavior of the Transition preference increasing the step size as shown in Figure 4-17. Unlike the stochastic algorithm the high repetition and variety values do not diminish the average step size and the overall step sizes are in a narrower range than the stochastic algorithm.

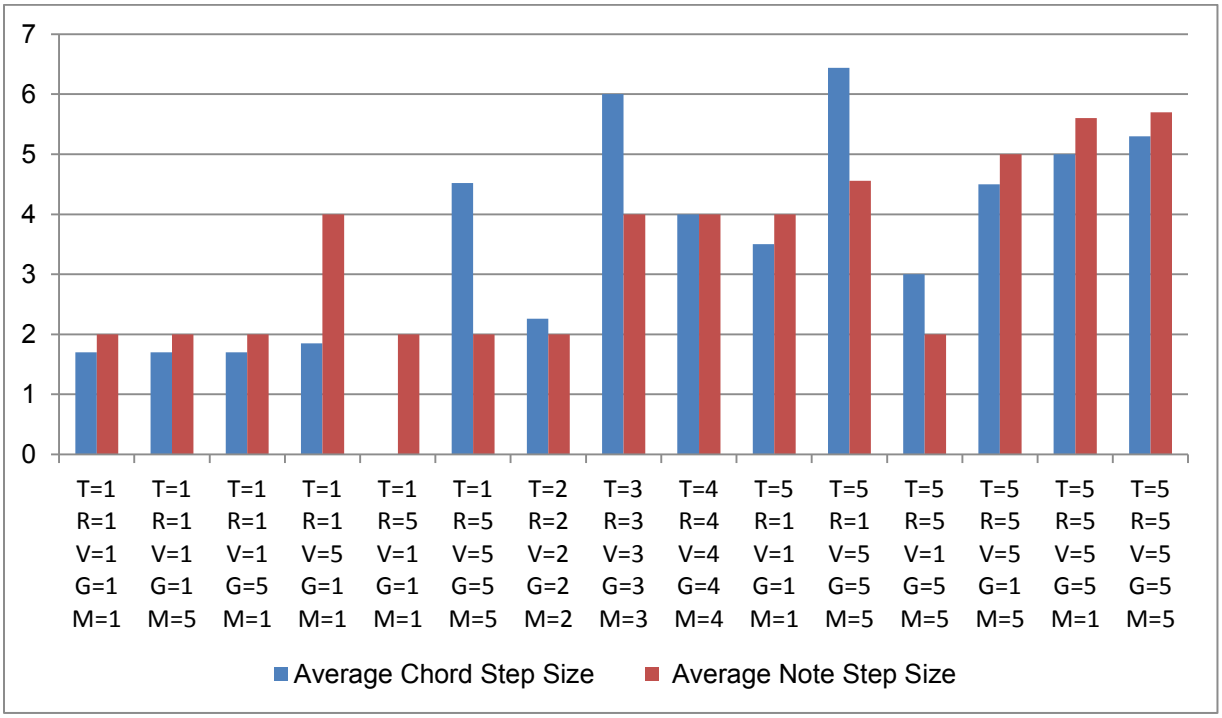

Figure 4-17 Chord and note step sizes

4.2.2.6. Chord and Note Octaves

Figure 4-18 and Figure 4-19 show the chord and note octaves used respectively. The data reflects the expected behavior of higher Range values producing songs spanning more octaves.

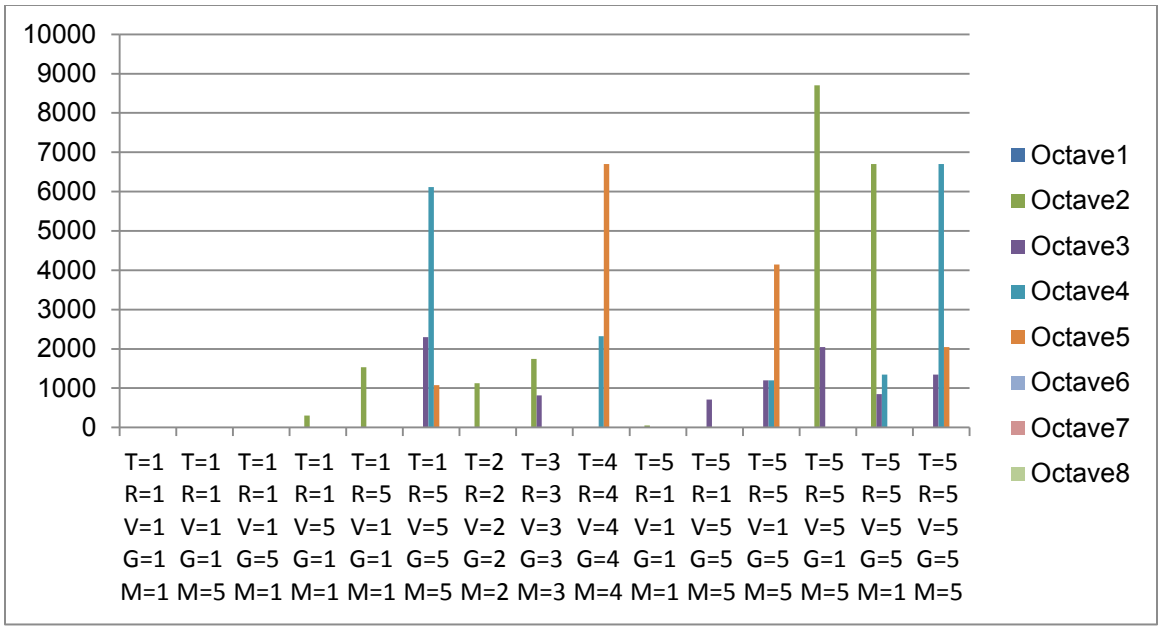

Figure 4-18 Chord Octaves Used

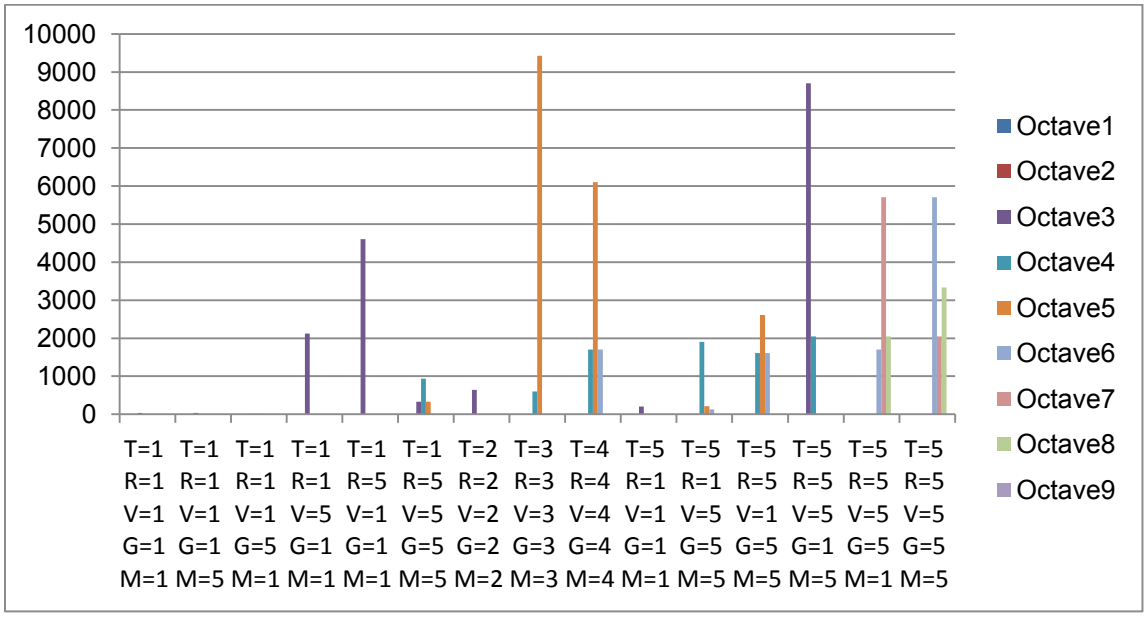

Figure 4-19 Melody octaves used

4.2.3. Genetic Algorithm

This section provides the same metrics for genetic algorithm produced songs as was done for the stochastic and planning songs. As with the other two algorithms any noteworthy data findings are elaborated upon.

4.2.3.1. Overall Song Structure

The song section and measure counts follow the preferences. Even though the actual sequences within the structures are evolved in the application of the genetic algorithm, these sizes are pre-determined by the preference values and so result in identical counts for the same values of repetition and variety as shown in Figure 4-20.

