Composing Music in Constrained Search Environments

Jeffrey Keene
University of Denver

Composing Music in Constrained Search Environments

A Thesis

Presented to

the Faculty of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jeff Keene

August 2010

Advisor: Mario Lopez, Chris GauthierDickey

Author: Jeff Keene
Title: Composing Music With Computers in Extremely Constrained Search Environments.
Advisor: Mario Lopez, Chris GauthierDickey
Degree Date: August 2010

Abstract

Composing music with computers in constrained search environments adds complexities and problems not present in the traditional problem domain of generative music. The traditional and well researched mechanisms of Markov chains, genetic algorithms and data driven rule based systems do not directly map to a problem domain in which pitch choice and rhythm choice are likely to be highly limited.

We therefore explore several possible solutions to generating rhythms in extremely constrained environments with the goal of generating music that adheres to user specified constraints and is aesthetically pleasing.

Contents

1 Introduction **1**

1.1 Motivation and Goals 1

2 Related Work **4**

2.1 Rule Systems and Constrain Satisfaction 4

2.1.1 Constraint Satisfaction in Music Composition 5

2.1.2 Applicability to our goals 6

2.2 Local Search Algorithms 7

2.2.1 Applicability to our goals 12

2.3 Neural Nets 14

2.3.1 Neural Nets and Music 16

2.3.2 Applicability to Our Goals 17

2.4 Depth First Searching 17

2.4.1 Algorithm 17

2.4.2 Backtracking 18

2.4.3 Performance 21

3 Generating Rhythm **23**

3.1 Commonality 23

3.2 Simple Search 24

3.3 Markov Chains 25

3.3.1 Construction 26

3.4 Concrete Markov Search 26

3.4.1 Concrete Markov Mining 27

3.4.2 Search Process 27

3.5 Relational Markov Search 29

3.6 Syncopation Search 30
3.7 Experimental Results 31
3.7.1 Data Sets 32

4 Algorithm Performance **44**
4.1 Data Set 1 45
4.2 Data Set 2 45
4.3 Data Set 3 45

5 Rhythms Survey **53**

6 Survey **54**
6.1 Output Parameters 55
6.2 t test 55
6.3 Simple Search and Relational Search 56
6.4 Simple Search and Syncopation Search 56
6.5 Relational and Syncopation Search 57

7 Conclusion **59**
7.1 Future Work 60
7.1.1 Data Sets 60
7.1.2 Rest in Rhythms 61
7.1.3 Syncopation Heuristic 61
7.1.4 Polyphony 61

8 Rhythm Event Selection Source Code **63**
8.1 Concrete Search Weighting Source Code 65
8.2 Relational Search Weighting Source Code 67
8.3 Syncopation Search Weighting Source Code 70
8.4 Syncopation Measurement 74

8.5 Markov Table Creation 76
8.6 Survey Sample Question 78

1 Introduction

Computers have been put to the task of generating music and analyzing music since the early 1960s (Roads [28, 27], Moorer [24], Camurri [7]). The most notable successes within the field build large databases of musical features through offline analysis of substantive data sets and use rule sets derived thereby to generate pitch, rhythm and form (ref Cope [9, 10, 11, 12, 13]). It is outside the scope of this paper to discuss the efficacy by which these systems meet their goal of creating art; suffice it to say that given large sets of stylistically similar music and rule inference systems, computer programs are able to create aesthetically pleasing music compositions.

We are exploring music composition guided by a fundamentally different goal; that of pedagogy. Pedagogical exercises (hereafter referred to as études) typically constrain themselves to specific rhythmic sets, specific interval sets or harmonic areas with the purpose of training a specific instrumental or music reading skill. Études may or may not be of particular musical interest to the student, but a specific additional goal of our work is to produce études of pleasant aesthetic quality.

1.1 Motivation and Goals

The context in which the algorithms discussed in this paper were implemented is a music composition program designed to generate sight reading études for stringed instruments. The user, likely a teacher, creates a configuration file specifying the form of the étude to be created, what rhythmic values to use, what portion of the finger board to use and other options extraneous to this paper. The set of legal rhythms supplied by the user is the only set of constraints with which we will concern ourselves in this text.

The principle goal of the software is to give music teachers a tool for generating sight reading exercises of perceived infinite variety but also of sufficient musical interest to motivate a student to read them. The goals specific to the rhythm generation algorithms follows therefrom:

1. Rhythms output by the composition software must use only rhythms specified by the user. Rhythm generation algorithms must be able to recognize a set of legal rhythm events and constrain composition to those events.

2. Each rhythm event specified by the user must be present in the output. Users specify rhythm events for use in the composition software for private pedagogical goals that cannot be predicted by the algorithm's implementation. An algorithm's failure to produce expected output diminishes the usefulness of the software considerably.

3. Each run of the algorithm should produce pseudo random output. Otherwise the goal of creating site reading études cannot be met because a sufficiently varied body of output is not being produced. This goal is the most difficult to quantify and prove because for a given set of legal rhythm events specified by the user a finite number of rhythm strings can be generated (though the number of rhythm strings is likely to be very large). For our purposes, we are satisfied if it appears to a user using the software under normal conditions that the output is unique between subsequent runs.

Music composition as a topic of artificial intelligence has been approached with neural nets (section 2.3), local search algorithms (section 2.2), and rule based systems (section 2.1). Such systems will subsequently be discussed but for now is it sufficient to state that not all algorithms handle well environments where the output is highly constrained by user input. Previous artificial intelligence systems could be used for such a purpose (we shall see that one of our methods is a data driven rule based system), but each has problems with its traditional approach that would render our tool cumbersome and unusable to the average pedagog. We felt compelled to look at developing algorithms designed specifically for the task of generating highly constrained musical output.

In particular, we explore the use of modified Markov algorithms in the generation of rhythms within études (section 3.3) We implemented a number of rhythm generating algorithms using tables computed from a subset of the J.S. Bach Chorales, which yield a surprising variety of rhythm shapes when used by the composition software. The efficacy of the algorithms is measured by using the algorithms to generate rhythm sets comprised of rhythm durations not prevalent or not at all present in the original data set (section 3.7).

We also explore a rhythm generating algorithm that is syncopation driven (section 3.6). No Markov process is used; rather, events are selected based on where the algorithm is in the search process and how much syncopation has been accumulated so far. A simple heuristic is employed to determine when the algorithm should select notes that emphasize syncopation and when to demphasize syncopation (Gomez [16]). This has the effect of providing an ordering to the rhythm event output that does not require large amounts of offline analysis as in the case of a Markov table ordering mechanism.

Measuring the aesthetic effectiveness of the algorithms was done using a survey with volunteer participants. Sample outputs of each algorithm were matched with sample outputs of each other algorithm and the participants were asked to select which sample they liked more. More details of the survey are discussed in the final section (section 6).

2 Related Work

This section will describe the parameter's of music composition as a field of study in artificial intelligence. The historic time line and personnel of computer music as it pertains to the techniques subsequently discussed will be touched upon, but time and relevance prevent us from providing a detailed discourse on the history of computers and musical art as would be present in a work on music history. For a detailed account of the use of computers in music performance and composition consult Cope [8], Lincoln [22], Ames [3], and Roads [28] [27].

Our discussion covers rule based constraint satisfaction problems, neural nets and local search algorithms. In the course of our research we have concluded that these areas of artificial intelligence research comprise the vast majority of research in computer music (Cope [8], Roads [28]). In each case a high level description of the artificial intelligence technique will be provided as well as a brief history of its application in computer music.

2.1 Rule Systems and Constrain Satisfaction

Constraint satisfaction problems are incremental decision problems comprised of a set of variables, each of which is assigned a value from a non empty domain (Russel [30]). The assignment of each variable is constrained by a set of rules involving other variables in the constraint satisfaction problem.

A formal definition is taken from Russel and Norvig (Russel [30]).

- Let V be a set of sequentially assigned variables $C_1, C_2, C_3, ..., C_n$.
- Each variable C_i in V has associated with it a domain D_i comprised of possible problem specific values that may be assigned to C_i.
- The state of a problem is defined as a set of assignments to a subset of V.
  $C_1 = v_1, C_2 = v_2, ..., C_j = v_j$, where $j < n$ and $v_1 \in D_1, v_2 \in D_2, ..., v_j \in D_j$.
- The problem is in a legal state if none of the assignments violate a constraint.
- The problem is solved if it is in a legal state and satisfies some goal function G specific to the problem.

The parameters that constitute a constraint for each variable C_i and the parameters that dictate success or failure within the goal function G are specific to the problem being solved by the constraint satisfaction algorithm. Given the nature of solving constraint satisfaction problems search trees with back tracking are often employed to look for a legal state that satisfies a goal function (Russel [30]). Search trees, in particular depth first search trees, will be discussed in a subsequent section (section 2.4).

Generally speaking, the set of values within each domain D_i can come from an arbitrary data source. In the simplest case, each domain is filled with randomly selected data from some finite or countable finite set. Other systems may use data computed from existing known solutions to generate the values for D_i. Such data driven systems are quite common in music composition systems (Cope [12, 11, 13, 9, 10, 8], Miranda [23], Roads [28], Verbeurgt [21]. Most such data driven systems utilize Markov tables to order events at D_i. Markov tables hold such a dominating role in composing music with computers (including our own system) that they will be discussed in their own section (see section 3.3).

2.1.1 Constraint Satisfaction in Music Composition

The use of rule systems in music composition software dates to the very first efforts of composers and computer scientists exploring music composition as a problem of artificial intelligence. These systems use various mechanisms in the construction of the variable domains $D_1, D_2, ..., D_n$, most frequently employing probability functions to event generation or precomputed databases of events computed from human composed music (our system uses the later method, which will be discussed in section 3.3.1).

Hiller applied rule systems to random stream of random numbers to compose his Illiac Suite for String Quartet (Cope [8], Hiller [19]). Iannix Xenakis used rule systems applied to Markov tables, stochastic modeling and probability functions to compose many works in the 1950's (Cope [8], Xenakis [32]). Brooks, Hopkins, Neumann and Wright created a rule based system that composes hymns by selecting random integers, comparing them to pre computed music event probabilities and recursively rewriting integers that violate probability constraints ([19]).

2.1.2 Applicability to our goals

Composition systems that rely exclusively on programmed logic to constrain output to a user defined feature set suffer from a lack of agility when changes to the desired features are made. Consider what would need occur in a programmatic system that composes Bach if the user suddenly desired Webern. New rule systems must be programmed when new output styles are desired and working with source code is far beyond the means of most musicians. Professional programmers would quickly look for more agile solutions if given a composition system that required extensive source code extensions every time a change in output is needed. The existence of modern programming languages that easily facilitate the creation of modules and dynamic code execution merely relieves some of the burden on the programmer, it does not negate the problem of working with source code.

On a philosophical level too pure constraint satisfaction systems fall short of the goal of creating aesthetically pleasing music. It is beyond the scope of this text to define the set of features present within what is widely agreed upon to be “good” music but we feel confidant that no definition would be limited to a set of constraints on the musical events present within said “good” music. Clearly other parameters are present within the set of goals, motivations and methods of any entity creating aesthetic music; be that entity machine or biology. For this reason, systems that rely purely on programmed rule sets have become rare in computer music, with data driven constraint satisfaction becoming more prevalent.

Despite the limitations of constraint satisfaction, it still plays a heavy role in music composition systems. David Cope uses constraint satisfaction with data driven systems in a system that composes music in the specific style of a given composer (Cope [12, 11, 13, 9, 10, 8]). Our use of constraint satisfaction derives from our goal of creating sight reading études. Users wish to generate music with one or more of the following constraints: a certain key, a given melodic range, a specified set of rhythm events, a specified set of pitch events or a specified set of pitch intervals. We use programmed rules to ensure that the set of generated music events conforms to the user’s expectations by backtracking when an event selected by our data driven system violates a constraint.

2.2 Local Search Algorithms

Local search algorithms offer an algorithmic alternative to tree search algorithms and can be applied to a wide range of problems in artificial intelligence including constraint satisfaction problems. Unlike problems for which tree searches are applicable, the path of the incremental decisions made is not important information in the solution to a local search problem. All local search algorithms have the advantage of requiring far less memory than tree searches because they only keep k states in memory at one time. K is controlled by the user and is thus likely to require far less memory than a tree search (for the complexity of tree searches, see section 2.4). However, local search algorithms suffer from an inability to guarantee a solution will be found. Bounds must be included by the programmer to ensure algorithm termination (Russel [30]).

As an example of the type of problem for which local search algorithms are applicable, consider the N-Queens problem. In the N-Queens problem an algorithm is faced with the task of arranging N queen chess pieces on an NxN chess board such that no two queens are attacking one another. A solution to this problem consist only of the final state, the steps involved with arranging the queens are not important and thus need not be kept in memory as the algorithm runs (as would be the case if a tree search were used).

Russel and Norvig define local search algorithms as follows (Russel [30]):

- Let c be the current state of the problem. The initial state of the problem is taken as a parameter to the algorithm and its construction is specific to the problem being solved.
- Let $S = s_1, s_2, ...s_i$ be the set of successor states of c.
- Let $successors(c)$ be a function applied to c that supplies $s_1, s_2, ...s_i$.
- Let $fitness(s)$ be a fitness function applied to each s_i.
- Let $F = f_1, f_2, ...f_i$ be the set of values given by $fitness(s_1), fitness(s_2), ...fitness(s_i)$.
- Let $choice(x)$ be a function that selects $f_j \in F$, where f_j is a maximal or minimal fitness value as selected be the choice function.
- Let f be $fitness(c)$.
- If f has more fitness according to $choice(x)$, then the algorithm terminates with c returned as the algorithm's result. Otherwise the algorithm continues with c_j selected as c for the next round of selection.

The current state c of the problem is one potential solution to the problem instance. In the case of the N queens problem the current state c is any arrangement of N queens on an N by N chess board. Successor functions in local search algorithms in general need not adhere to any specific set of criteria, but they should not move the state too far from the current state or a potential state with maximal utility could be missed. In the case of N queens a successor function $successors$ would move one queen one space on the chess board, or it may swap row or column coordinates of two queens on the chess board.

The fitness function $fitness$ evaluates the extent to which a state solves a problem instance. A simple fitness function for the N-queens problem might count the number of queens that are under attack on the chess board. The choice function $choice$ selects a state that has the minimum number of queens under attack.

The simplicity and low memory consumption of local search algorithms comes with several significant drawbacks. There is no mechanism within the algorithm to prevent the search from revisiting states it has already visited. Keeping track of states already visited would require keeping states in memory and dynamically looking them up when the successor function is supplied. Additionally, the algorithm can return sub optimal results. Consider a two dimensional graph with states enumerated on the x axis and state fitness supplied by the fitness function graphed on the y axis. Graphs that contain not just global maxima and minima but also local maxima and minima may have such local maxima and minima returned as results because there is no successor state with better fitness that the choice function can see.

Solutions to problems in local search algorithms come in the form of modification to the basic algorithm discussed previously. For a thorough discussion see Russel and Norvig (Russel [30]). We will limit our discussion of modified local search state algorithms to that of genetic algorithms because genetic algorithms in particular have received much research and press in the computer music community (Sheikholharam [26], Cope [13], Hall [18], Miranda [23], Rowe [29], Biles [5] [6]).