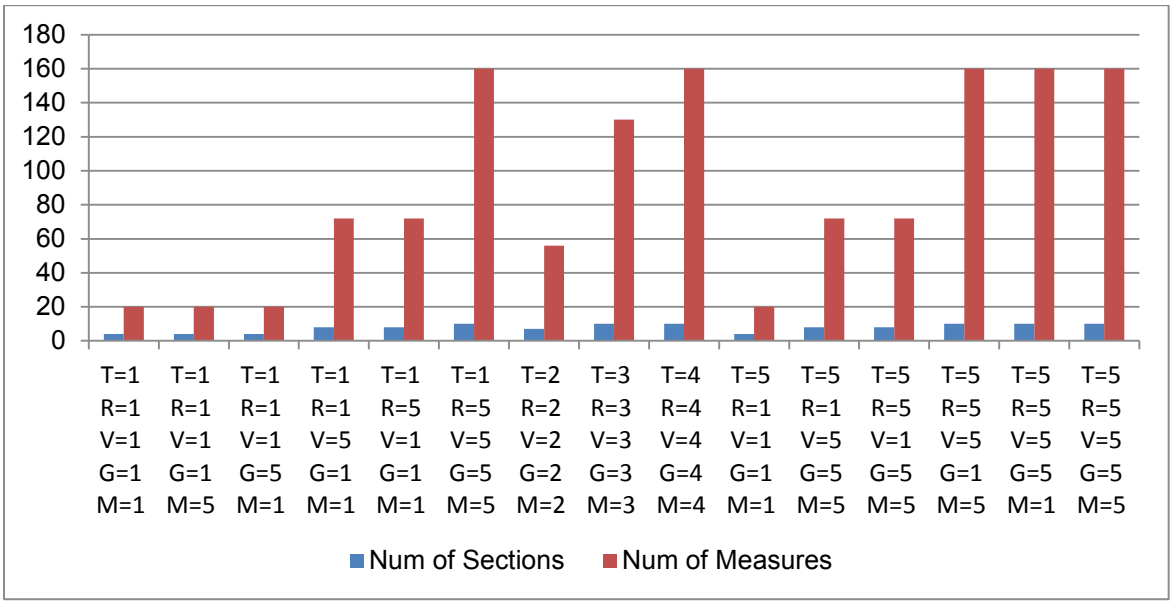

Figure 4-20 Overall Song Structure

4.2.3.2. Chord and Note Counts

The chord and note counts mirror the preference requirements since they are pre-determined by the size of the chromosomes that make up the genetic algorithm. This is show in Figure 4-21

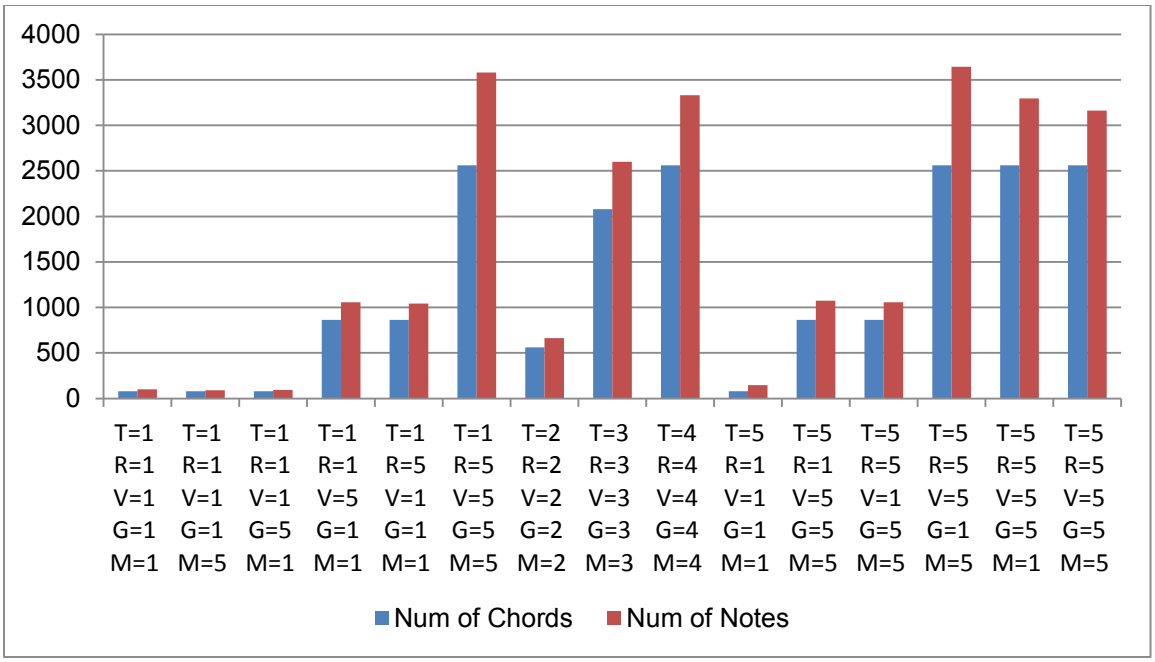

Figure 4-21 Chord and note counts

4.2.3.3. Chord and Note Durations

The chord and note durations when compared to the other two approaches show a more even spread of durations applied to the chords and melody as can be seen in Figure 4-22 and Figure 4-23 respectively.

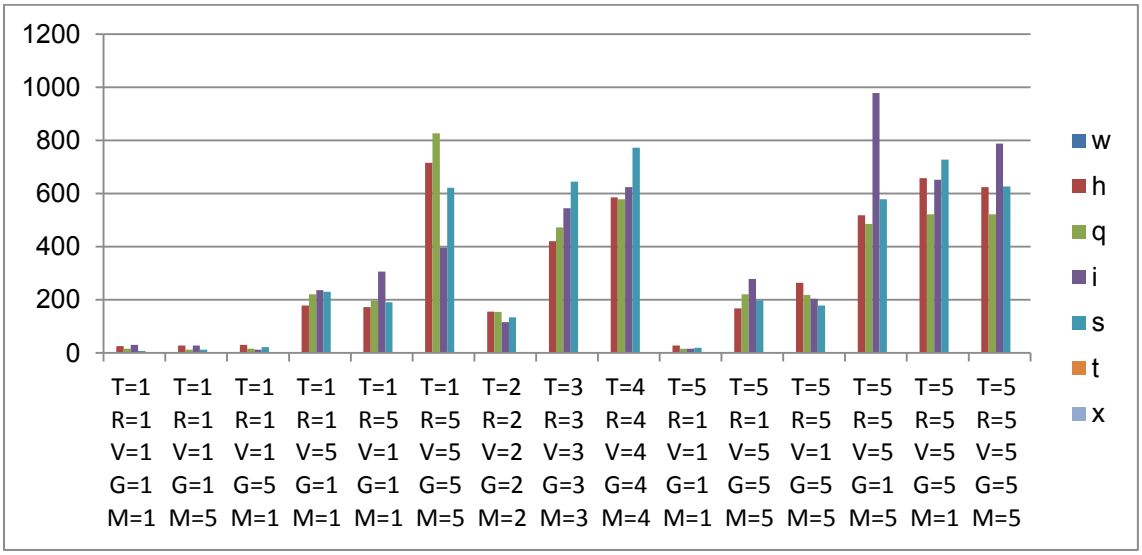

Figure 4-22 Chord Durations

This data reflects the effect of the mutation operator that changes durations and results in a more even spread of durations. The Variety and Repetition factors do not appear to impact the bias towards any particular duration.

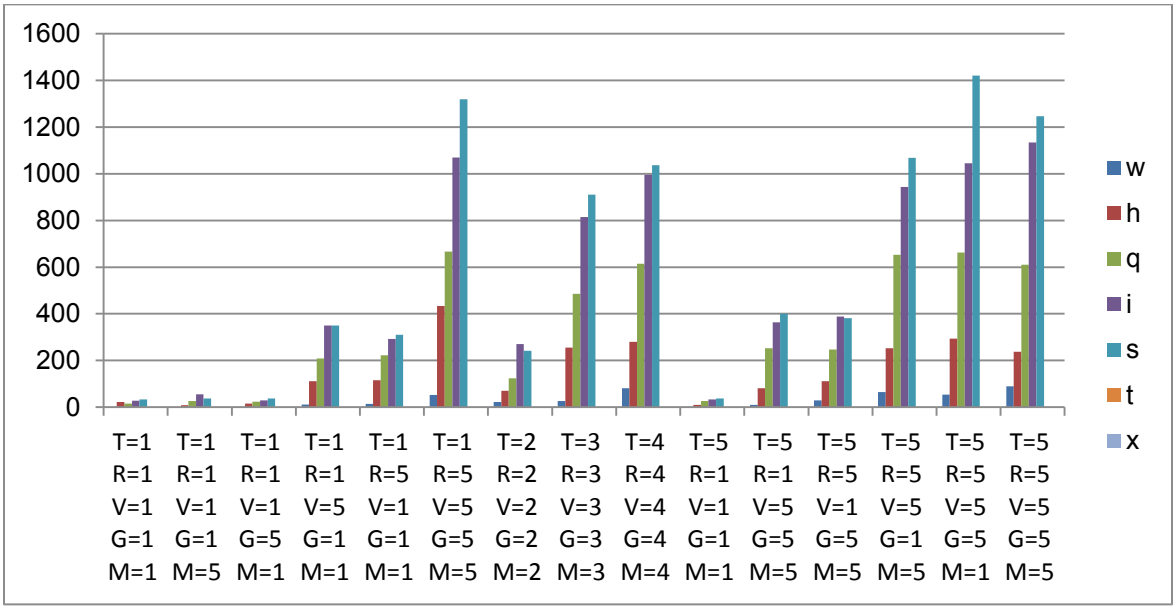

Figure 4-23 Note durations

4.2.3.4. Major and Minor Chords

Major and minor chord distributions reflect the preference parameter of Mood. Since the use of the key (Major or Minor) is determined in the same way it is for the other two approaches no significant difference was expected from the other approaches as shown in Figure 4-24

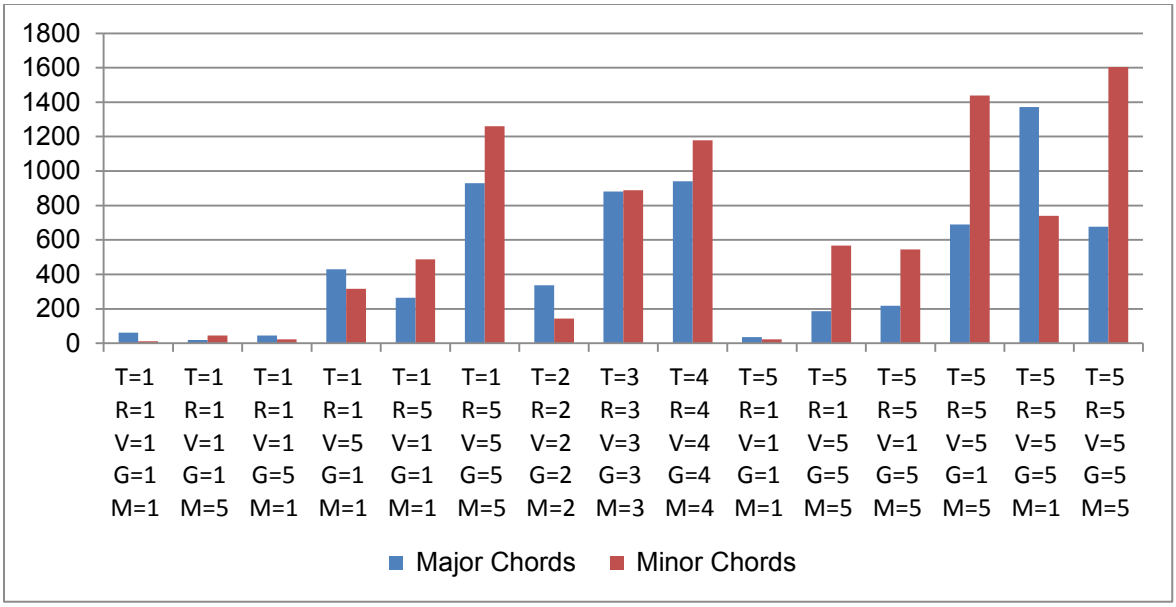

Figure 4-24 Major and minor chords used

4.2.3.5. Step Size

Compared to the other two approaches the step sizes don't vary as much between songs. The genetic algorithm does have fitness functions for Transition as well as it does for Repetition and Variety. However these fitness functions seem to have a cancelling effect on each other, resulting in a nearly even distribution of step sizes as shown in Figure 4-25

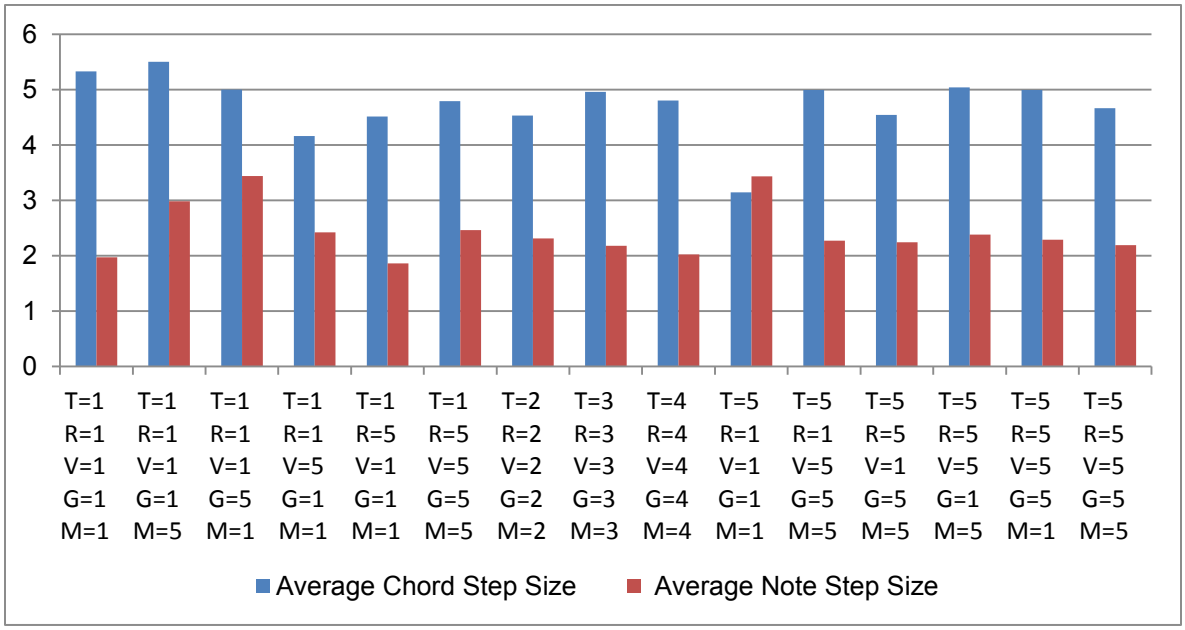

Figure 4-25 Chord and note step sizes

4.2.3.6. Chord and Note Octaves

The octaves are determined when the overall song structure is evolved. As a result the distribution is less uniform as compared to durations that are evolved at the melody and chord sequence level and looks similar to the octaves in the stochastic and planning approaches. Higher Ranges result in more octaves used by the chords and melody notes as can be seen in Figure 4-26 and Figure 4-27 respectively.

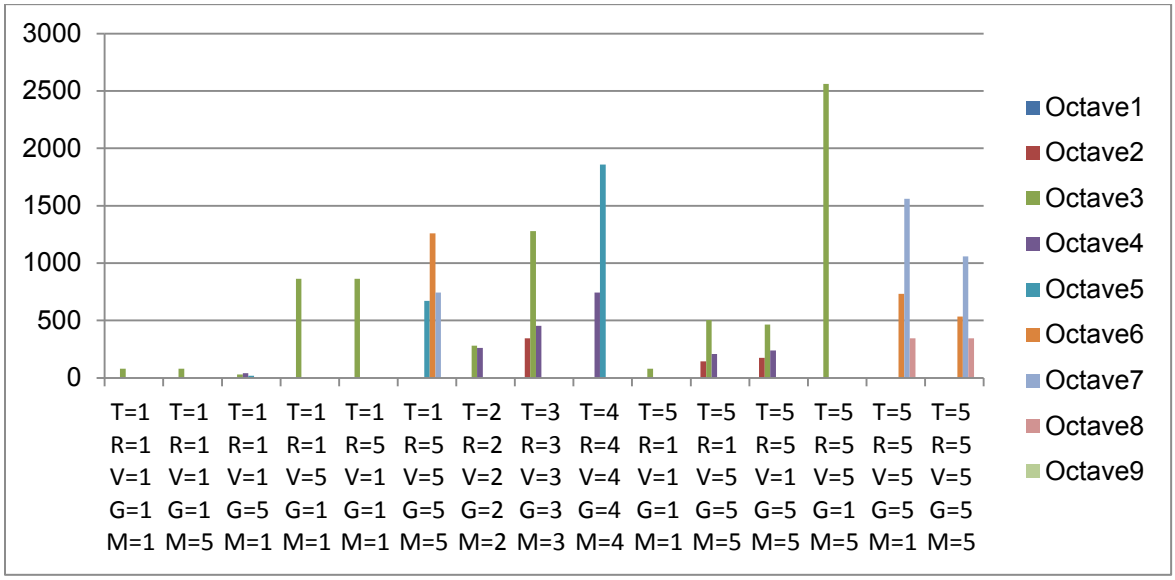

Figure 4-26 Chord octaves used

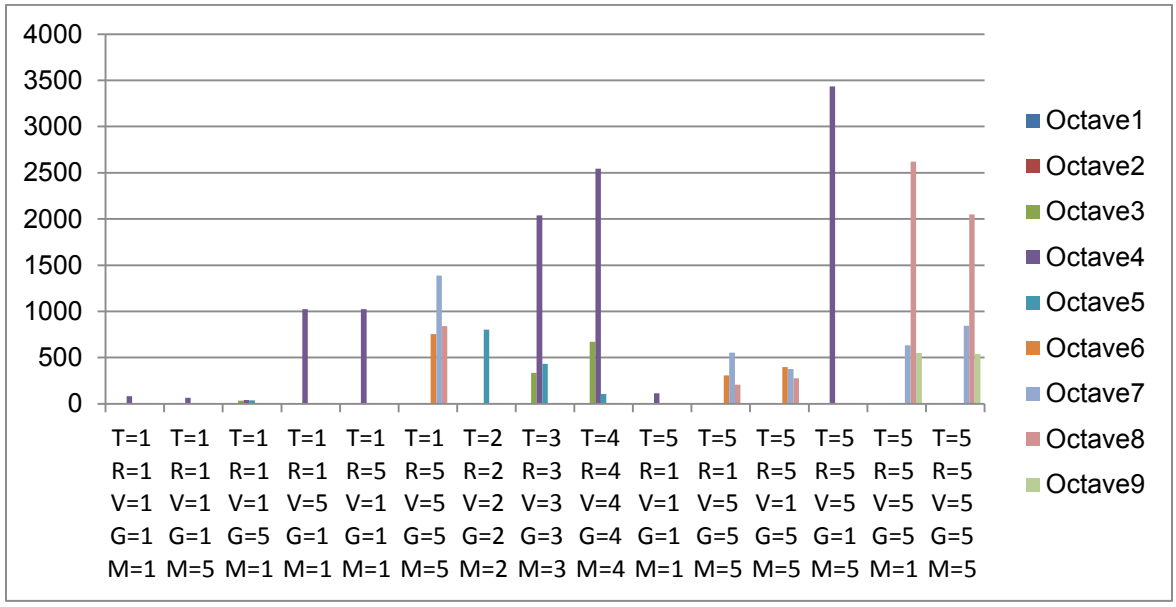

Figure 4-27 Note octaves used

4.2.4. Comparison of the Three Algorithms

We next compare the output of the three algorithms with each other and look at maximum/minimum values and averages where appropriate. The data from the charts on the individual algorithms is used to directly compare the values. Starting with the song component counts, as can be seen in Figure 4-28, the counts are similar in the minimum and maximum extremes. The measure counts for the planning are roughly half those of the stochastic and genetic algorithm which closely match each other as can be seen in Figure 4-29 Comparison of measure counts.