Genetic algorithms modify the basic local search algorithm by adding a recombinate step to the successors of the current state. Successor states are sorted by their fitness and the successors of highest fitness are then spliced and recombined (for a visual example see figure 1 through figure 4). The resulting states of the recombination of successors then have their fitness evaluated and are considered by the successor function as potential successor states (in Hiller's ([19]) definition of genetic algorithms, the original set of successors S is discarded).

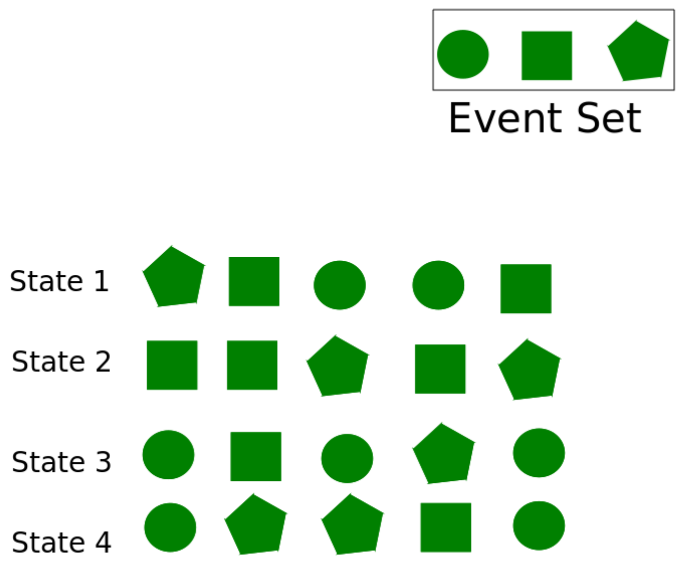

Figure 1. A state in the course of the running of a local search algorithm. States, one per line, are comprised of events contained within the set displayed in the upper right corner. States are order by the utility function (left up to the reader's imagination) with the most fit state located on top.

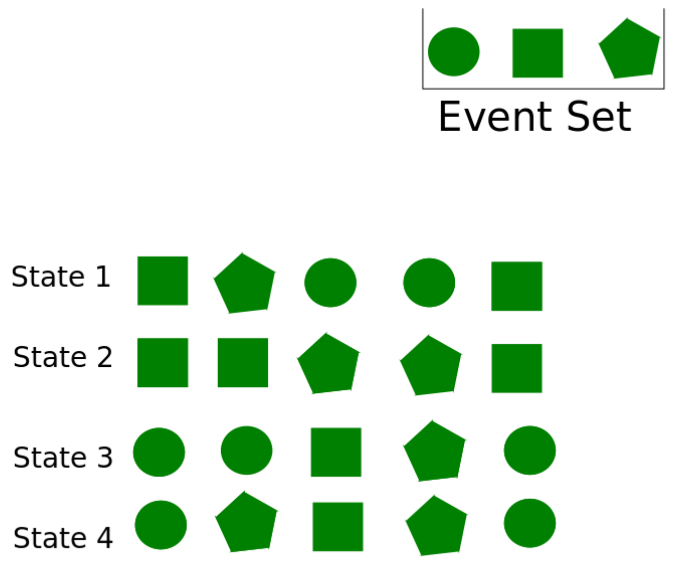

Figure 2. The random mutation step of a local search algorithm. Random events of each state are swapped or mutated completely to other events.

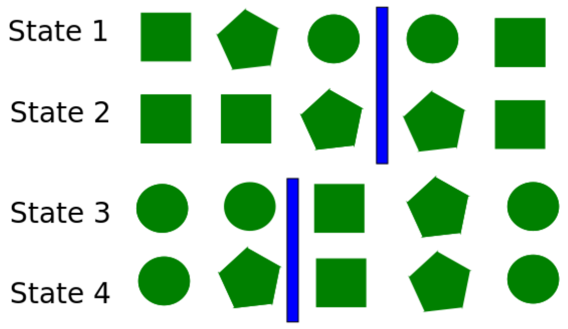

Figure 3. This figure shows the first recombinate step of a genetic algorithm. Recombination between state pairs is taken along randomly selected split indexes, shown as lines splitting each state pair.

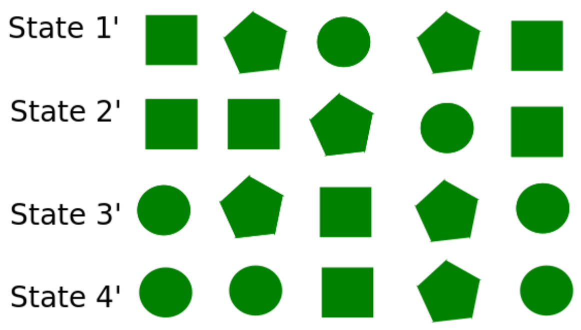

Figure 4. The second phase of a genetic algorithm's recombinant step. Each state within the state pairs are split along their indexes and recombined with their pair's respective sub state. Notice how in the case of the second state pair, no new state is generated after the recombinant step. The computations involved with performing the recombinant phase of this iteration in the local search algorithm have no effect on finding and optimal state. It is not just the recombinate phase of a genetic algorithm that can have no effect on finding an optimal solution, as the mutation phase of all local search algorithms has no guarantee of generating a new state.

The effect of recombining successor states is similar to that of evolution in biological life forms. Two states of high fitness may spawn offspring of higher fitness. The fitness of recombined states is however by no means guaranteed by the algorithm and thus genetic algorithms have mostly been applied to optimization problems (Russel [30]). Genetic algorithms work well with computer music composition because music is an inherently recombinate art form and both melodic structures and rhythmic structures lend themselves well to being adapted and recombined with other musical structures.

2.2.1 Applicability to our goals

Local search algorithms pose two problems for generating musical rhythm sequences in highly constrained environments. The first involves the initial state that must be supplied as a parameter into the algorithm. The initial state itself must violate no constraints and creating an initial state with no constraints efficiently requires the use of a search tree (which follows from the definition that the initial state have no constraints). This in itself would not prevent us from employing a local search algorithm but it does show that composing music is a problem that requires some degree of algorithmic design beyond that of the optimization problems towards which local search algorithms are frequently applied.

The second, and by far the most important problem, stems from the fitness function. If the initial state of the local search violates no constraint then the local search algorithm is charged only with the task of mutating the initial state into something that is aesthetically pleasing. To our knowledge, a purely algorithmic measurement of musical quality does not exist and the development of one that encompasses all musical style and listener taste will likely never happen. Computing mathematical properties of each state, such as event frequency, Markov tables (see section 3.3.1), or syncopation (see section 3.6) and comparing the results to precomputed measurements of human composed music is possible and easy to implement, but the relevance of one state in the search process against hundreds of pre-analyzed music scores is very small. We could also use direct human feedback as the fitness function, asking the user to evaluate and rank each state in the search process. The problem with this approach should be obvious. We would be requiring the user to evaluate potentially thousands of states and we would thus no longer be using computers to compose quality music but rather using computers to overwhelm the user with an obscene volume of data to manually process. This violates not only our goals but the goals of all computer science in general.

The third problem with using local search algorithms is their optimization nature. Because we are looking for algorithms that generate different output on each run, an algorithm that always finds an local maxima according to a fitness function will severely limit the diversity of output produced by the algorithm. It would be necessary to modify the local search algorithm to find a solution within some defined margin. Exploring the modification of local search algorithms may be an avenue for future research in our context of algorithmically composing music.

2.3 Neural Nets

The use of neural networks in artificial intelligence, particularly machine learning, is an area of artificial intelligence research that tries to mimic the inner workings of the biological brain on the level of the neuron (Russell [30]). Large networks of relatively simple structures, called neurons, are wired together and in theory can perform computations in the same fashion as neurons present in animal brains. Like genetic algorithms, neural nets have excited the imaginations of artificial researchers because of their apparent resemblance to biological systems at play in the natural world. But we shall see that neural nets are not so simple to implement as are local state algorithms (of which genetic algorithms are one).

The topic of neural nets is vast and complicated. We are only able to provide a high level description of the mechanisms at play and leave low level details as a topic of discovery for the reader. Understanding how neural nets work, even at the high level description provided here, starts with defining the cellular neuron. It is far beyond the scope of this paper to attempt a biological definition of a cellular neuron. We will therefore restrict our discussion to a generalization of neurons as they function in neural networks.

A neuron has the following components:

- A set of weighted inputs $I = i_1, i_2 ... i_n$. The weight type will be specific to the problem, but is typically some sort of numeric.
- A set of weighted outputs $O = o_1, o_2 ... o_m$. The individual neuron need not know what weights are attached to its output, nor is it necessary to know to what neuron each output is attached.
- An activation function f and a threshold value t that the neuron uses to dictate when it should fire a signal to each of its output links. When a neuron receives a signal on $i_j \in I$ it computes $v = f(\sum_{a=1}^{n} i_a)$. If $v \geq t$ then the neuron fires a signal on its output set O.

A neural net can therefore be thought of as a directed graph, where the neurons them-

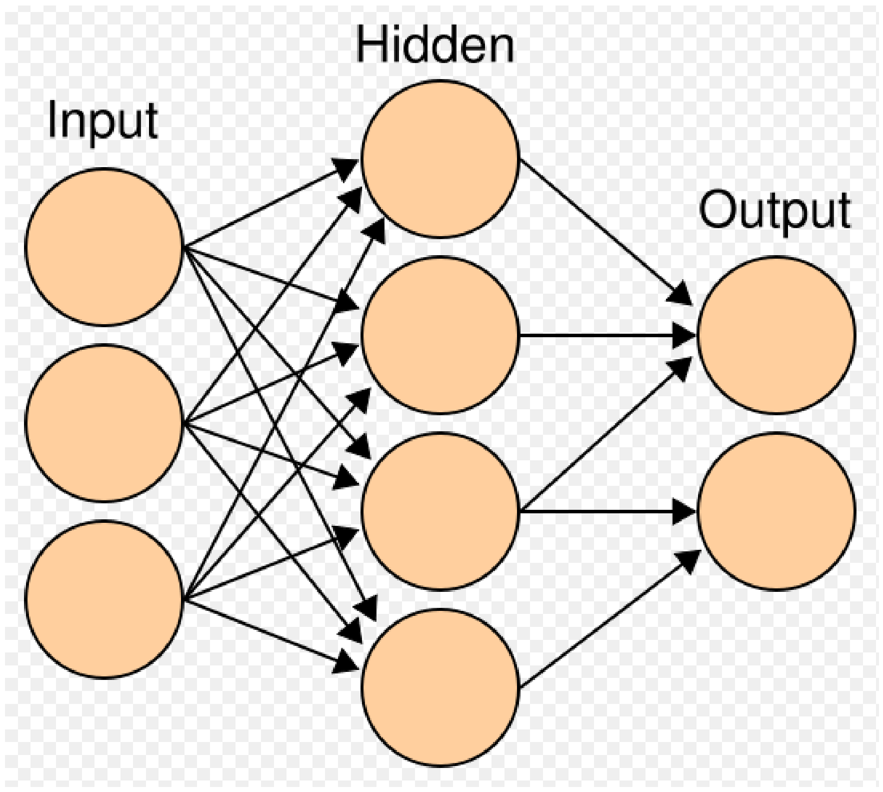

Figure 5. The complexity of neural nets prevents us from giving even a toy example of music composition, as we have neither a fitness function with which to train a neural net nor do we have a set of activation functions for the nodes of a neural net. This particular abstract visual representation of a neural net is an example of a feed forward neural net.

selves are the vertices and each neuron's O and I comprising the edges. The graph can either be acyclic, as in the case of feed forward neural nets, or cyclic as in the case of recurrent networks. The neurons themselves can be thought of as logic gates, that only propagate a signal if given the correct set of inputs (Russell [30]).

Of principle importance to implementations of neural nets is the discovery of the activation function that each neuron must implement to achieve a solution to any given problem. Because neural nets that have the same activation function for each neuron are of little use (though some nuerons in the neural net may implement the same function), the discovery of activation functions is a nontrivial problem. Researchers have discovered mathematical mechanisms to derive activation functions for neural nets and problem definitions that meet strict requirements (Russel [30]), but for many problems researchers fall back on using local search algorithms to "find" the activation functions rather than calculating them mathematically. The set of activation functions comprises the local state in the local search algorithm, with the fitness function applied to the output of the neural net (the input of the neural net is problem dependent). The algorithm then procedes until a suitable set of activation functions is found to maximize return from the fitness function. In the lingo of neural net research, the use of a local search algorithm to find a reasonable set of activation functions is called "training a neural net".

2.3.1 Neural Nets and Music

Neural nets have been exciting the algorithmic music community for quite some time, particularly in the area of recreating musical style ([12, 11, 13, 29, 23, 4, 15, 25, 26, 31, 20]). For the most part, music composition systems employ recurrent neural networks that take a small amount of random input, produce some amount of output and then feed that output back into the neural network until composition is complete ([25, 26]).

However, successful modeling of music style using neural networks has been done with limited success. The problem of neural net complexity and neural net training means that a non trivial amount of time must be spent in neural net preparation and leaves one neural net implementation unsuitable for deriving solutions to other problem definitions. David Cope, whom we have cited frequently and is perhaps the most knowledgeable figure in computer music composition research, said of neural nets in 2005 "...we should not overestimate the abilities of neural networks or let comtivity mask a lack of true creativity" (Cope [13]).

Despite Cope's dismissal of neural nets as a cure all algorithm for computer music composition, they still retain a presence within his composition systems. He uses Markov table mechanisms to compose rhythm and pitch but uses neural nets to control the larger picture musical constructs of phrase construction and structural form.

2.3.2 Applicability to Our Goals

The complexity of neural nets, both in terms of implementing them successfully in software and in training them to produce a desired output, precludes us from using them to compose music under highly constrained search environments. It is also the case that neural nets share many of the problems associated with local search algorithms when applied to our computational goals. Specifically, the problem lies with the implementation of an effective fitness function for training the neural nets. Additionally, were it the case that a fitness function could easily be computed, a different neural net would need be trained for each style of music desired by the user and each set of legal rhythm events provided by the user. Training different neural net each time a different style is desired or each time a new rhythm event is added to the legal event set would not deter a researcher in computer music composition but it does conflict with our goal that a musician without computer expertise find our system useful.

2.4 Depth First Searching

Having discussed common artificial intelligence algorithms frequently applied towards the goal of algorithmically composing music, specifically why they are unsuitable to our problem statement, we will in this section give an informal and high level view of how depth first trees behave. Search trees (both breadth first and depth first) are a common method of solving problems in artificial intelligence. Backtrack and its need in depth first searches will also be discussed in this section.

2.4.1 Algorithm

Depth first search trees consist of a collection of nodes, each of which storing arbitrary problem domain specific information, and recursively defined in one of the following ways:

- A leaf in the tree, with one parent node and no children nodes.
- The root of a depth first tree, with a set of child nodes and one parent node.

- The root of a depth first tree, with no parent node. This root node, sometimes referred to as the head of the tree or the parent of the tree, often contains no information an functions only as the root of the tree.

Part or all of the information stored at each node is an element present within some finite set S. The search proceeds as follows (see figure 6 for a visual example).