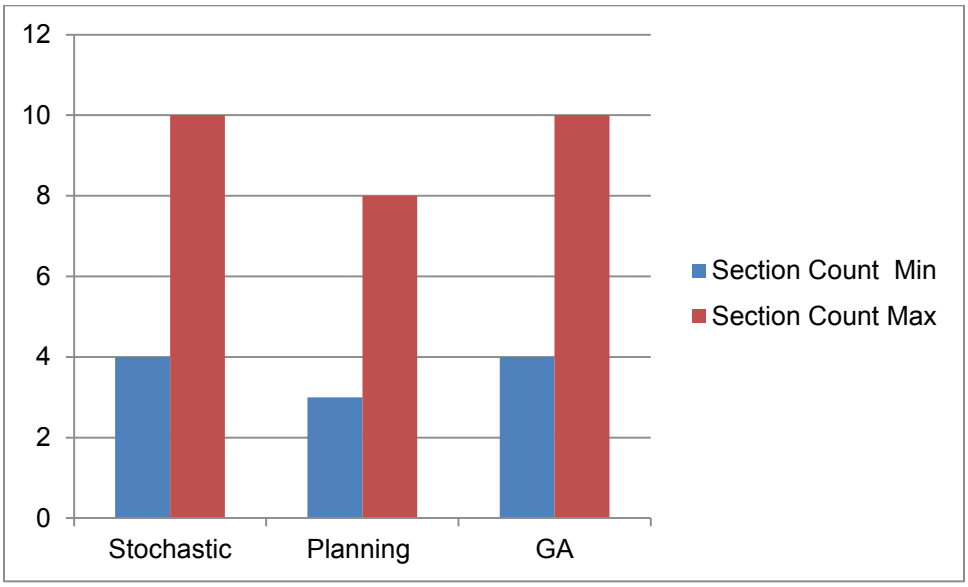

Figure 4-28 Comparison of the section counts

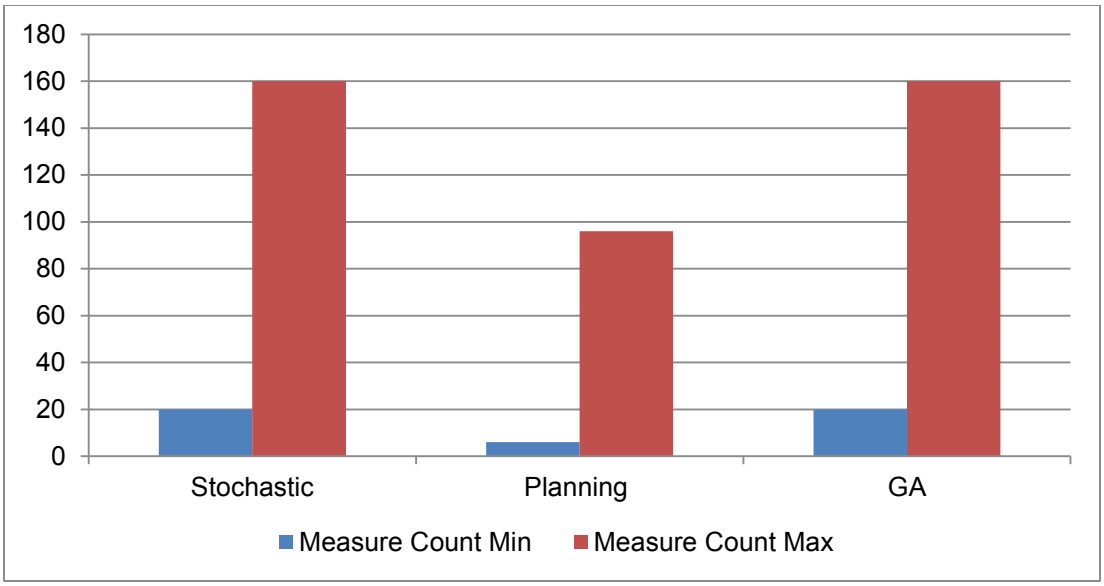

Figure 4-29 Comparison of measure counts

Next we compare the chord and note counts between the three algorithms. As can be seen in Figure 4-30 and Figure 4-31 minimum and maximum counts counts vary significantly with the planning having the most notes and chords while the stochastic has the fewest.

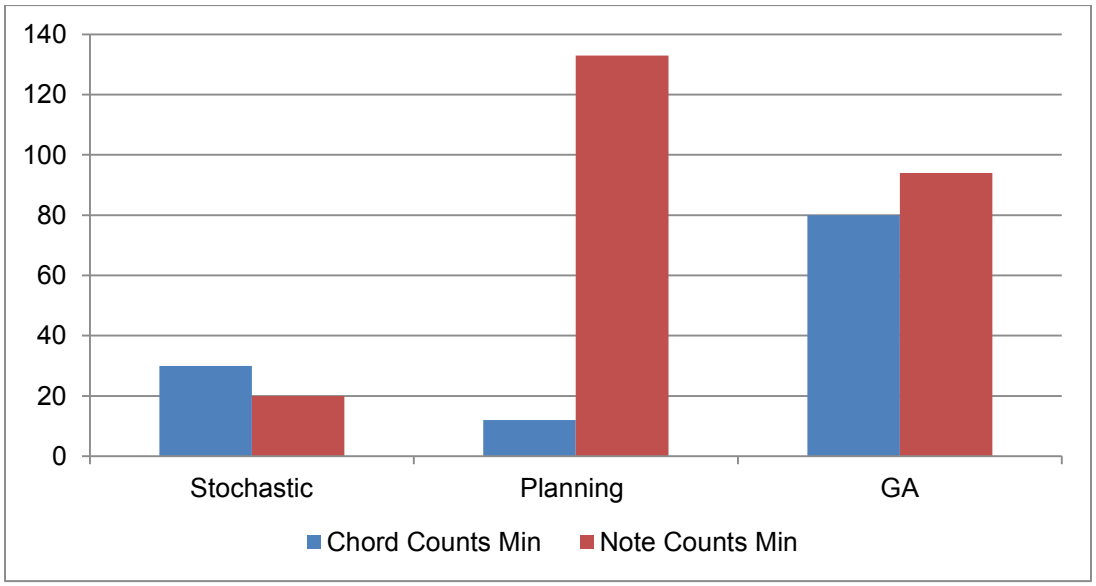

Figure 4-30 Minimum Chord and Note Counts

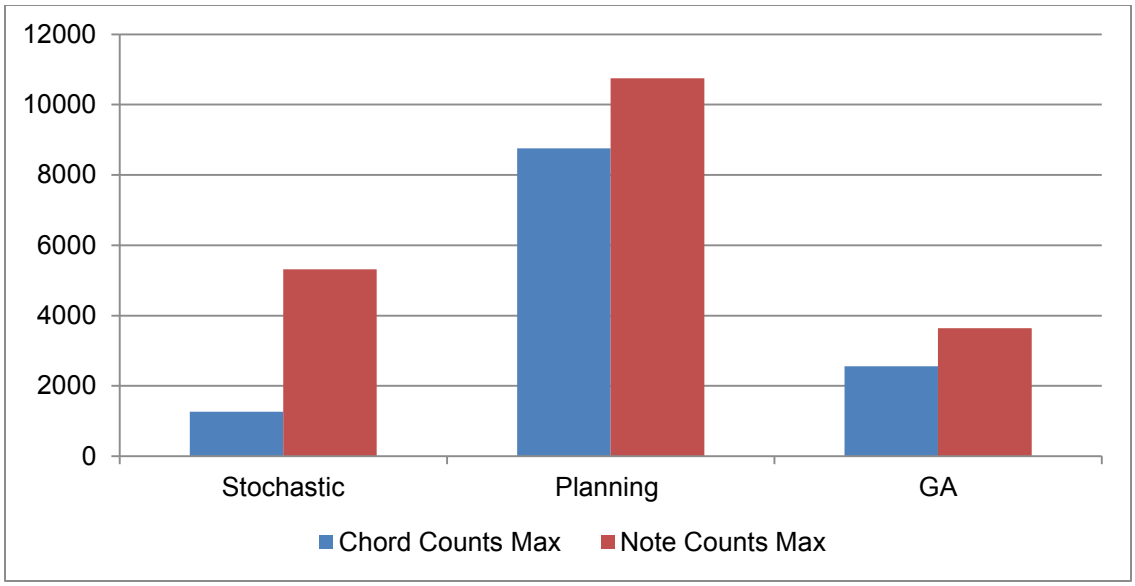

Figure 4-31 Maximum Chord and Note Counts

Looking at the variety of durations used between the algorithms as depicted in Figure 4-32 shows that the stochastic and genetic algorithm use a larger variety of durations on average than the planning algorithm.

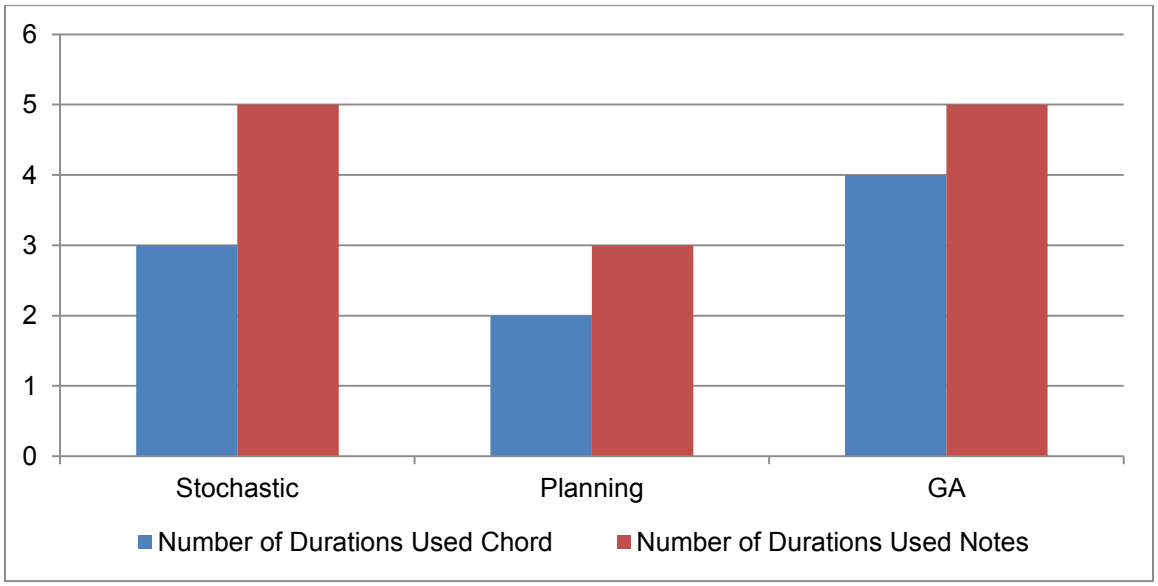

Figure 4-32 Comparison of Durations used

Next, we look at the step sizes between the algorithms in Figure 4-33. The average step sizes between the algorithms vary significantly. The genetic algorithm has a larger minimum

chord step size whereas the stochastic algorithm has the largest maximum chord step size. For the notes, the algorithms are producing a similar minimum step size while the maximum step size does vary significantly, with the stochastic algorithm having the largest step size. However the difference between the step sizes among notes is not as pronounced as it is with chords between the three algorithms.

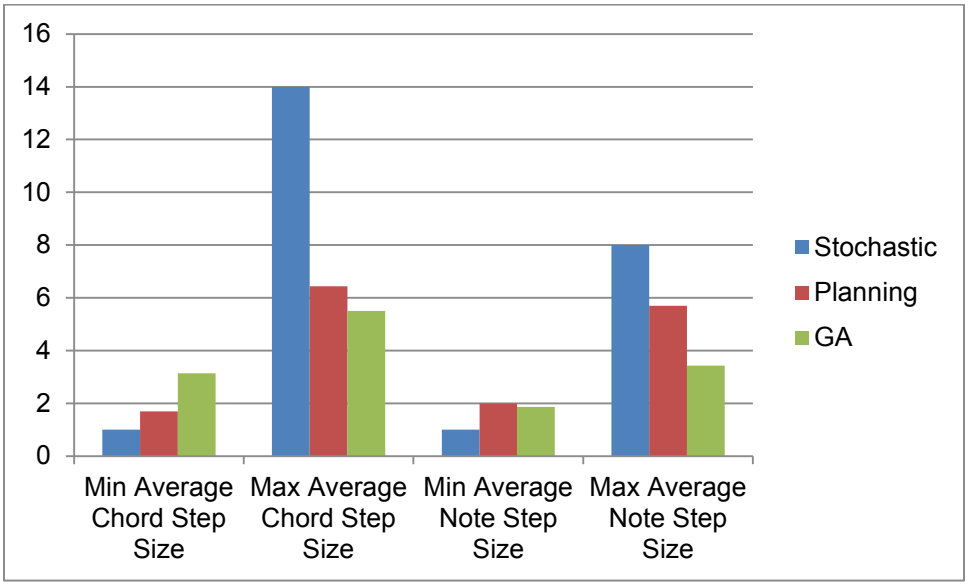

Figure 4-33 Comparison of step sizes

Figure 4-34 shows the differences between the major and minor chord counts of the three algorithms when producing songs that have few chords. The differences are minimal when the overall count is low and the planning has no difference. When producing songs with a high number of chords the differences are quite significant. As can be seen in Figure 4-35.The planning chord counts have the most difference since the planning system is producing the highest chord counts.

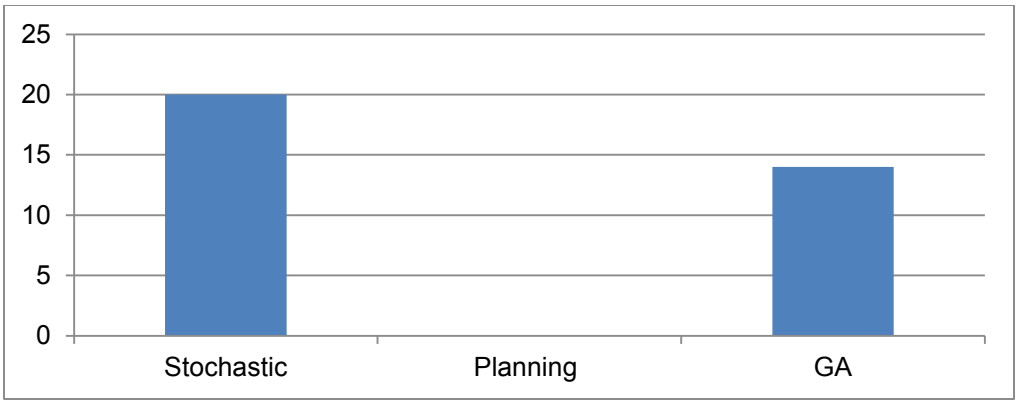

Figure 4-34 Minimum Chord Differences

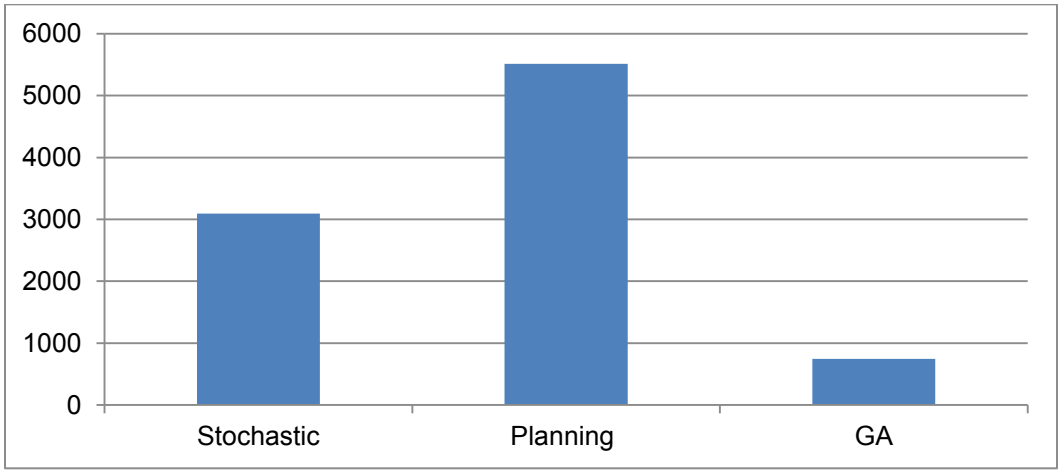

Figure 4-35 Maximum Chord Differences

Finally, Figure 4-36 shows the comparison between the variety of octaves generated by each of the algorithms. Since the Octave selection mechanism is similar in all the algorithms the variety of octaves between the algorithms is similar.