1. At each step i of the search process there exists a set of child nodes s, one for each $e \in S$. A node c is selected from s (the selection procedure is problem specific and will not be discussed here). If $i = 0$, then the root node is selected as c_i.
2. Let p be the path from c_i to the root of the tree, excluding the root. That is to say that p is $c_1, c_2...c_i$.
3. A termination function is applied to p. The termination function is problem domain specific and signals when the search should stop.
4. (a) If the termination function signals the search should stop, then the depth first search algorithm terminates. p is the problem's solution.
   (b) Else the algorithm generates a new set of child nodes s and adds them to c's set of child nodes. The algorithm then continues to step $i + 1$.

2.4.2 Backtracking

Backtracking is required if the depth first search algorithm is to solve constraint satisfaction problems. Backtracking allows alternate paths in the tree to be explored in the case a constraint is detected. Backtracking is not required in breadth first searches because all paths of the tree are explored simultaneously. While the lack of backtracking in breadth first trees may render them attractive at first glance, their algorithmic complexity (discussed shortly) renders them useless for all but the smallest of searches.

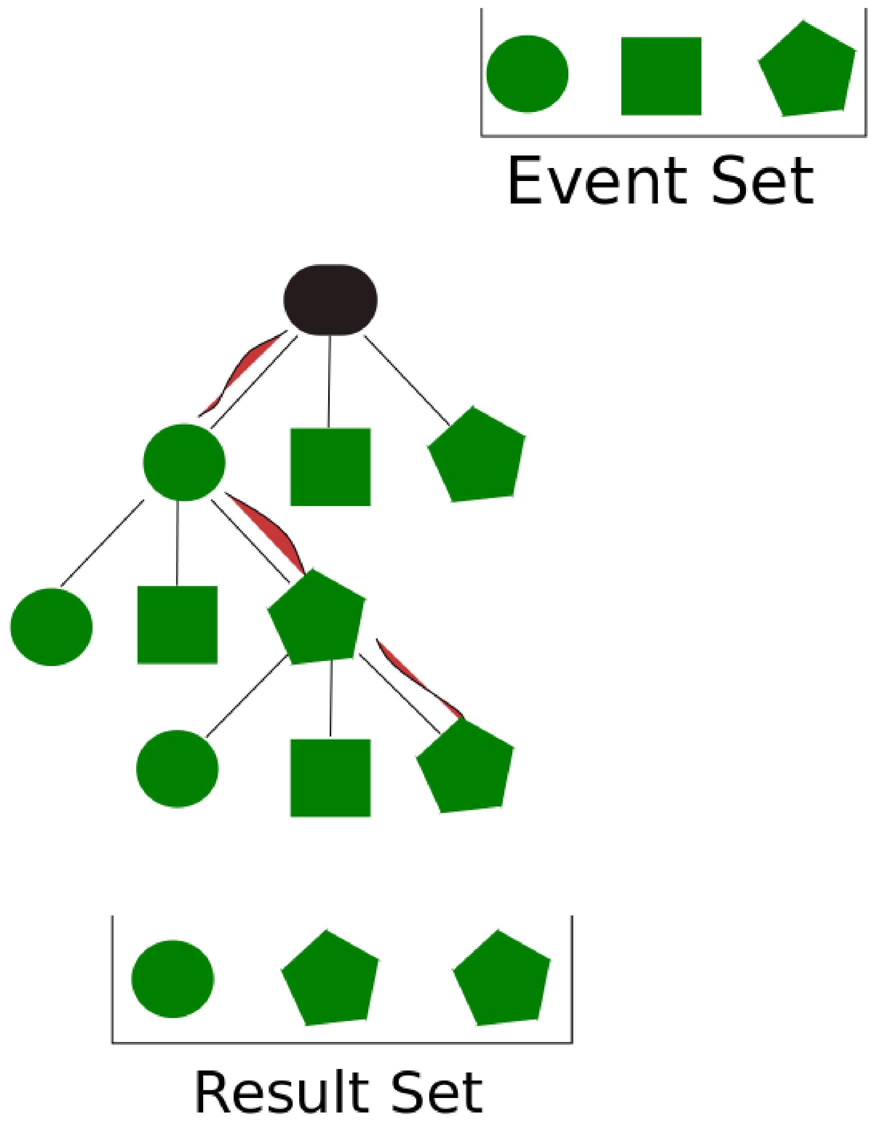

Figure 6. Full depth first search tree. The first node (with no parent) represents the root node with each level of the tree made up of events in the event set listed top right. The curved line shows the path through the search tree to a leaf. This path is the sequence of events selected by the search, and is also represented below the tree as three unconnected shapes.

Depth first search trees do suffer from the problem of undecidability nin the case that the problem domain does not have a limit in the depth of the tree. The current path of

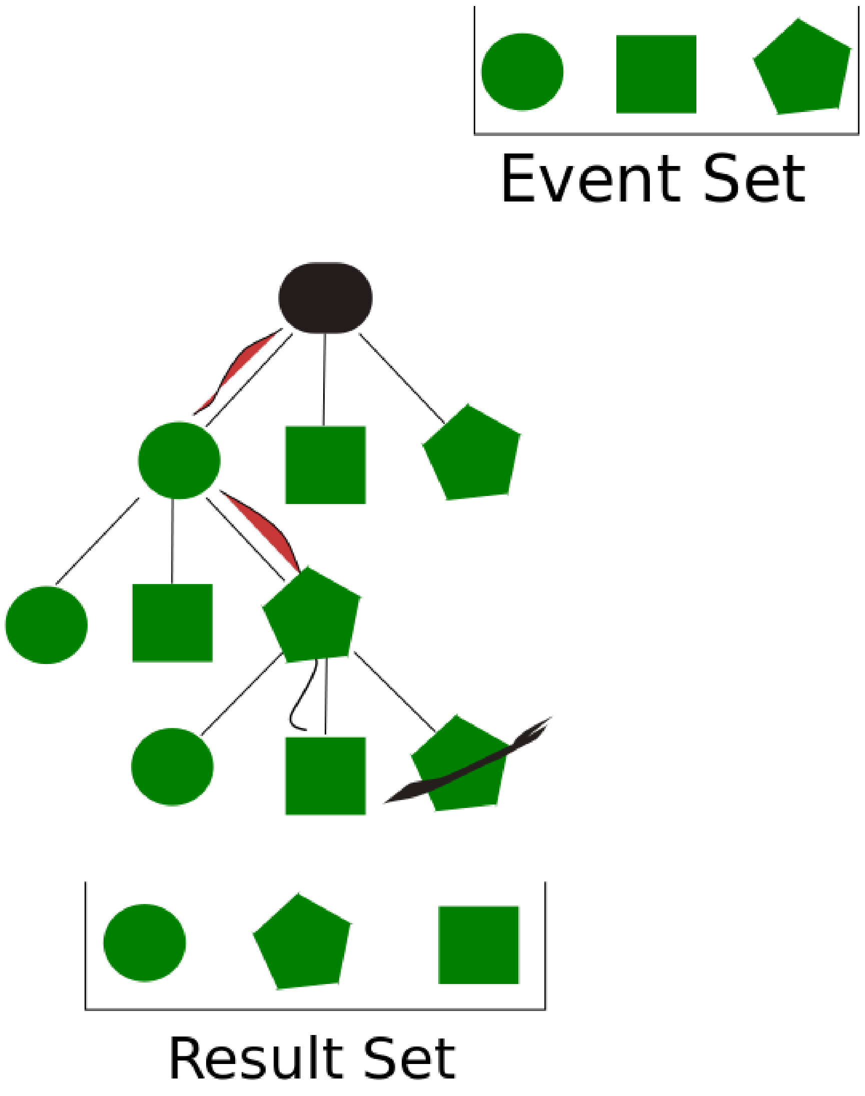

Figure 7. The depth first search tree after one backtrack. The previously selected leaf has been trimmed from the tree and there is a new path from root to leaf. The new sequence of nodes as chosen by the algorithm is displayed below the tree as unconnected shapes.

exploration may provide no solution and also violate no constrains, resulting in an infinite plunge down one path of the tree. Our problem domain does not have this problem for reasons that will be explained in section 4.

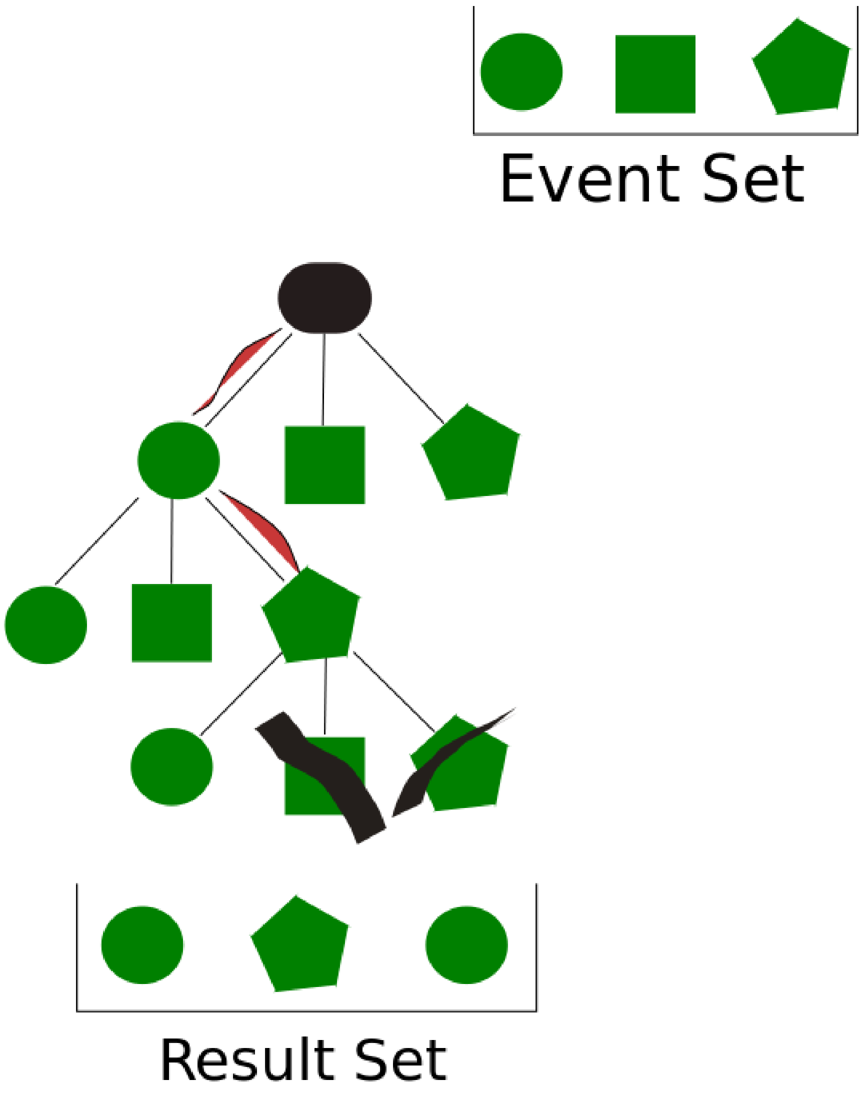

Figure 8. The depth first search tree after two back tracks. The new sequence of nodes as chosen by the algorithm is displayed below the tree as unconnected shapes.

2.4.3 Performance

Depth first searches have an algorithmic performance of $O(bm)$, where b is the branching factor at each node and m is the maximal depth explored in the tree. If the algorithm first search: $O(b^m)$.

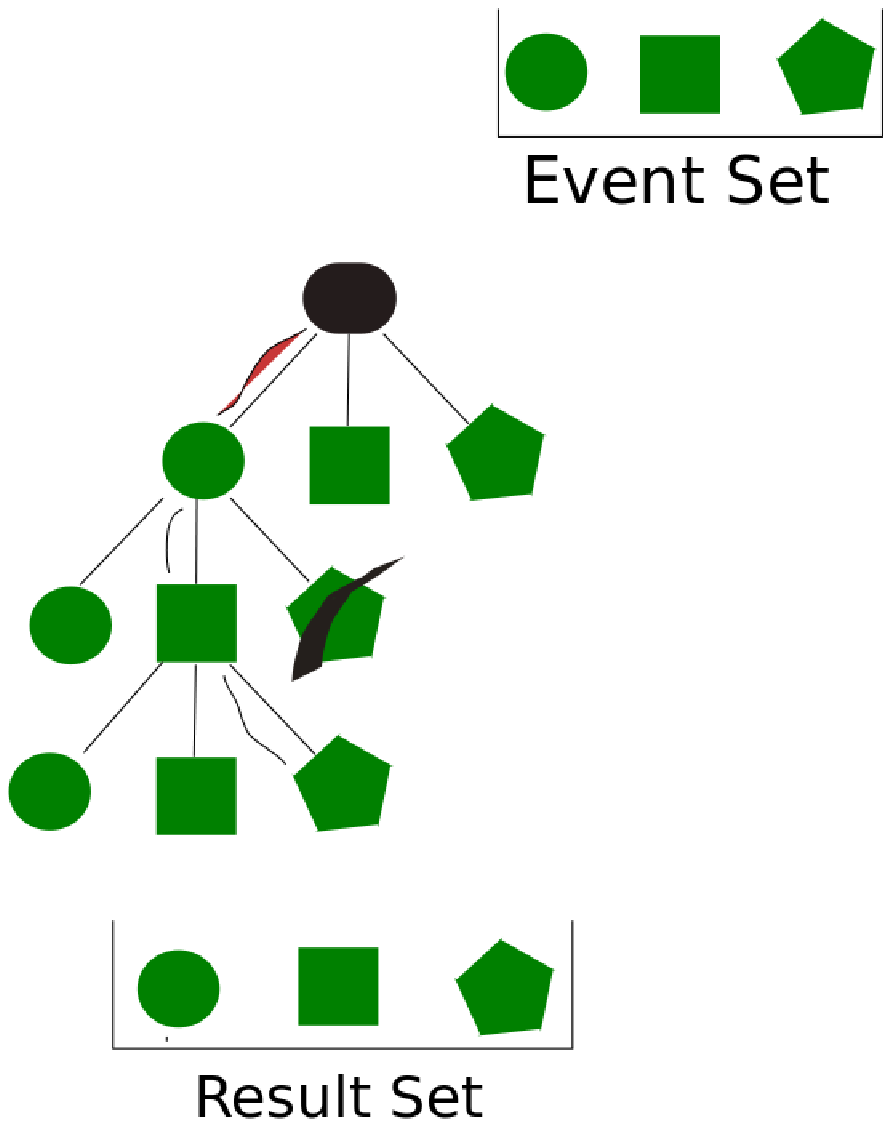

Figure 9. The depth first search tree after three back tracks. The algorithm has back-tracked a sufficient amount to deplete all generated nodes on one level of the tree and must therefore delete a node from the previous level and generate a new generation of candidate events. The new sequence of nodes as chosen by the algorithm is displayed below the tree as unconnected shapes.

backtracks, as ours do, then the worst case performance is equivalent to that of a breadth first search: $O(b^m)$.

3 Generating Rhythm

This section describes four algorithms for generating rhythms within a constrained search space, Simple Search, Concrete Search, Relative Search and Syncopation Search. Distributions were taken over output of the algorithms to measure the extent to which each algorithm is able to successfully generate specified event sets.

3.1 Commonality

There are common features in all four algorithms that can be discussed at a high level without delving into very much detail. Like virtually all constraint satisfaction problems a search tree is used to incrementally generate the candidates with a backtrack occurring in the case of a constraint violation.