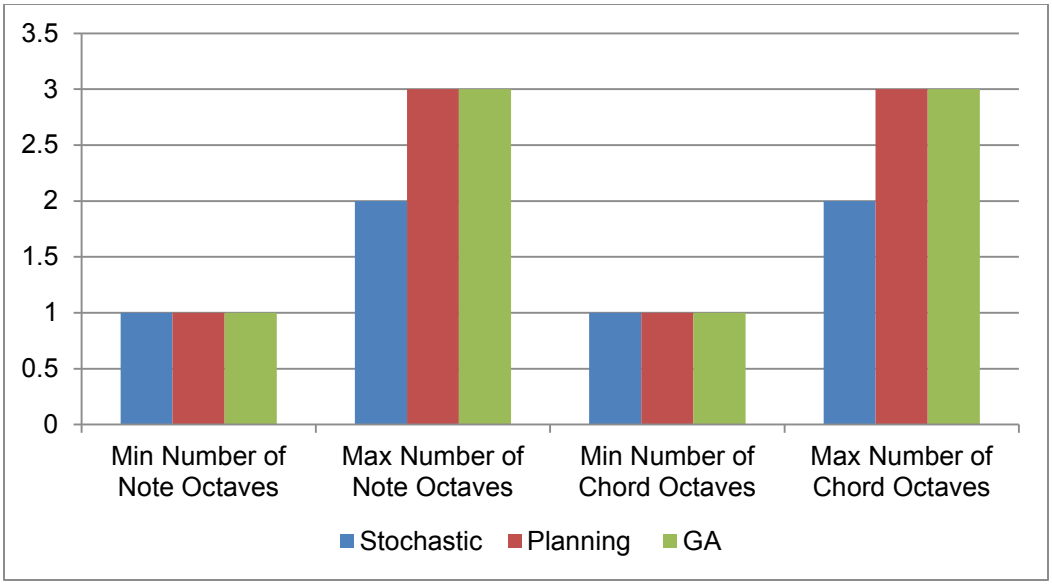

Figure 4-36 Comparison of different Octaves used by each algorithm

4.3. Subjective Results

Now that we have seen the impact of the different preferences on the compositions generated by MAGMA, we turn our focus to the quality of music that is generated. How 'listenable' a piece of music is, is ultimately a subjective observation. However, we can provide some measurements on these pieces that can be compared to the data provided in the objective analysis section of this chapter. Each of the songs described in the following sections were chosen after trying various combinations and generating about 15 to 20 songs for each algorithm and tweaking the preferences till the most pleasing song was produced. As before, we break down the analysis of the listenable songs based on algorithm, starting with the stochastic algorithm.

4.3.1. Common Factors

In order for each of the three algorithms in MAGMA to produce a pleasing sound relies on the delicate balance between preference attributes. Many trials with tweaking of the preferences are required to produce songs that can be considered of good quality. Additionally some preferences impact multiple aspects of the song which can lead to unintended effects. We describe here some of the findings from these experiments as they relate to each of the preferences.

Transition: Since this preference only impacts the step sizes between the chords and notes, the effects are easily predictable, namely larger step sizes can produce discordant pieces albeit more interesting ones. Smaller step sizes do produce smoother music and should be combined with a higher variety to ensure less boredom/monotony in the music.

Repetition: This preference impacts the overall song structure, the measure sequence and the chord and note values. Since this value is used in all of these layers anything but the smallest values of repetition such as 1 or 2 results in extremely repetitive music. For example with a repetition of 3 or higher, the song sections are repeated along with the measure sequences. Adding on to this repetitive chord and note sequences results in songs that are almost entirely a few notes/chords repeated over and over.

Variety: Like repetition, variety impacts all the layers of the song. Higher levels of variety have the opposite effect of the repetition which results in songs that have no discernible pattern to the listener.

Range: Range impacts the choices of instruments to be used. Higher ranges can result in more instruments being selected. This can be good if the instruments selected are compatible, otherwise songs with lower ranges sound just as good with a single or only two instrument changes. Along with instrument changes the range also impacts the number of octaves used. It is possible at high values of Range to get octaves that result in music that is too shrill if the instrument chosen has that timbre.

Mood: Since mood only impacts the tempo and the type of chords (major vs. minor) the selection of mood can help accentuate a good song, provided the other preferences are in balance. A mood that results in too high a tempo with high variety results in a lot of notes played at a high rate and sounds more like noise than music.

In the next subsections we look in detail at the attributes of the songs that were most pleasing. These can be compared to the objective analysis and provide insight into areas where MAGMA can be improved in the future.

4.3.2. Stochastic Algorithm Song

The variety of the songs produced by MAGMA using the stochastic approach is directly dependent on the transition matrix it uses which in turn is built from the parsing of existing MIDI songs. Currently the matrices are built from 30 popular songs. An attempt has been made to select songs that are quite varied so that the transition matrices can provide a varied range of possibilities.

Table 4-2 gives the attributes that produced the song chosen to be of good quality from the pool of stochastically generated songs. The song sections are limited to 7, including repeated sections, with the average number of measure in each section at 8. The mood was set at a medium setting so the major and minor chords are approximately evenly split. The low range value limits the song chord and notes to a single octave each.

Table 4-2 Stochastic song attributes

Attribute	Value
Preferences Used	Transition=3, Repetition=1, Variety=4, Range=1,Mood=3
Average Step Sizes	Chord=3.7, Notes=4.11
Number of song sections	7
Average number of measures per section	8
Total Number of Chords	280
Total Number of Notes	498
Chord Durations	Whole=69,Half =123, Quarter=83 ,qqq=5
Note Durations	Half=116,quarter=374,32nd=9
Chord Octaves	4 (only one octave)
Note Octaves	6 (only one octave)
Major / Minor Chord Counts	Major=165, Minor=115

4.3.3. Planning Algorithm Song

The variety of the songs using the planning approach is limited to the database of sequences that MAGMA contains. Since each song is tagged with an attribute for preferences the pool of songs the system can draw from for a particular preference is even smaller. However this algorithm produces a listenable song more frequently than the others. Furthermore, the same sequence can be listened to with different instruments and tempo by modifying the preferences of Mood and Range that do not rely upon the database.

Table 4-3 shows the details of the planning song that was one of the most pleasing to listen to. Low transition variety and repetition allows the song to have meaningful sequences in recognizable patterns, and have discernible start and endings. Chords are limited to one octave even though the range is high but due to lower variety there aren't as many chords to spread the octaves over. Same is the case with the notes which are limited to two octaves. The high value for Mood gives a slower tempo and mostly minor chords.

Table 4-3 Planning song attributes

Attribute	Value
Preferences Used	Transition=2, Repetition=1, Variety=2, Range=4,Mood=4
Average Step Sizes	Chord=1.79, Notes=3
Number of song sections	6
Average number of measures per section	3
Total Number of Chords	96
Total Number of Notes	241
Chord Durations	Half =52, Quarter=44
Note Durations	Half=11,Quarter=52,Eighth=74,Sixteenth=104
Chord Octaves	5 (only one octave)
Note Octaves	8 , 9
Major / Minor Chord Counts	Major=16, Minor=80

4.3.4. Genetic Algorithm

The genetic algorithm has the most potential to produce new and interesting music since it is not constrained by a database or existing plans of song components. However it also has the tendency to produce the most non coherent music if not constrained. Moderate values for transition and variety and a low value for repetition tends to produce the most melodic music.

Table 4-4 shows the attributes for the song generated from this algorithm with the most listenable quality.

Table 4-4 Genetic Algorithm Song

Attribute	Value
Preferences Used	Transition=3, Repetition=1, Variety=3, Range=5,Mood=3
Average Step Sizes	Chord=4, Notes=2.45
Number of song sections	6
Average number of measures per section	7
Total Number of Chords	336
Total Number of Notes	492
Chord Durations	Half =94, Quarter=116, Eighth=64, Sixteenth=62
Note Durations	Whole=4, Half=54, Quarter=88, Eighth=150 , Sixteenth=196
Chord Octaves	Octave 3
Note Octaves	Octave 9
Major / Minor Chord Counts	Major=166, Minor=138

4.3.5. Conclusions

MAGMA is capable of producing a variety of songs currently. However not all combinations produce songs that could be classified as 'listenable'. The factors of repetition and variety have the most overwhelming impact since they are implemented in all the layers of the song. The best combination has been determined to be the one using low repetition and a moderate level of variety. This combination results in songs that have enough repetition to produce recognizable patterns and enough variety to not be too monotonous. Smaller transition values also produce better songs in terms of consonance. Different mood values work equally well and produce songs of different tempo based on the value. High Range values can produces songs that have too high an octave and this may or may not work properly with the instrument that is selected as some instruments aren't very pleasant to listen to at high octaves, whereas others can be difficult to hear at the lower octaves. Ultimately allowing the user to tweak the values to their tastes through a friendly user interface will be the most efficient manner to experiment with the variety of songs that can be generated.

Each of the algorithms has its unique strengths and weaknesses. The stochastic algorithm is capable of producing interesting melodies and chord sequences for short sequences. If the sequences get too long then music theory conventions tend to be broken and bizarre sounding melodies are produced. The planning algorithm is capable of producing the most listenable music but is limited by the amount of data present in the databases. By increasing the variety of songs in the sequence databases the variety of the planning approach should improve as well. The genetic algorithm thanks to its fitness functions on music theory is capable of producing long sequences of chords and notes that sound melodious and harmonic, but if these sequences get too long then there is no pattern discernable to the listener.

4.4. Summary

This chapter explored the output of MAGMA by looking at the statistics on songs generated by various preferences from each of the three algorithms. The preferences as implemented within the system generate the type of songs that the user defines. However the subjective analysis of these songs show that only a narrow selection within the preference possibilities produces the most listenable music. Some of the preferences such as Variety and Repetition have a lot of influence over the output of the system to the point where they can make the contributions of the other preferences ineffective. This analysis has provided insight into areas of the system that require improvement and is explored in the next chapter.

Chapter 5. Conclusions and Future Work

5.1. Conclusions

The goal of this thesis has been to demonstrate AI approaches to the automatic composition of music. Music composition is a distinctly creative act when executed by humans. Since computers have no mechanism for creativity, AI techniques have to be employed to compensate for this deficiency. AI in itself is a vast and ever growing field of research. By utilizing a subset of the techniques available in AI we can explore and compare the efficacy of different approaches. The software implemented for this thesis, MAGMA, has been built using the AI approaches of stochastic reasoning, planning and genetic algorithms to generate songs in the genre of popular music.

As seen in the data presented in chapter 4, MAGMA is able to produce music that conforms to the preference values input by the user. The combination of the various preference values result in a diverse output in terms of length of song, its melodiousness, consonance and instrument or octave ranges.

As also described in chapter 4 these preferences and the outputs they result in, do not necessarily constitute listenable or high quality music. In fact the experimentation has shown that in MAGMA's current incarnation there is only a small subset of preference choices that produce good quality songs.

In addition to this, each of the algorithms has its strengths and weaknesses when it comes to the quality of music it produces. The stochastic algorithm for example is not directly bound to the structure of the data that is used to build the transition matrices and can produce new music based on the probabilities of the existing data set even with small sample of existing data. The planning algorithm is limited to producing variations on the music that is presented in the sequence of chords and notes in the database but produces listenable music more often than any of the other algorithms. The genetic algorithm is capable of producing interesting music that is not dependent on any prior music but is also prone to generating compositions without any discernable patterns.

The plan decomposition method of generating the song in MAGMA works well for the relatively simple structure of popular music. By generating the song structure first, followed by the measure sequence for each song section and then the chord and melody for each measure it has been straightforward to implement the different algorithms. Indeed, if other algorithms are

desired to be added to MAGMA this can be accomplished without having to make any changes to the algorithms in the existing implementation.

The user selected preferences are also effective in impacting the characteristics of the output music. However some of the preferences like repetition and variety tend to impact the song more than others. Further work will need to be done to determine if more and/or a different set of preferences are needed and how they should interact with each other.

5.2 Future Work

The ultimate goal of a music compositional system should be to consistently produce music that is of listenable quality and accurately reflect the characteristics desired by the user. This implies that no matter what combination of preferences are input by the user the music generated should be considered by a majority of listeners to be of acceptable quality. This further indicates that for any music composition system to be considered mature, it has to pass the acceptability threshold by multiple users and not just the developers of the system. Future work on MAGMA falls into these two categories: produce better music consistently, and be tested by multiple users.

The major drawback in the current implementation of MAGMA is the lack of a context other than what is provided by the preferences selected by the user. As indicated earlier and described in detail in chapter 3, the music produced by MAGMA via plan decomposition constructs the song structure first. This step sets up the sections of the song and allows the future steps to know how many distinct and repeated sections there are.

The first section by default is the intro to the song. Similarly the last section is the outro. However there is no other facility that imparts additional features to the intro or outro of the song. In most popular music the intro is distinct and sets up for the listener expectations on what the rest of the song will sounds like. In MAGMA, the intro is just another section of the song that happens to come first.

The same is true for all the other sections of the song and so there is no consistent theme or common pattern between the sections. In songs produced by MAGMA that happen to sound good, it is only through random luck that the variety among the sections works to produce a pleasing pattern. The same issue is prevalent in the measure sequences that are built for each song section. For each unique measure in the measure sequence the chord and melody sequence is constructed without prior context of the measure that preceded it.

For better results MAGMA will have to employ a mechanism that keeps track of the artifacts generated in the past and allow them to influence the artifacts that are generated in the future. This should be implemented at all layers. So when the first section of the song i.e. the intro, is built the subsequent sections should have characteristics that are related to it. Similarly the verse and chorus sections should complement each other as well. It remains to be seen how much complexity this adds to the system.

Another area of improvement is the selection of the preferences and the interactions between them. Currently all the preferences are applied to all steps in the plan decomposition. For example repetition value of 1 will have no repeated song sections and no repeated measures. The chord and note sequences will be built with very low repetition as well. This creates songs with no recognizable pattern. On the opposite end, a repetition value of 5 will have multiple repeated song sections, multiple repeated measures and chords and melody with a lot of repeated chords and notes. This creates extremely monotonous music. A better option, perhaps, would be to allow the user to select the preferences and target them to a specific step in the plan decomposition. This way one could have repetitive chord/melody but non-repetitive song or measure sequences. A further alternative would be to allow the user to choose which layer of the song to apply the preferences to and use some heuristic to determine the preferences for the other layers.

The next area of improvement would be to allow the user to combine the algorithms in different stages of the plan decomposition to take advantage of each algorithm's strength. For example the GA provides the most diverse range of chord and melody sequences. However there are not that many different combinations of song structure or measure sequences. Allowing the user to build the song structure and measure sequence using the planning algorithm and then build the chord and melody sequences using the genetic algorithm could potentially provide interesting variety of songs than planning alone does; while at the same time are more controlled than the genetic algorithm. Within the algorithms themselves, more can be done to improve the output. For the planning algorithm, adding more songs to the database will improve the variety in the output. Similarly 'harvesting' melody sequences from additional and more varied MIDI songs can benefit the output of the stochastic algorithm as well. Adding different rules to the music theory fitness function in the genetic algorithm, such as rules of chord and melody harmonization can positively impact the output generated there as well. All the algorithms can also benefit from the melody sequence generation to be influenced by the chord sequence beyond just using the same key, as is currently done.

Finally, MAGMA should be tested by a substantial variety of users and feedback

obtained to improve it beyond the subjective preferences of its developers. In order to facilitate this, the system needs to be packaged into user friendly software such as a graphical user interface or a web based application. The system can also be built to solicit data from the user, such as the user providing chord and melody sequences, to enrich the database of the system. If MAGMA is deployed as a web based application then these additions by one user can be available to all users. Ultimately, simplifying the user interface of the system will allow it to be used by a diverse group of users and provide for a qualitative analysis of the system based on feedback from a larger set of participants.

Appendices

Appendix A: Songs used for Melody Pitches and Durations in the Stochastic algorithm

Artist	Song
ABBA	Chiquitita
ABBA	Fernando
John Barry	You Only Live Twice
The Beatles	Hey Jude
The Beatles	Wait
Bon Jovi	Living On A Prayer
Celine Dion	My Heart Will Go On
Celine Dion	Power of Love
Coldplay	Clocks
Eminem	Without Me
Enya	May it Be
King Crimson	21st Century Schizoid Man
Madonna	Get into the Groove
Madonna	Like a Prayer
Madonna	Like a Virgin
Metallica	Seek and Destroy
Michael Jackson	Billy Jean
Michael Jackson	You Are Not Alone
Pet Shop Boys	Se Vida Ei
Pet Shop Boys	Go West
Pink Floyd	Another Brick In the Wall
Queen	Bohemian Rhapsody
Queen	Crazy Little Thing Called Love
Simon and Garfunkel	Sounds Of Silence
Frank Sinatra	Girl from Ipanema
Frank Sinatra	Love and Marriage

Joe South	Rose Garden
Bruce Springsteen	Born in the USA
Sting	Desert Rose
Andrew Lloyd Webber	Phantom of the Opera

Appendix B: Songs used for the Chord Pitches and Durations in Stochastic Algorithm

Artist	Song
Adele	Rolling In The Deep
Adele	Set Fire To The Rain
Adele	Skyfall
The Beatles	Across The Universe
The Beatles	Let It Be
Billy Joel	Piano Man
Bon Jovi	Living On A Prayer
Bryan Adams	Everything I Do
Celine Dion	My Heart Will Go On
Deep Purple	Smoke On The Water
Eric Clapton	You Look Wonderful Tonight
Gotye	Somebody That I Used To Know
Guns N Roses	November Rain
Jimi Hendrix	All Along The Watchtower
Journey	Dont Stop Believing
Madonna	Girl Gone Wild
Mumford And Sons	The Cave
Nintendo	Guiles Theme
Pink Floyd	Wish You Were Here
Shakira	Waka Waka
The Police	Every Breath You Take

Appendix C: Songs used for the Chord and Melody Sequences in the Planning Algorithm

Artist	Song
Adele	Rolling In The Deep
Adele	Set Fire To The Rain
Adele	Skyfall
The Beatles	Across The Universe
The Beatles	Let It Be
Billy Joel	Piano Man
Bon Jovi	Living On A Prayer
Bryan Adams	Everything I Do
Celine Dion	My Heart Will Go On
Deep Purple	Smoke On The Water
Eric Clapton	You Look Wonderful Tonight
Gotye	Somebody That I Used To Know
Guns N Roses	November Rain
Jimi Hendrix	All Along The Watchtower
Journey	Dont Stop Believing
Madonna	Girl Gone Wild
Mumford And Sons	The Cave
Nintendo	Guiles Theme
Pink Floyd	Wish You Were Here
Shakira	Waka Waka
The Police	Every Breath You Take

References

[1] Luger, George F. “Artificial intelligence: Structures and strategies for complex problem solving”. Addison-Wesley Longman, 2005.

[2] Turing, A.M. “Computer machinery and intelligence”, Mind, LIX, 236 (1950), 433–460.