Some terminology will be useful:

- An event is a single musical rhythm. For example, an eighth note or quarter note. Events are notated as rational numbers, with a whole note selected as 1. A quarter note is $\frac{1}{4}$, an eighth note $\frac{1}{8}$, a dotted quarter $\frac{3}{8}$, etc.

- Let L be the set of legal rhythm events specified by the user. That is, what rhythms the user desires be present in the output of the algorithm. For instance, a user might specify an eighth note triplet ($\frac{1}{12}$) and a quarter note ($\frac{1}{4}$) if they are interested in practicing eighth note triplets.

- Let G be the sum of the rhythms to be generated by the algorithm.

- Let Q_i be the priority queue of events representing the child nodes at step i of the search. The measurement of priority differs from algorithm to algorithm and is equivalent

- Let $C(Q_i)$ be the head of Q_i. This represents the event of maximal "utility" and is the event for which child nodes will be generated at Q_{i+1}. The other events in Q_i will be leaves unless a backtrack removes $C(Q_i)$.

- Let T be the search tree of i queues.

- Let $P(T)$ be $\{C(Q_1), C(Q_2)...C(Q_i)\}$, or the path from the root of the search tree to the bottom most leaf $C(Q_i)$.

At each step i the algorithm generates a set of candidate events, chooses the best candidate event and moves on to step i + 1 of the search. If the set of candidate child nodes at i is , then the algorithm removes $C(Q_{i-1})$ and tries again to generate Q_i. The algorithm terminates successfully if the sum of event durations within $P(T)$ equals the goal duration for the search process. It terminates unsuccessfully if the algorithm backtracks to an empty search tree.

The individual algorithms deviate from one another when populating priority queues of child nodes and in their implementations of choosing $C(Q_i)$.

3.2 Simple Search

The first algorithm is the simplest algorithm one could possibly use when searching in a constrained environment and is presented as a basis for comparison. At each step i in the search process Simple Search computes the remaining duration $R = G - sum(P(T))$. It then removes from L all elements j such that $j > R$ and places them in a queue to be placed in the search tree. The choice of $C(Q_i)$ is done randomly.

Despite the algorithm's simplicity and lack of attempt to model human composed music, its tendency to produce rhythms with near even event distribution could be very useful to a musician involved with pedagogy. This algorithm could easily be extended to produce specific events at specific rates specified by the user, but such an extension is not implemented here.

3.3 Markov Chains

The next two algorithms both use Markov chains when choosing $C(Q_i)$. Markov chains are an incremental decision making process that use decisions made at $n-k, n-k+1...n-1$ to make a decision at step n([23]). The internal mechanisms of Markov Chains are revealed in the explanations of the following algorithms, and a visual example is provided in figure (10).

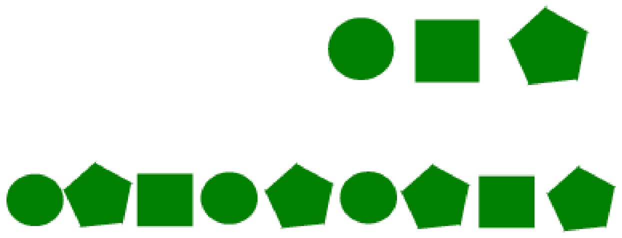

Figure 10. An example sequence of events using the same event set as the depth first search tree visual aids. The table below lists the event set as column names and the order 2 event strings as row names. The entries in the table represent the probability that a given column's value will follow a given row's event sequence.

Markov Table Example			
Event String	C	S	P
C,P	$\frac{1}{3}$	$\frac{2}{3}$	0
P,S	$\frac{1}{2}$	0	$\frac{1}{2}$
S,C	$\frac{1}{1}$	0	0
P,C	$\frac{1}{1}$	0	0

We use as a data source a subset of the Bach chorales (Site [1]). The decision to use the Back chorales is a result of availability and our desire to show that a modified Markov algorithm can yield results that are musically interesting without being limited to the set of events present within the data set. The basic problem we are trying to overcome is that a user can specify a set of legal events for use in composition that are not present in the data set used to build Markov tables. The rigors of pedagogy are such that this problem will manifest itself very frequently. One possible solution to the problem would be to find data sets that contain the same set of events as desired in output études. As previously discussed, this is likely too burdensome to the user as it would require large amounts of research and experimentation with analysis programs on the part of the user. While not an insurmountable task, the use of the modified Markov algorithm aims to give acceptable musical output across a wide scope of user constraints without demanding work on the part of the user.

3.3.1 Construction

It is worth briefly mentioning how we mined the Markov tables themselves. The Bach chorales data set we used is encoded in a format called musicxml. The musicxml (Cite [2]) is turned into a more easily parsed format by other programs via an arbitrary python program. The output is then fed into another set of python programs that maintain a set of hash tables of integers, keyed by the chains of events corresponding to the depth of the Markov chain. That is to say, there is a table for chain depth $1, 2..n$.

For each chain depth d the program steps through each rhythm event e_i constructing a hash table key k out of events $e_{i-n}, e_{i-n+1}...e_i$ (the key construction is arbitrary). The integer value within the hash table for chain depth d at key k is then incremented.

After stepping through all pieces of music within a data set and constructing the n hash tables the program writes the tables to a flat text file that can be loaded and parsed by the Java composition programs.

3.4 Concrete Markov Search

The étude generator implements a traditional Markov chain with search backtracking both as an algorithm against which to test the relational Markov table algorithm and because traditional Markov searches could still hold value to a pedagog.

Concrete Markov search, hereafter Concrete Search, differs from Simple Search in the way it selects $C(Q_i)$. A random choice is still made but the events in Q_i have probability weights attached to them via the Markov tables that are precomputed from the Bach chorales. However, the process differs slightly from a straightforward Markov implementation because we have to prune events from the Markov table that are not members of the set of legal events supplied by the user. Additionally, backtracking may occur and invalidate choices made at previous iterations of the search process. Concrete Search is in some sense the combination of traditional constraint satisfaction searches and Markov table algorithms.

3.4.1 Concrete Markov Mining

Events mined from Bach chorales do not take into account how the rhythm events are represented in the musicxml. Rhythm events can be constructed in western music notation (and musicxml) as being comprised of an infinite number of sub events tied together to form one aggregate event duration. Our Markov table generation program treats events written as tied sub events as an event of the aggregate tied duration regardless of the set of sub rhythm events used to sum to its duration. See fig 11 for a visual example of how the Markov table building program handles tied rhythm events.

3.4.2 Search Process

Because the set of valid events specified by the user are virtually guaranteed to be a sub set of the set of events present within the Markov table, Concrete Search must normalize the event probabilities before making a choice. The procedure is straightforward and is also used in the relational Markov search algorithm (see figure 15 through 16).

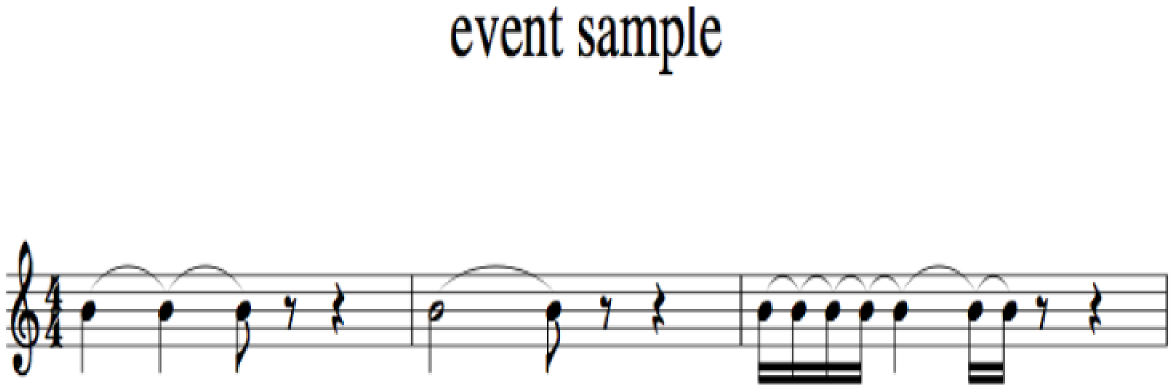

Figure 11. Examples of tied rhythm events that sum to the same event duration. The Markov table building programs would treat all three tied events as rhythm event $\frac{5}{8}$.

- Let M be the set of events and associated probabilities in a j order Markov table for previous events $C(Q_{i-j}), C(Q_{i-(j+1)})...C(Q_{i-1})$.
- Let $L` = L \cap M$.
- Let $S =$sum of event probabilities in $L`$. Note that because the sum of probabilities in $M == 1$, $S <= 1$ by virtue of it being a subset.
- For each $e \in L`$ assign $e.probability = e.probability/S$.
- Sort $L`$ on the probability of each $e \in L`$.
- For each $e_i \in L$ set $e_i.probability = e_i.probability + e_{i-1}.probability$.

After normalization of the table, a random number is generated between $(0, 1]$ which is used to select $C(Q_i)$. The normalized table reflects the probabilities of the complete Markov table but omits all events $e \notin L$. This normalization must take place for Q_i in the event of a backtrack as well.

3.5 Relational Markov Search

The Relational Markov Search, hereafter Relational Search, is largely the same algorithm as the Concrete Search with one significant difference. The Markov tables are built in such a way that values within the table are not the literal events within the mined data set. Rather, the tables store the multiplands between events. Formally:

- Let $M(C(Q_{i-1}), C(Q_i))$ be the multipland resulting from the operation $C(Q_i)/C(Q_{i-1})$.
- Let K be the concatenation of: $M(C(Q_{i-(order)}), C(Q_{i-(order-1)})), M(C(Q_{i-(order-1)}), C(Q_{i-(order-2)}))...M(C(Q_{i-2}), C(Q_{i-1}))$.
- K is used to look up the probabilities and multiplands by which $C(Q_{i-1})$ must be multiplied to get the events for Q_i.

See figure 12 for an example of how relationships between events are handled in the Markov Table building processes.

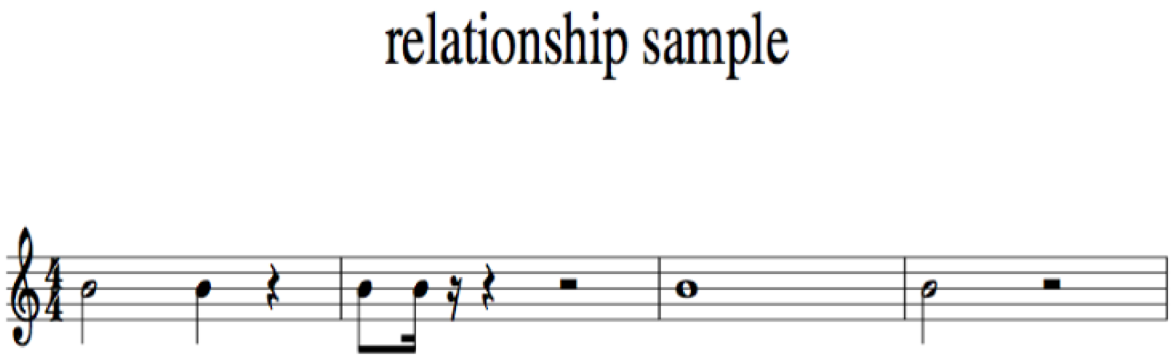

Figure 12. Examples of an equivalent rhythm event relationship between rhythm events of different duration. The program that builds Markov tables for the Relation Search algorithm would treat the relationship between rhythm events in the first bar, second bar and the events present in bars 3 and 4 as $\frac{1}{2}$.

3.6 Syncopation Search

Syncopation Search uses a syncopation function to measure the syncopation that has been accumulated at each step of the search process. Syncopation is defined as the momentary contradiction of the prevailing meter or pulse [14]. From its definition syncopation requires a steady pulse to contradict, else the resulting rhythms are likely cacophony. While cacophony is a valid compositional goal, it is undesirable in this context.

Gomez defines the syncopation function we use as a component of a rhythm event's duration and distance from metric beats [16]. This measurement is far from perfect but is sufficient to allow an ordering of events at stages in the search.

The Syncopation measurement is restated here for convenience.

- Let e_i and e_{i+1} be two strong beats (pulses) in a time signature.
- Let x be a rhythm event that happens on or after e_i but before e_{i+1}.
- Let d(x, e_j) be the distance between the start of x and the start of e_j. This distance is measured by musical rhythm.
- Let T(x) = min(d(x, e_i), d(x, e_{i+1})).
- Let D(x) be a function defined as:
  - if x starts on e_i then D(x) = 0.
  - else if x ends before or on e_{i+1} then D(x) = $\frac{2}{T(x)}$.
  - else D(x) = $\frac{1}{T(x)}$.
- Let S = $\sum_{\forall x}^{X} T(x)$, where X is a finite string of rhythms, usually a piece of music.

Syncopation Search generates rhythms in the following way:

- The duration G is divided into four chunks of equal duration.
- At each step i of the search a syncopation measurement S_{i-1} is taken over $P(T)$.

- For each $event_j \in Q_i$ S_{i-1} is used to compute the syncopation $S_{i-1,j}$ when $event_j\ is\ C(Q_i)$. The $S_{i-1,j}$'s are then used in the following heuristic:
  - If the sum of events $\in P(T) \leq \frac{G}{4}$ or if the sum of events $\in P(T) \geq \frac{3G}{4}$, or if $S_{i-1} \geq$ some user defined constant then the utility of $event_j$ is set to $K - S_{i-1,j}$, where K is a sufficiently large constant.
  - Else the utility of $event_j$ is set to $S_{i-1,j}$.
- The utility of each $event_j \in Q_i$ computed in the previous step is then used to build a probability table for a randomized selection.

Syncopation Search de-emphasizes syncopation during the first and last quarter of G. Syncopation is emphasized in the middle of G but only in the case that there is not too much syncopation in the rhythms already. The effect is a period of relative rhythmic calm that implies a metric pulse, followed by a period of syncopation that challenges the metric pulse, followed by a period of calm that reinforces the metric pulse.

Events are still chosen with a random choice but the probability for each event is weighted by the extent to which the event adds syncopation. Thus, Syncopation Search is quite suitable for generating varied output for sight reading études.

3.7 Experimental Results

Each of the three algorithms (and Simple Search) is tested for average distribution of events to determine the extent to which they are able to generate output consistent with the expectations of a user. Seven sets of rhythm events were used as L and fed into the composition program, each of which was comprised of rhythms that would hopefully manifest characteristic distributions in each of the algorithm's output and give insight as to the applicability of each algorithm within a given context (L).

For each set of rhythm events L 50 output sets of size 100 were generated and measured, with the average and standard deviation taken over the entire generated output. The goal was to demonstrate that a modified Markov process that uses relationships between events rather than events themselves provides a better distribution of events in the highly constrained environment of musical pedagogy.

We begin with a few general observations about the shape of distributions in all generated output data sets measured. The Simple Search algorithm shows approximately the same distribution for all data sets, but the distribution isn't exactly even across all events as expected. Instead the distribution is biased towards events with shorter durations. This phenomenon exists because as the allotted space G becomes full longer events are pruned from the available data set L. This means that shorter event durations are available for a larger subset of the algorithm's iterations.