[3] Hiller, Lejaren, and Leonard Maxwell Isaacson. “Illiac suite, for string quartet.” Vol. 30. No. 3. New Music Edition, 1957.

[4] Edwards, Michael. "Algorithmic composition: computational thinking in music." *Communications of the ACM* 54.7 (2011): 58-67.

[5] BRINGSJORD, SELMER. "Chess Is Too Easy." MIT's Technology Review 101 (1998): 2.

[6] Minsky, Marvin L. "Why people think computers can't." AI Magazine 3.4 (1982): 3.

[7] Eaglestone, B. M., et al. "Composition systems requirements for creativity: what research methodology." Proceedings of Mosart Workshop on Current Research Directions in Computer Music, Barcelona. 2001.

[8] Chaslot, Guillaume, et al. "Monte-carlo tree search: A new framework for game ai." Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference. 2008.

[9] Ritchie, Graeme. "The JAPE riddle generator: technical specification." Institute for Communicating and Collaborative Systems (2003).

[10] Cohen, Harold. "A self-defining game for one player." Proceedings of the 3rd conference on Creativity & cognition. ACM, 1999.

[11] Turner, S. R. 1994. “The Creative Process: A Computer Model of Storytelling and Creativity”. Lawrence Erlbaum Associates.

[12] Meehan, J. R. 1976. “The Metanovel: Writing Stories by Computer”. Ph.D. Dissertation, Dept. of Computer Science, Yale University.

[13] Muscutt, Keith. "Composing with algorithms: An interview with David Cope." Computer Music Journal 31.3 (2007): 10-22.

[14] "MusicRadar.com." A Brief History of Computer Music. Web. 24 Mar. 2013.< http://www.musicradar.com/news/tech/a-brief-history-of-computer-music-177299>

[15] "First Digital Music Made in Manchester." (Digital 60). Web. 24 Mar. 2013. <http://www.digital60.org/anniversary/digital_music.html>

[16] "History of MIDI." History of MIDI. N.p., n.d. Web. 26 Mar. 2013.<http://www.midi.org/aboutmidi/tut_history.php>.

[17] Miranda, Eduardo Reck, and Marcelo M. Wanderley. “New digital musical instruments: control and interaction beyond the keyboard”. Vol. 21. AR Editions, Inc., 2006.
[18] Maurer, John. "A brief history of algorithmic composition." (1999).< https://ccrma.stanford.edu/~blackrse/algorithm.html>
[19] Papadopoulos, George, and Geraint Wiggins. "AI methods for algorithmic composition: A survey, a critical view and future prospects." AISB Symposium on Musical Creativity. Edinburgh, UK, 1999.
[20] Holland, John H. "Genetic algorithms." Scientific American 267.1 (1992): 66-72.
[21] Hazewinkel, M. ed. (2001), "Markov chain", Encyclopedia of Mathematics, Springer.
[22] Brinkop, A., Laudwein, N., and Maasen R (1995). "Routine Design for Mechanical Engineering." AI Magazine, p 74-85, AAAI.
[23] Ferrucci, David, et al. "Building Watson: An overview of the DeepQA project." AI magazine 31.3 (2010): 59-79.
[24] Dasgupta, Sanjoy, Christos H. Papadimitriou, and Umesh Virkumar Vazirani. "Algorithms." (2006).
[25] Russell, Stuart Jonathan, et al. “Artificial intelligence: a modern approach”. Vol. 2. Englewood Cliffs: Prentice hall, 2010.
[26] Newell, Allen, John C. Shaw, and Herbert A. Simon.” Report on a general problem-solving program”. Rand Corporation, 1959.
[27] Brooks, Rodney. "A robust layered control system for a mobile robot". Robotics and Automation, IEEE Journal of 2.1 (1986): 14-23.
[28] Hendler, James A., Austin Tate, and Mark Drummond. "AI planning: Systems and techniques." AI magazine 11.2 (1990): 61.
[29] Fikes, Richard E., and Nils J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." Artificial intelligence 2.3 (1972): 189-208.
[30] Tomassini, Marco. "A survey of genetic algorithms." Annual Reviews of Computational Physics 3.2 (1995): 87-118.
[31] Holland, John H. "Genetic algorithms." Scientific american 267.1 (1992): 66-72.
[32] Conway, John. "The game of life." Scientific American 223.4 (1970): 4.
[33] McAlpine, Kenneth, Eduardo Miranda, and Stuart Hoggar. "Making music with algorithms: A case-study system." Computer Music Journal 23.2 (1999): 19-30.
[34] Serra, Marie-Hélène. "Stochastic composition and stochastic timbre: Gendy3 by Iannis Xenakis." Perspectives of New Music (1993): 236-257.

[35] Bokesoy, Sinan, and Gerard Pape. "Stochos: Software for Real-Time Synthesis of Stochastic Music." Computer Music Journal 27.3 (2003): 33-43.
[36] "Webmonkey." Webmonkey Conways Game of Life in HTML5 Comments. N.p., n.d. Web. 13 Apr. 2013. < http://www.webmonkey.com/2010/07/conways-game-of-life-in-html5/>
[37] Hiller, Lejaren A., and Robert A. Baker. "Computer Cantata: A study in compositional method." Perspectives of New Music 3.1 (1964): 62-90.
[38] Ebcioğlu, Kemal. "An expert system for harmonizing four-part chorales." Computer Music Journal 12.3 (1988): 43-51.
[39] Moorer, James Anderson. "Music and computer composition." Communications of the ACM 15.2 (1972): 104-113.
[40] Bharucha, Jamshed J. "MUSACT: A connectionist model of musical harmony." Machine models of music. MIT Press, 1992.
[41] Hörnel, Dominik, and Thomas Ragg. "Learning musical structure and style by recognition, prediction and evolution." Proceedings of the International Computer Music Conference. INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1996.
[42] Hild, Hermann, Johannes Feulner, and D. Menzel. "{HARMONET}: a neural net for harmonising chorales in the style of {JS Bach}." (1992).
[43] Arcos, Josep Lluís, Ramon Lopez De Mantaras, and Xavier Serra. "Saxex: A case- based reasoning system for generating expressive musical performances." Journal of New Music Research 27.3 (1998): 194-210.
[44] Narmour, Eugene. “The analysis and cognition of melodic complexity: The implication-realization model”. University of Chicago Press, 1992.
[45] Biles, John. "GenJam: A genetic algorithm for generating jazz solos." Proceedings of the International Computer Music Conference. INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1994.
[46] "JFugue - Java API for Music Programming." JFugue - Java API for Music Programming. N.p., n.d. Web. 15 Apr. 2013.
[47] "What Is Pop Music?" About.com Top 40 / Pop. N.p., n.d. Web. 21 Apr. 2013. <http://top40.about.com/od/popmusic101/a/popmusic.htm >
[48] "Music Theory You'll Enjoy." Hooktheory.com. N.p., n.d. Web. 23 Apr. 2013. http://www.hooktheory.com/
[49] "The Open Music Encyclopedia." Musipedia: Musipedia Melody Search Engine. N.p., n.d. Web. 23 Apr. 2013.< http://www.musipedia.org/>
[50] "MuseScore." MuseScore. N.p., n.d. Web. 23 Apr. 2013.< http://musescore.org/ >

[51] "MusicXML for Exchanging Digital Sheet Music." MusicXML. N.p., n.d. Web. 23 Apr. 2013 http://www.musicxml.com/
[52] Wolf, Denis F., et al. "Autonomous terrain mapping and classification using hidden markov models." Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on. IEEE, 2005.
[53] Rich, Elaine, and Kevin Knight. "Artificial intelligence." Computer Science Series. McGraw-Hill 8 (1991).
[54] Knoblock, Craig A. "An analysis of ABSTRIPS." Proceedings of the first international conference on Artificial intelligence planning systems. 1992.
[55] Brinkop, Axel, Norbert Laudwein, and Rudiger Maasen. "Routine design for mechanical engineering." AI Magazine 16.1 (1995): 74.
[56] Georgeff, Michael P., and Amy L. Lansky. "Reactive reasoning and planning." Proceedings of the sixth national conference on artificial intelligence (AAAI-87). Vol. 677682. 1987.
[57] Math - 596 : Lectures." Math - 596 : Lectures. N.p., n.d. Web. 21 May. 2013. <http://www-rohan.sdsu.edu/~rcarrete/teaching/M-596_patt/lectures/lectures.html
[58] Chandrasekaran, B., et al. "Building routine planning systems and explaining their behaviour." International journal of man-machine studies 30.4 (1989): 377-398.
[59] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." Computational Intelligence and AI in Games, IEEE Transactions on 4.1 (2012): 1-43.
[60] "Home." Groovy -. N.p., n.d. Web. 25 May 2013.< http://groovy.codehaus.org/ >
[61] "How To Write Songs Like The Pros Using Unity And Variety." N.p., n.d. Web. 08 Oct. 2013. http://www.ultimate-guitar.com/lessons/songwriting_-_lyrics/how_to_write_songs_like_the_pros_using_unity_and_variety.html
[62] Hewitt, Michael. “Music Theory for Computer Musicians”. Course Technology PTR, 2008
[63] "Some Melody-writing Hints." Melody Guidelines. N.p., n.d. Web. 09 Oct. 2013. <http://www.donaldsonworkshop.com/coriakin/melody.html>