Another broad trend within the data, and one obvious from the algorithm's pseudo code, is that Concrete Search will produce events with distribution probability 0 if the event is not present within the original data source. Or it will produce events with very low distribution in relation to other rhythms in the legal event set if the event had low probability in the data source. Concrete Search can still perform adequately if all the events have reasonably similar distributions in the original data set, but is limited in application when one event dominates all others.

The events and corresponding distributions can be found in figure 13. Relationship distributions between events can be found in figure 14. Noticeable but not surprising is the overwhelming prevalence of $\frac{1}{1}$ as a relationship between events in the Bach chorales. Despite this bias Relational Search can still generate a reasonable event distribution for a wider variety of input sets than Concrete Search.

3.7.1 Data Sets

The first data set examined experiments with rhythm events well represented in the Bach chorales 18, namely quarter notes and eighth notes. All three algorithms generate approximately the same distribution, and for such a simple data set the results shouldn't be too surprising. It is, however, worth noting the extent to which a constraint modification to the

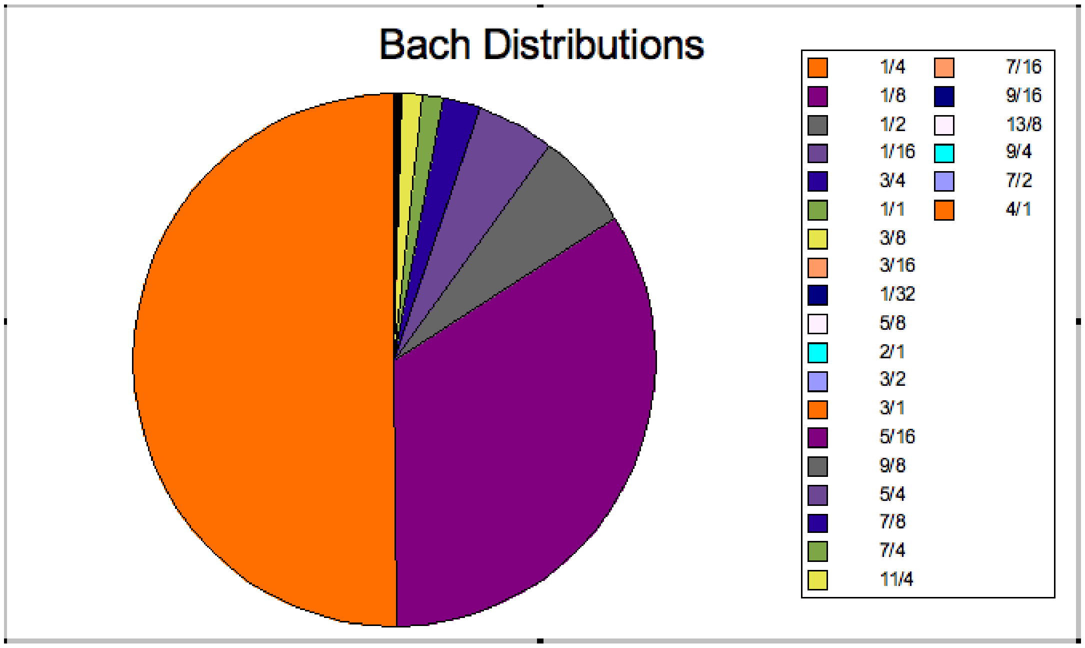

Figure 13. Distribution for events present within the Bach chorales.

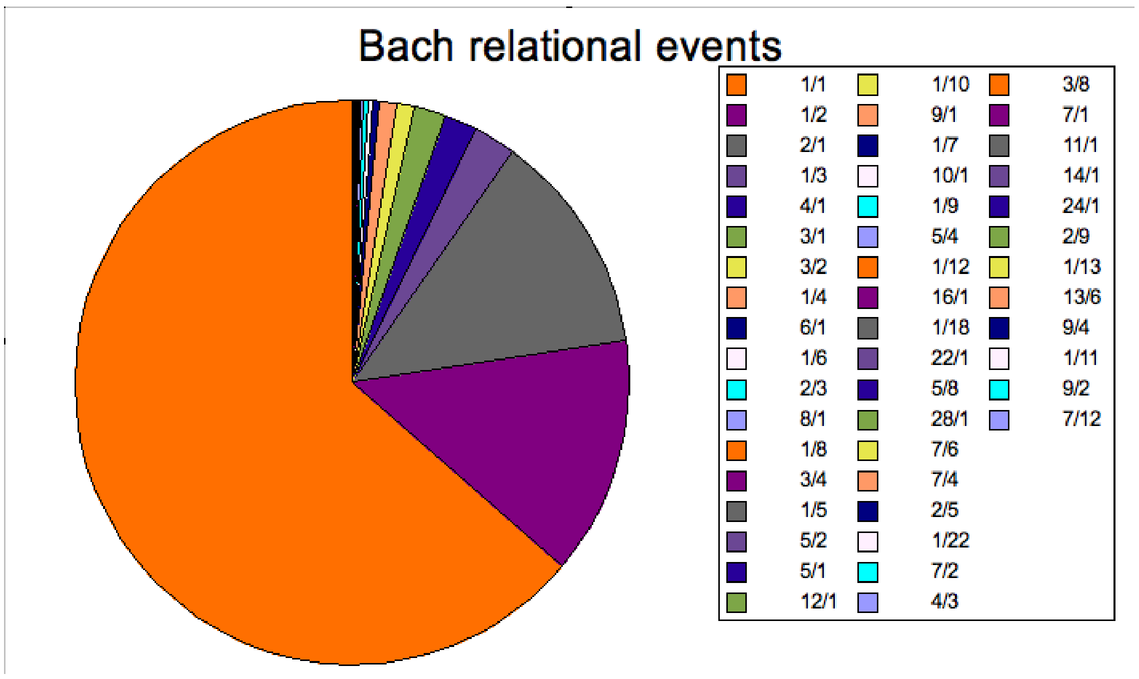

Figure 14. Distribution for relationships between events within the Bach chorals.

Markov process effect's Concrete Search's distribution. The expectation is that Concrete Search would closely mirror the distribution present in the Bach chorales 13, but here the distribution is almost evenly split. This is likely due to backtracking at the end of the algorithm as the space filled approaches the space allotted by the user.

Data sets two through four juxtapose events that are very uncommon with events that are very common within the Bach chorales 19 20 21. Concrete Search reflects the choice of events by producing very poor counts of the events that are not well represented in the Bach. Relative search provides the most even distribution, which is surprising as the expectation is that Simple Search would give the most even distribution across all data set inputs. Conversely, relative search does show a more even distribution of events.

Data sets five through seven have events within their event sets that are not present in the Bach chorales 22 23 24. Concrete Search does not produce such events and is thus unsuitable for the type of pedagogical work that requires their use.

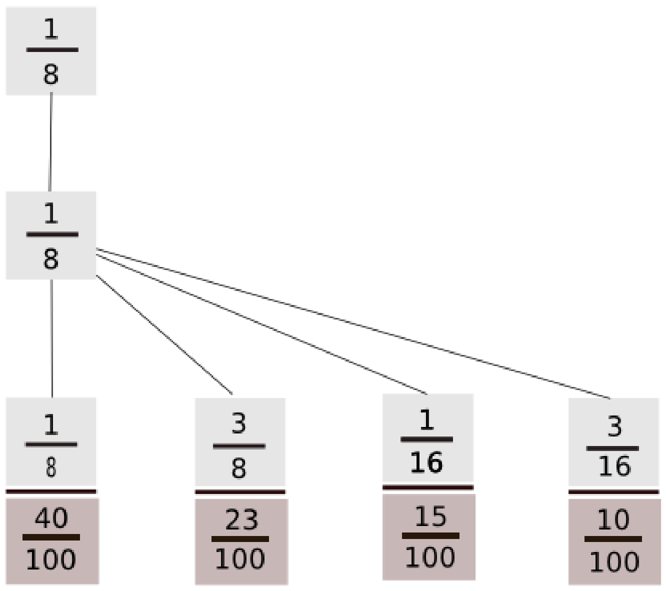

Figure 15. This graphic represents the a level in a depth first search tree after events have been given weights but before the weights have been normalized. Four events from L are present in the set of child nodes (the top half of each child node), each with a fractional weight shown as parts per 100 for convenience (the bottom half of a child node). Notice how the sum of the weights of the child nodes is not 1. This happens very frequently in our depth first search trees because the utilities provided by Markov tables are probabilities computed on a set of possible events greater than or equal to the set of events provided by the user (L). Also, after a backtrack the sum of child node utilities is guaranteed to be less than one because one node was removed from the set. Note that the values in the example are hypothetical and not pulled out of any Markov table we built.

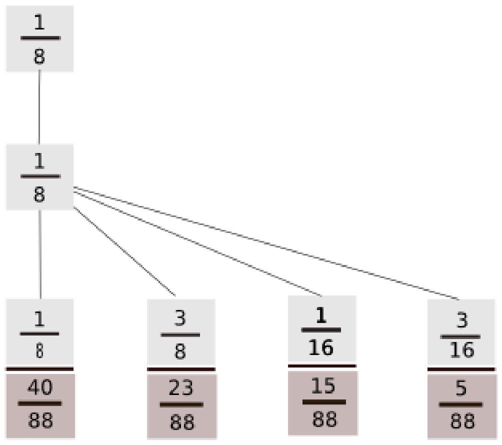

Figure 16. The sum s of the child node weights in the previous graphic is $\frac{88}{100}$. This figure shows the weights of the child nodes after they have been divided by s (shown as parts per 88 for convenience).

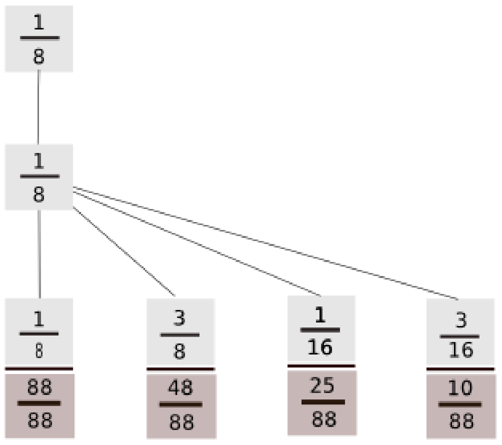

Figure 17. The utility at each child node n_i is set to $utility(n_i) + utility(n_{i-1})$, moving from right to left omitting the right most node. The child nodes are ready for selection now that normalization is complete. For instance, if the number $\frac{34}{88}$ is selected the rhythm event $\frac{3}{8}$ because the utility of that child node is the first node that has a utility greater than or equal to $\frac{34}{88}$.

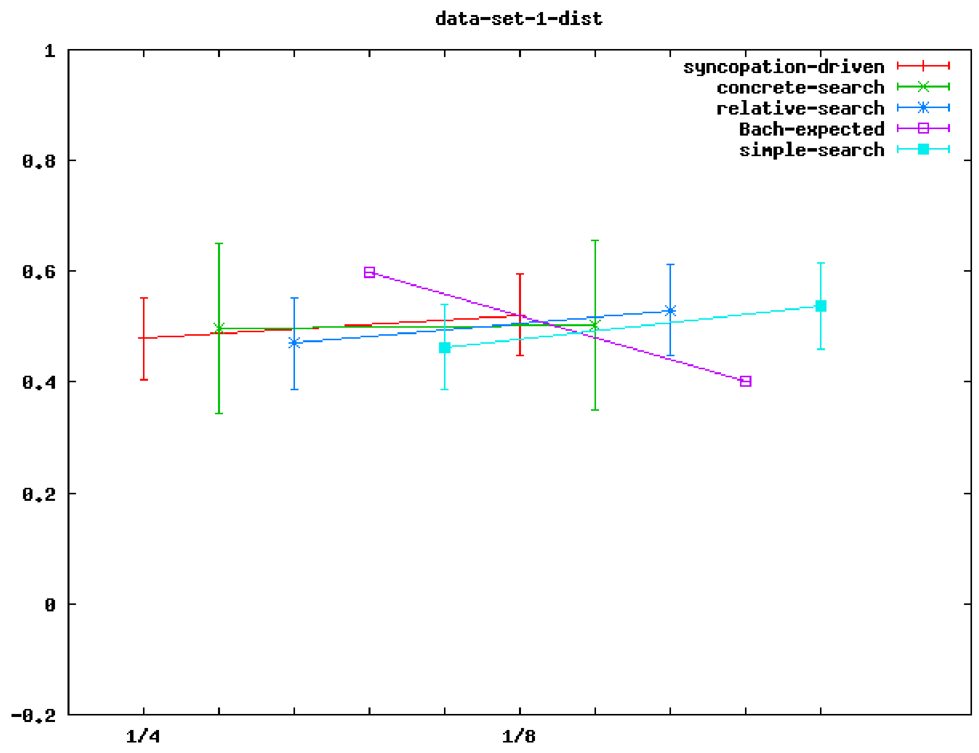

Figure 18. Distribution for the event set consisting of a quarter note and an eighth note. All four algorithms measured show a high degree of effectiveness at generating output with this data set, which is expected given the prevalence of the events included within the Bach data set. A some what surprising result is that Concrete Search does not match the curve given by the expected values within the Bach. Instead is generates approximately the same number of each event.

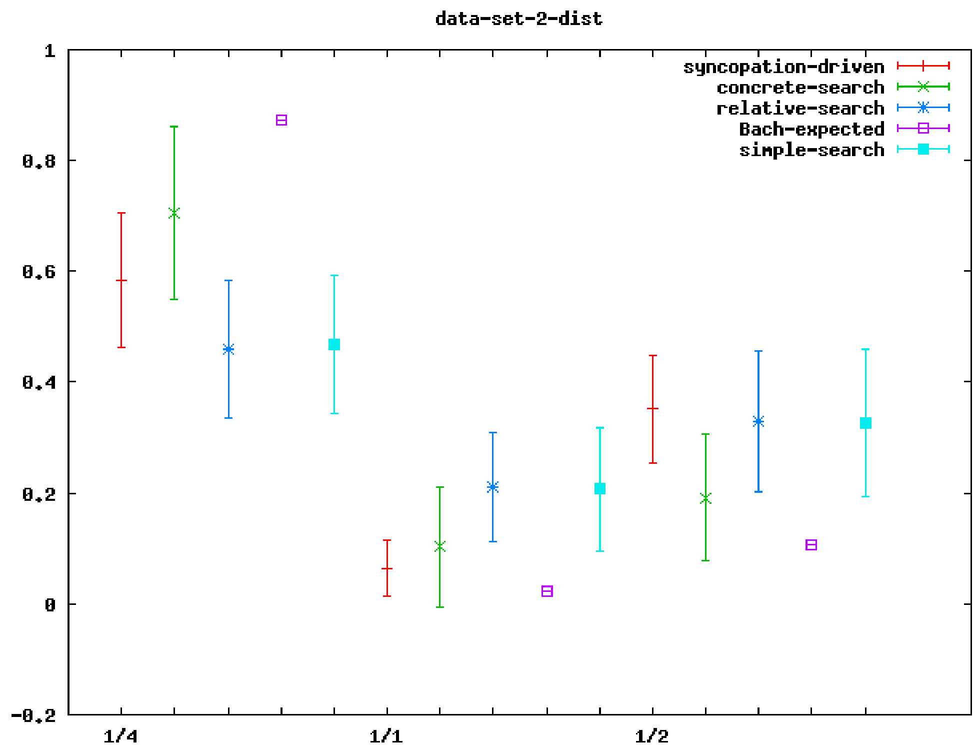

Figure 19. Distribution for the event set consisting of a quarter note, a half note and a whole note. The events in this set were chosen because the relation ships between the events are prevalent within the Bach data set, but within the Bach data set the quarter note is far more prevalent than either the half note or whole note. Concrete Search shows more preference for the quarter note event than the other algorithms; although all algorithms do to a certain extent.

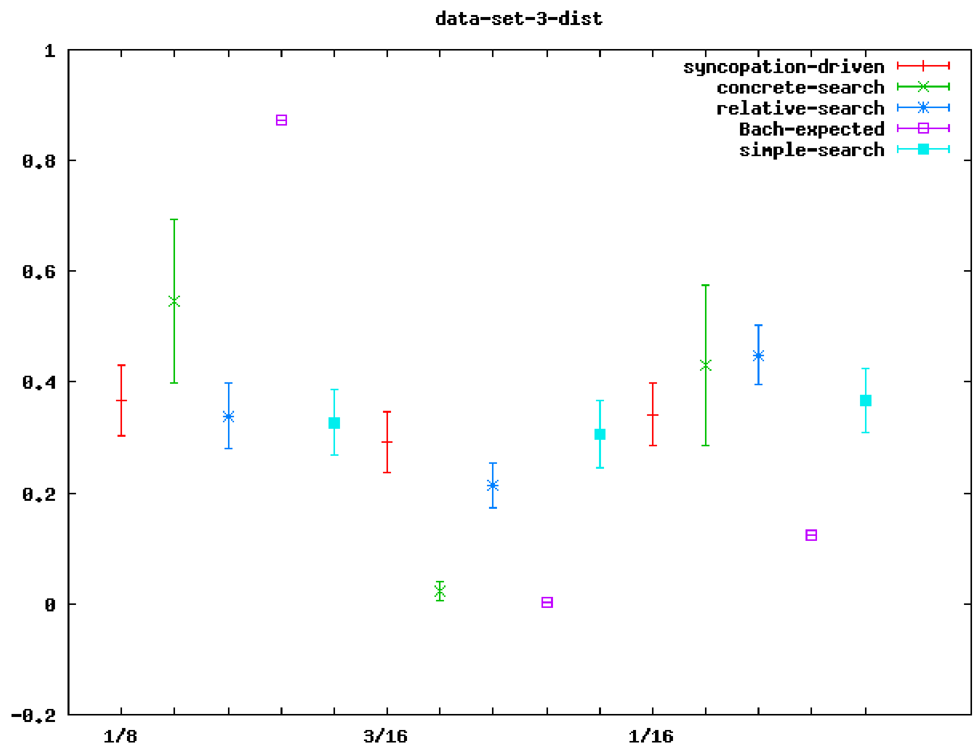

Figure 20. Distribution for the event set consisting of an eighth note, a dotted eighth note and a sixteenth note. The dotted sixteenth note in this data set represents a real problem for Concrete Search. Its near zero occurrences in the Bach data set means that Concrete Search is unable to produce sixteenth notes with much frequency (near zero).

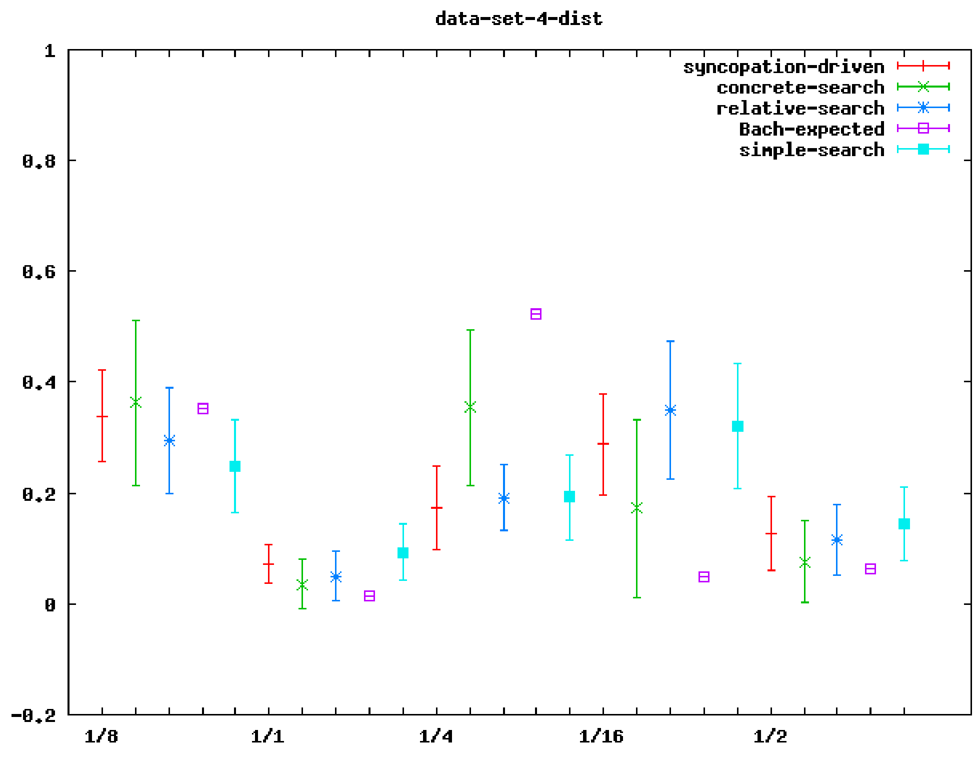

Figure 21. Distribution for the event set consisting of an eighth note, a quarter note, a whole note, a sixteenth note and a half note. This event set was chosen for the same reasons as data set 2, but with more events functioning as "noise" to the events well represented in the Bach data. Concrete Search mimics the curve of the Bach expected values while Syncopation Search and Relative Search deviate from expectations on the quarter note and sixteenth note.

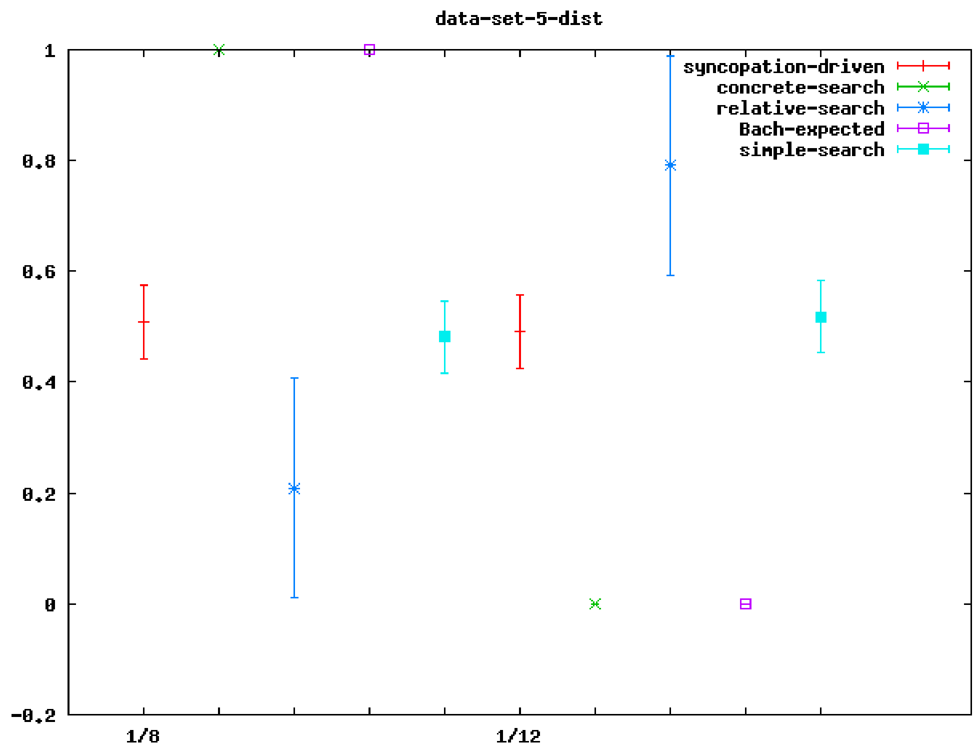

Figure 22. Distribution for the event set consisting of an eighth note and an eighth note triplet. The results show Concrete Search's inability to output events desired by the user with a very small input set consisting of one event well represented in the Bach data set and one event not represented. Concrete Search produces no eighth note triplets.

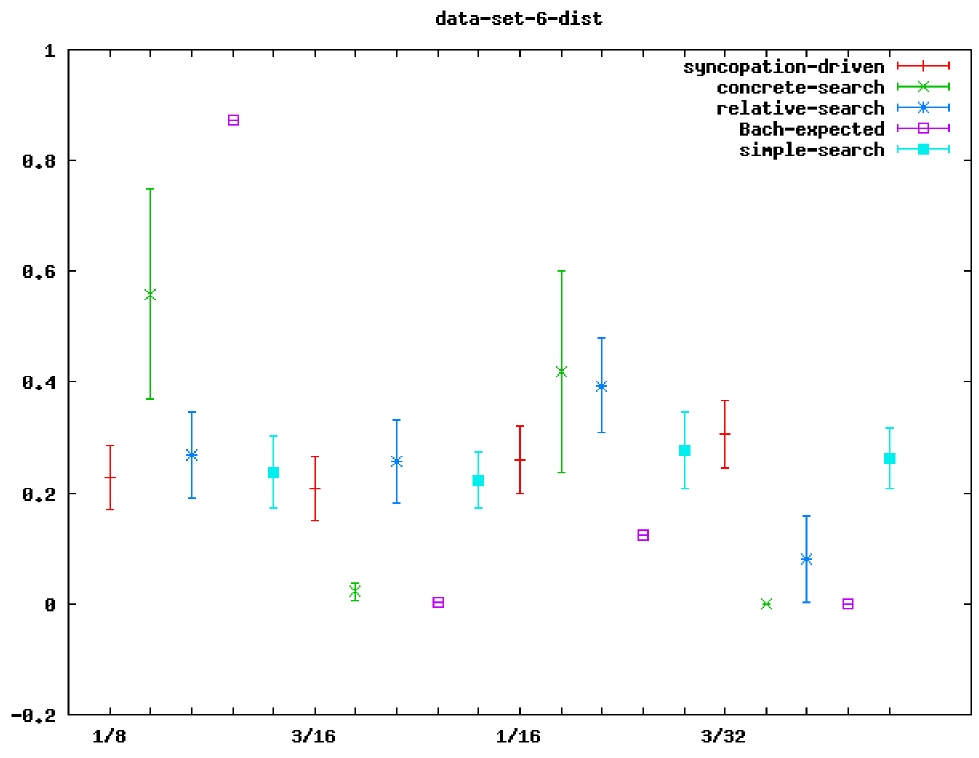

Figure 23. Distribution for the event set consisting of an eighth note, a dotted eighth note, a sixteenth note and a dotted sixteenth note. Here again, Concrete Search fails to produce dotted eighth notes or dotted sixteenth notes in its output. All other algorithms do produce acceptable distributions of the event set.

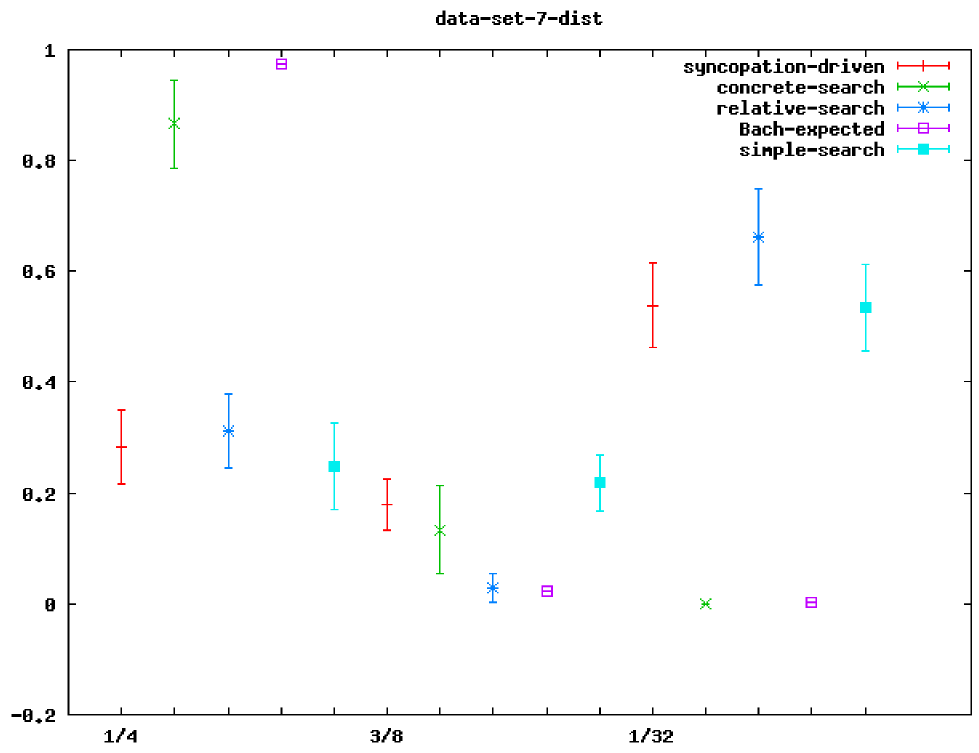

Figure 24. Distribution for the event set consisting of an eighth note, a dotted eighth note, a sixteenth note, and a dotted sixteenth note. Here, Concrete Search again fails to generate events in the legal event set, specifically $\frac{1}{32}$. Relative Search, while outputting some $\frac{1}{32}$ events does not produce them sufficiently high numbers to render Relative Search useful for this data set. This suggests that while Relative Search is able to produce suitable output for more data sets than Concrete Search, it is possible to contrive event sets for which Relative Search is unable to accommodate suitable output.

4 Algorithm Performance

In this section, the performance of the algorithms is examined. As stated in section 2.4, depth first searches have an algorithmic performance of $O(bm)$, where b is the branching factor at each node and m is the maximal depth explored in the tree. If the algorithm backtracks, as ours do, then the worst case performance is equviealent to that of a breadth first search: $O(b^m)$. The maximal depth m is limited within our problem domain by the fact that the searches do not extend beyond G, the duration of the musical chunk to be filled by the search process (2 measures for example). The branching factor b is bound by the number of events within the user specified event set. The depth m is bound by the event $e_i \in L$ such that $\forall e_i \forall e_j, j \neq i, e_i < e_j$. Given e_i, $m = \frac{L}{e_i}$.

A series of performance measurements show that the algorithms backtrack much less than one would think, and when they do backtrack do so under a common set of conditions. One thousand rhythm templates were generated with each algorithm under different event sets and the number of backtracks and nodes created was recorded. The rhythm templates created are all four measures of common ($\frac{4}{4}$) time.

Note that in most data sets explored in this section Relative Search produces on average fewer nodes than either Simple Search or Syncopation Search but demonstrates a much wider standard deviation. This is because the nature of using Markov tables shrinks the branching factor at many nodes in the tree to less than $|L|$. Because the branching factor at a given node may actually be 0 (3.1) more backtracks may occur within relative search, though this is not gauranteed by the algorithm's parameters. The end result is that relative search shows within the number of nodes it creates a very wide standard deviation but fewer nodes created on average. Relative Search also backtracks more frequently. Because Syncopation Search uses the exact same node creation procedure as simple search (with different utility computation), the two algorithms have an equivalent average number of nodes created.

4.1 Data Set 1

The first data set uses as L the rhythms $\frac{1}{8}, \frac{1}{4}$ and $\frac{3}{8}$. This set of rhythms is, generally speaking, well represented in popular music such as jazz standards and therefore represents a "common" data set. No algorithm backtracks with great frequency 25 and all three algorithms produce approximately the same number of nodes on average in the search tree 26.

4.2 Data Set 2

The second data set was selected to compare Relative Search and Syncopation Search under conditions that would stress Relative Search 22 23 24. The events in use, $\frac{1}{8}$ and $\frac{1}{12}$, have between them a relationship of $\frac{\frac{1}{12}}{\frac{1}{8}} = \frac{2}{3}$, which is poorly represented within the Bach data set 14. As expected, Relative Search both backtracks and generates more nodes than either Simple Search or Syncopation Search. Indeed both Simple Search and Syncopation Search backtrack very infrequently 27 and generate an average number of nodes approximately equal to the bottom end of Relative Search's standard deviation of nodes created 28.

4.3 Data Set 3

The third data set is an example of how Relative Search can backtrack much more frequently than Simple Search or Syncopation Search 29 and yet still have a smaller number of nodes created on average 30. This figure is somewhat misleading, however, as the standard deviation of Relative Search's nodes created is very wide. The events within this data set are $\frac{1}{8}$, $\frac{3}{16}$, $\frac{1}{16}$ and $\frac{3}{32}$.

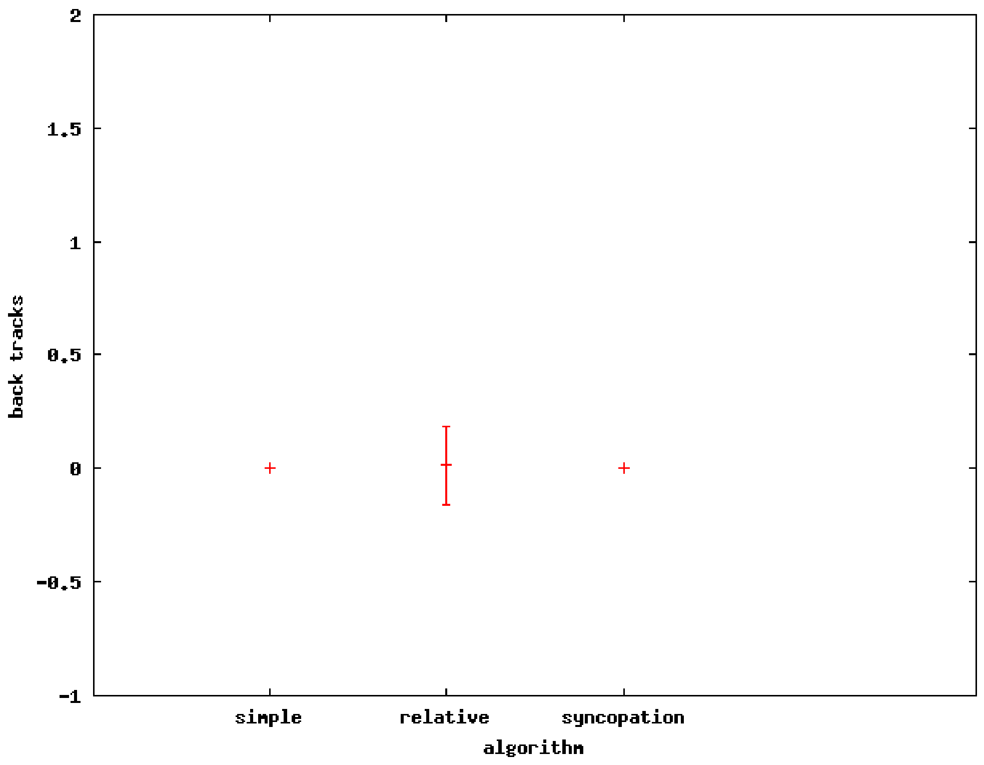

Figure 25. Backtracks for an event set consisting of rhythms $\frac{1}{8}, \frac{1}{4}, \frac{3}{8}$. For this data set no algorithm backtracks with great frequency. Relational Search has an average that is near zero but has a wide standard deviation. This suggests that Relational Search rarely backtracked for this event set but occasionally backtracks a lot.

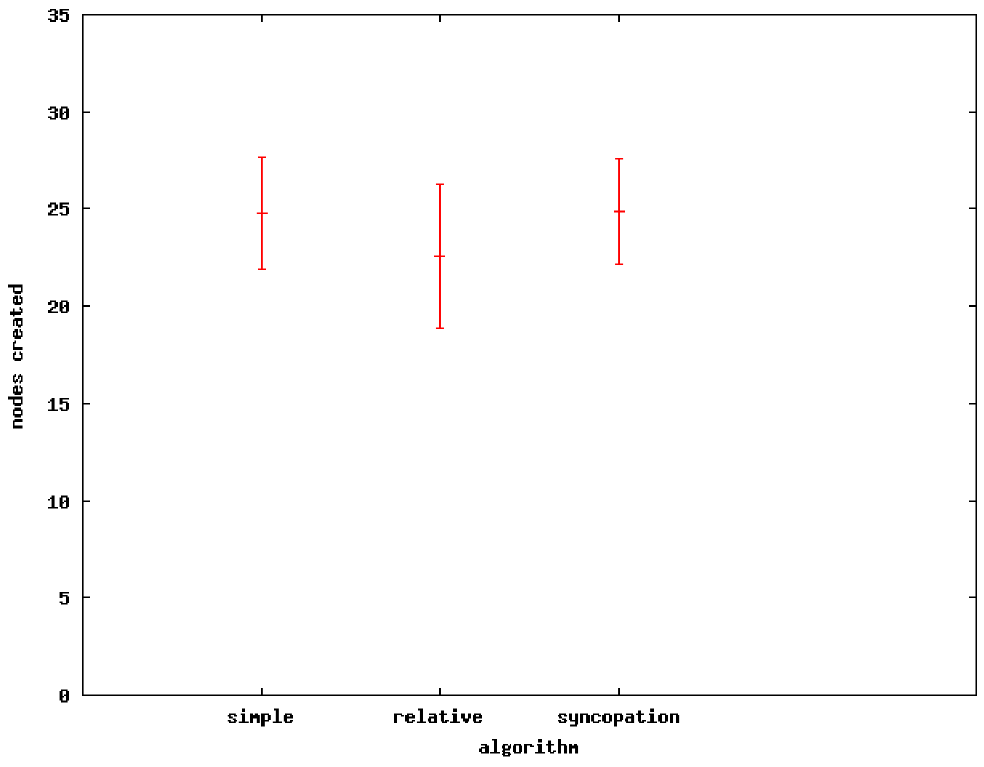

Figure 26. Nodes created for an event set consisting of rhythms $\frac{1}{8}, \frac{1}{4}, \frac{3}{8}$. Relational Search creates fewer nodes on average because the Markov table limits the branching factory at each step of the search tree.

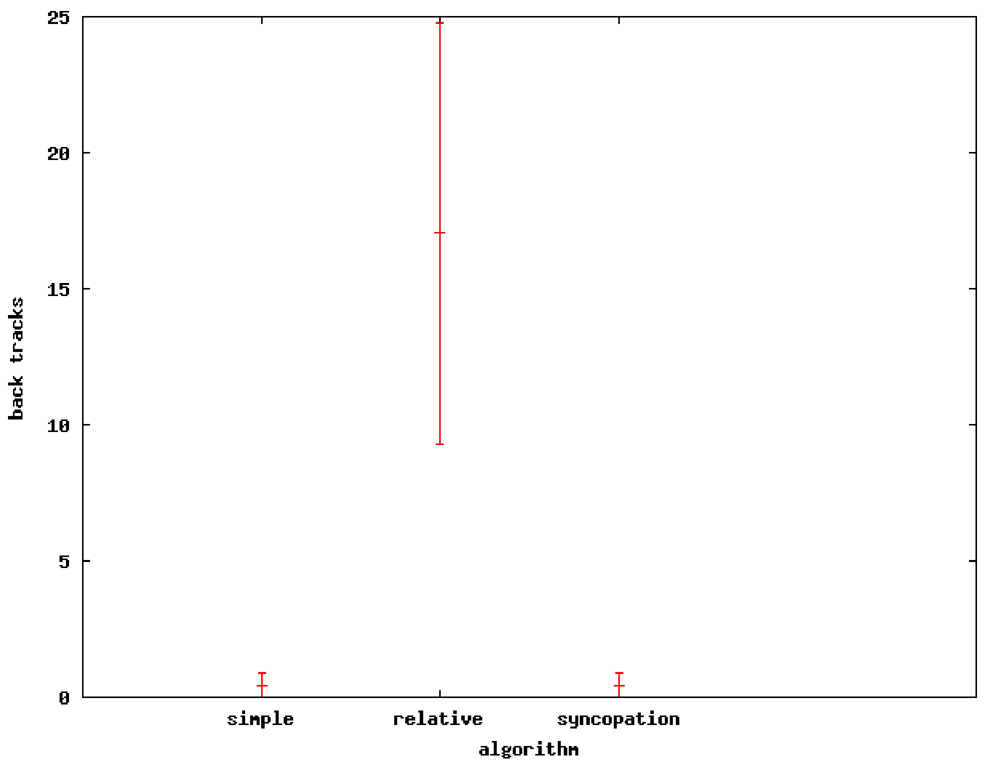

Figure 27. Backtracks for an event set consisting of rhythms $\frac{1}{8}, \frac{1}{12}$. The number of backtracks between algorithms in this case is quite significant. Simple Search and Syncopation Search back track very little on average, with very small standard deviations. Relational Search backtracks on average an order of magnitude more frequently and shows a very wide standard deviation.

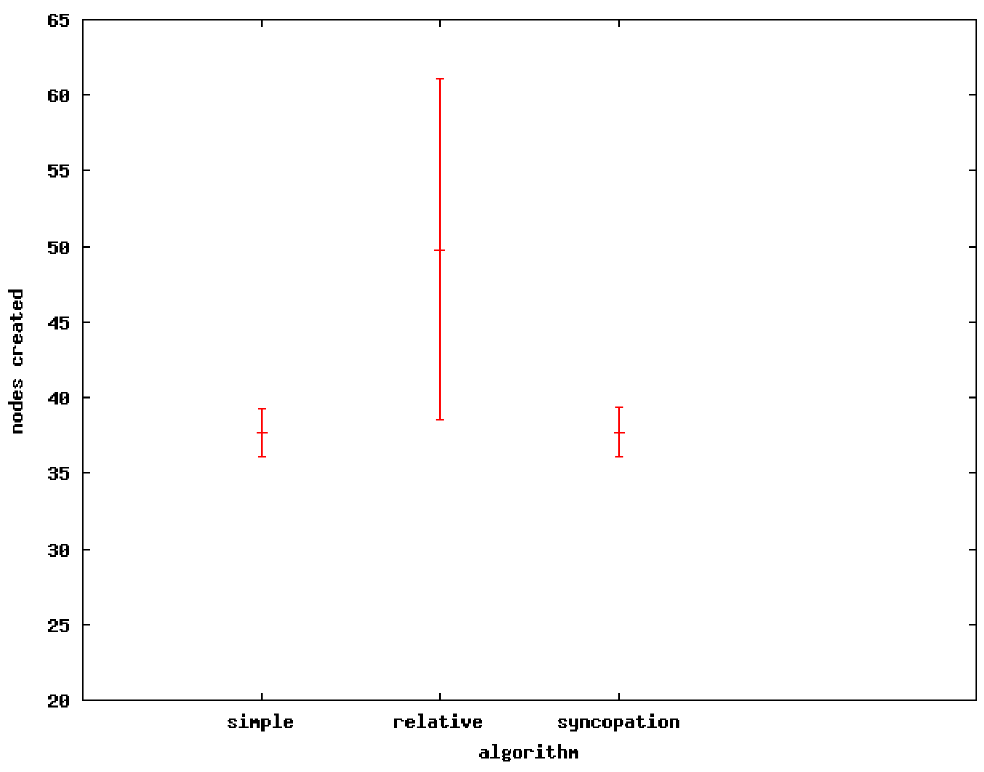

Figure 28. Nodes created for an event set consisting of rhythms $\frac{1}{8}$, $\frac{1}{12}$. As with all the data sets in this section, Simple Search and Syncopation Search generate on average the same number of nodes. Relational Search generates on average far more nodes than the other two algorithms measured. This makes sense because Relational Search backtracks far more frequently.

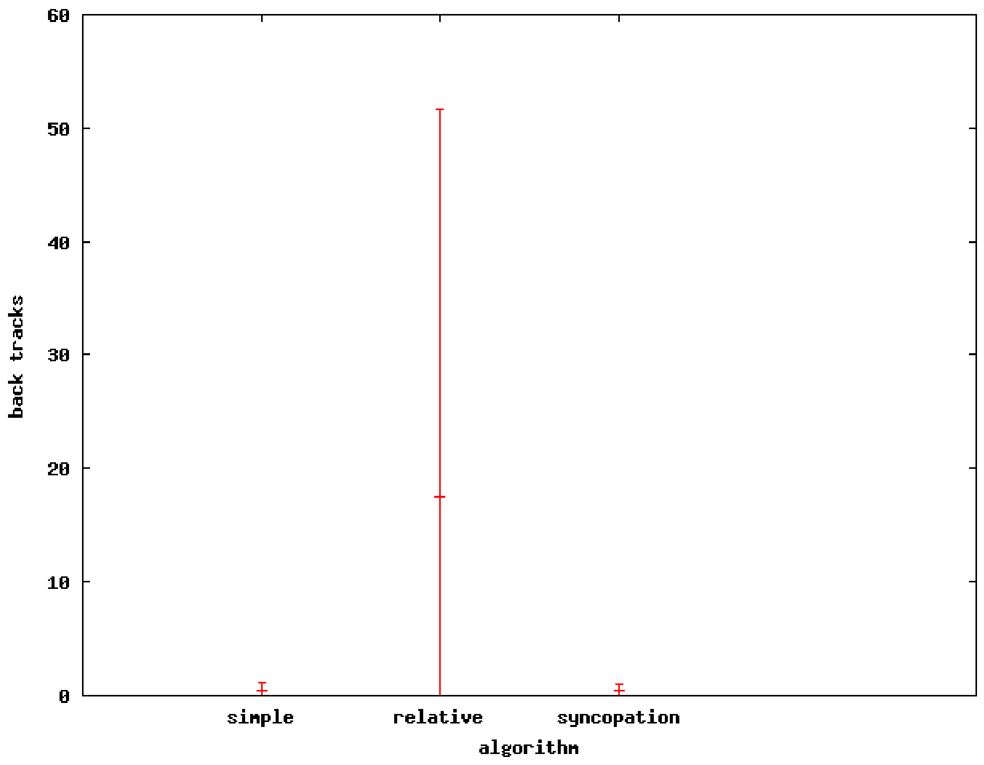

Figure 29. Backtracks for an event set consisting of rhythms $\frac{1}{8}, \frac{3}{16}, \frac{1}{16}, \frac{3}{32}$. Like the other data sets we measured Simple Search and Syncopation Search rarely backtrack, while Relational Search backtracks on average an order of magnitude more. Relational Search's standard deviation for this data set also shows how Relational Search can backtrack on a very wide latitude.

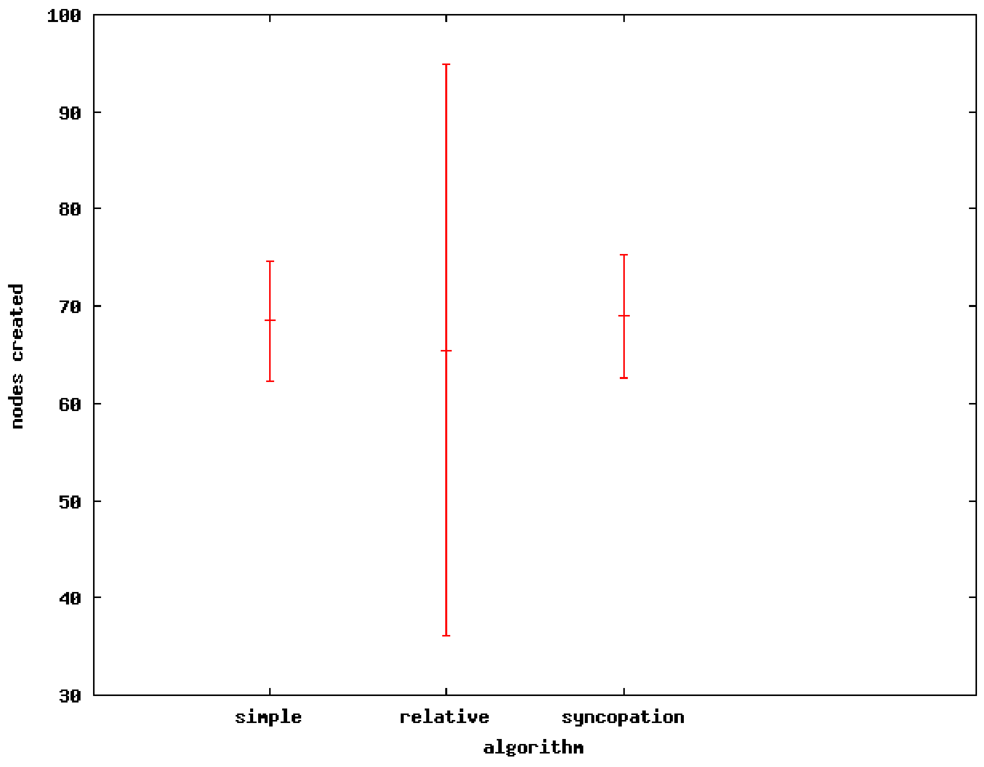

Figure 30. Nodes Created for an event set consisting of rhythms $\frac{1}{8}, \frac{3}{16}, \frac{1}{16}, \frac{3}{32}$. Despite backtracking more frequently, Relational Search produces on average few nodes. This suggests that while Relational Search backtracks more frequently than Syncopation Search and Simple Search, it does not back track so frequently that it erases all child nodes at a given spot in the search tree.

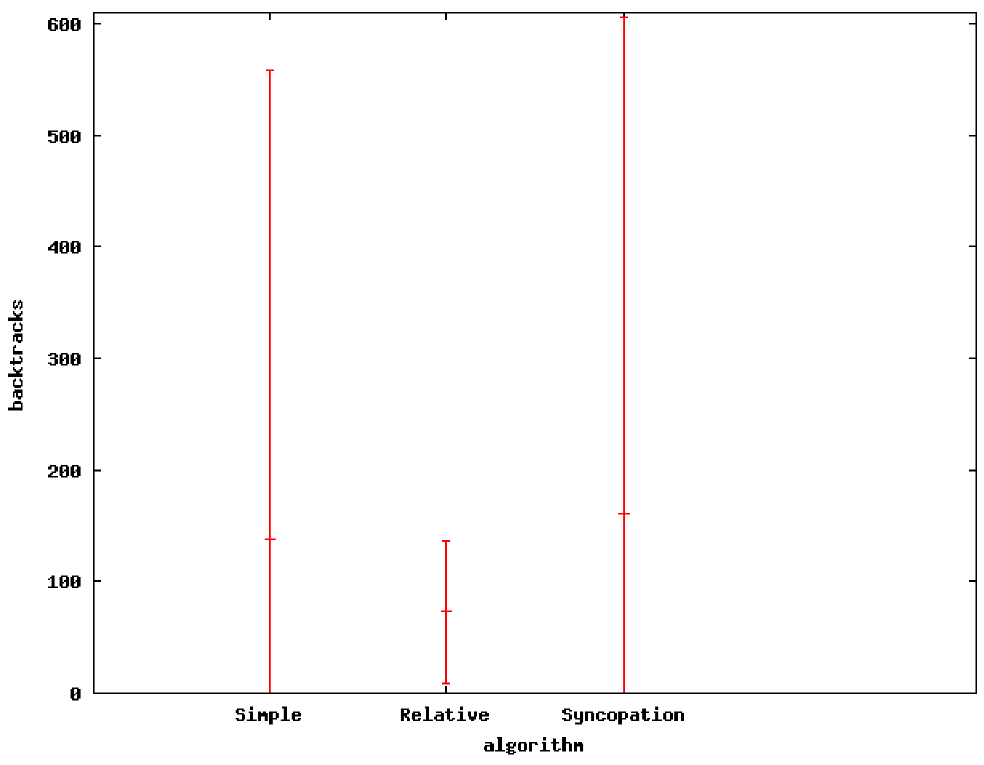

Figure 31.

5 Rhythms Survey

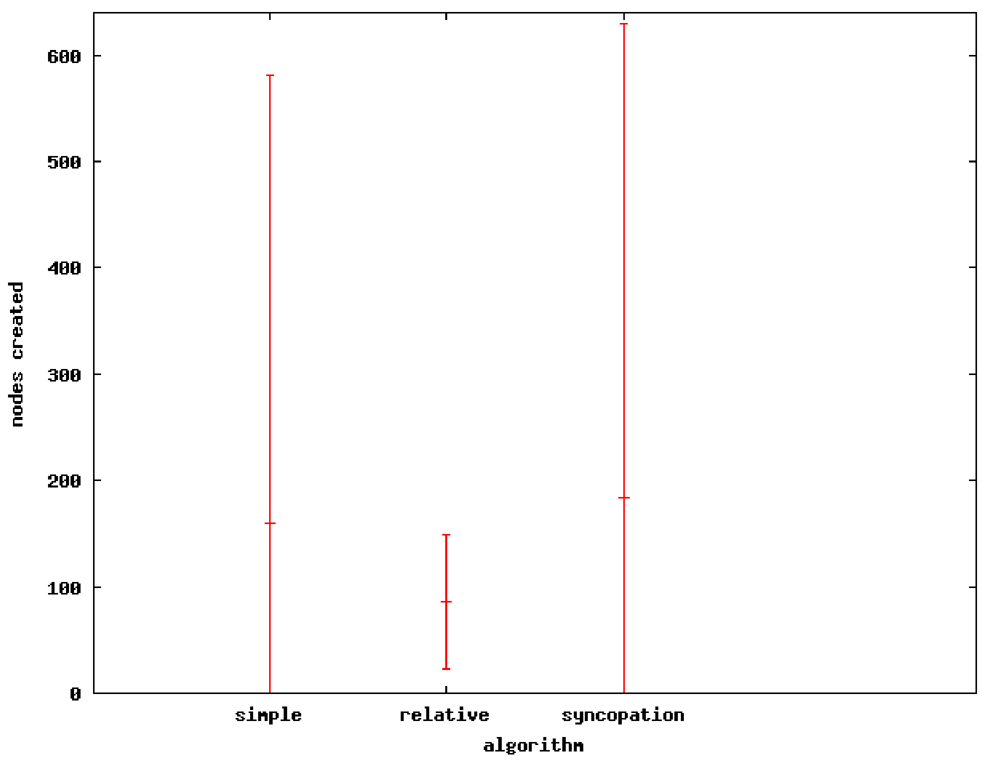

Figure 32.

6 Survey

In attempting to validate the use of more sophisticated rhythm generation algorithms rather than a simple randomized selection we set up an on line survey for voluntary participants. Participants were asked fifteen questions, each of which asked the participant to select their preferred rhythm as between two rhythms generated by two algorithms. That is to say that five questions pertained to a pairing of Simple Search against Relational Search, five questions pertained to a pairing of Simple Search against Syncopation Search and five questions pertained to a pairing of Relational Search against Syncopation Search. Question order was randomized and the participants were not told which algorithm's output they were hearing.

Scoring of each question type was done by assigning zero or one as values to a participant's selection of an algorithm's output. For example, in a question that paired output from Relational Search to output from Simple Search the value one is assigned if the participant selected Relational Search and zero if the participant selected Simple Search. These value assignments are then summed and a total score for the question type is given. In the mentioned example, a higher score shows a participant more frequently selected Relational Search over Simple Search. The population for the set of sums is 0 ... 5 for all three algorithm pairings.

In all three question types we assume as our null hypothesis that output from each algorithm is equivalently pleasing to a human listener. Therefore, the distribution of all participants for a given question type should average to 2.5 in a Gaussian distribution. The likelihood that our sample deviation differs to the extent that is does from the expected average of the null hypothesis is then measured by a t test.

6.1 Output Parameters

All sample output for the survey was done using the event set $L = \frac{1}{8}, \frac{1}{4}, \frac{3}{8}, \frac{1}{16}, \frac{3}{32}$. This event set was chosen with several goals in mind. It is fairly large in the number of events and range of durations and exhibits relation ships between events that are prevalent within the Bach data set distributions (figure 14). Specifically, relationships $\frac{1}{2}, \frac{1}{4}$ and 2 are very well represented. This gives Relational Search a reasonable set of input events with which to work such that a diverse set of outputs can be created by that algorithm. The input set is also comprised of rhythms that are, in my experience, likely to be found in virtually all musical styles.

6.2 t test

In this section the t test is described for those who are not familiar with it and may be skipped by readers who are.

The t test evaluates the probability p that some measured phenomena λ with standard deviation $\bar{s}$ happened given some assumed or measured phenomena γ. γ can come from previous measurements, or can be a pure construct of the null hypothesis. The following equation computes the t value:

$t = \frac{\lambda - \gamma}{\bar{s}}$

This t value is then compared against a table of t values with degrees of freedom to give p [17]. If the p value is small, one can say that given γ measured event λ is unlikely to have happened under normal statistical deviation given by γ's distribution.

6.3 Simple Search and Relational Search

The histogram of participant's scores for the Simple Search/Relational Search algorithm pairing shows a superficial preference for Relational Search (figure 33). The difference between the measured average and the null hypothesis in conjunction with a low p value (figure 6.3) give a marginal probability that the results of the survey are significant in their bias towards Relational Search. We can't say with statistical certainty that our null hypothesis is incorrect, but we do have some idea that Relational Search produces rhythms that are preferable to purely random output.

t values table					
Algorithms	Average	Std. Deviation	difference	t value	probability
Simple and Relational	3.75	1.47	1.25	0.82	0.30
Simple and Syncopation	2.37	1.07	0.13	0.12	0.39
Relational and Syncopation	1.24	1.28	1.26	0.98	0.24

6.4 Simple Search and Syncopation Search

In the evaluation of Simple Search and Syncopation Search, value zero was given for the selection of Simple Search and value one was given for the selection of Syncopation Search. The histogram for participant's scores when Simple Search and Syncopation Search shows much less of a bias in either direction (figure 34). In fact, the sample population's average is very close to that of the expected average and the high p value of 0.39 suggests that our null hypothesis in this case is accurate. If there is a bias in algorithms, it is very slightly in favor of Simple Search.

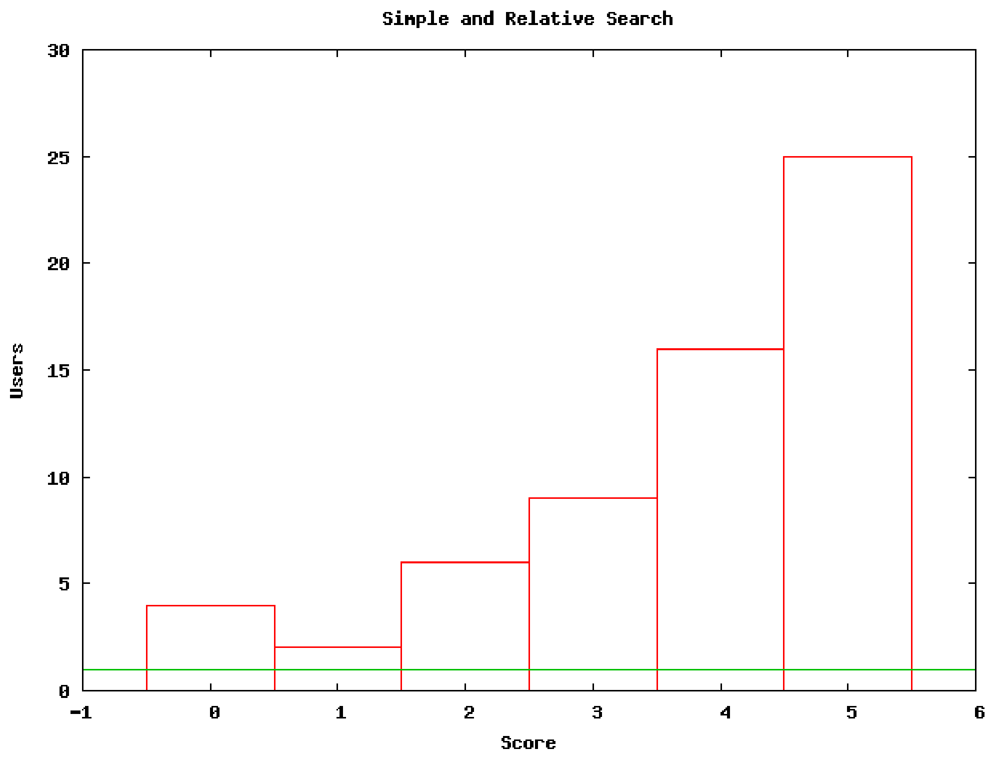

Figure 33. Histogram for Simple Search and Relational Search

These results are some what disappointing to us; we expected the syncopation algorithm to score quite well. We hypothesize that the set of valid input events was sufficiently large and varied that the syncopation algorithm functioned quite closely to that of the Simple Searching algorithm, and thus produced output that was similarly disordered. This in itself points to room for improvement within the heuristic and we would like to test Syncopation Search in future surveys with different input sets.

6.5 Relational and Syncopation Search

The final question pairing juxtaposed output from Relational Search and Syncopation Search. Value zero was assigned to Relational Search with value 1 assigned to Syncopation Search. The histogram (figure 35)shows a bias towards Relational Search that is further enforced by the t test (6.3)

Curiously, the margins of difference between the expected average of the null hypothesis and the measured average in the pairing of Relational Search and Simple Search and the

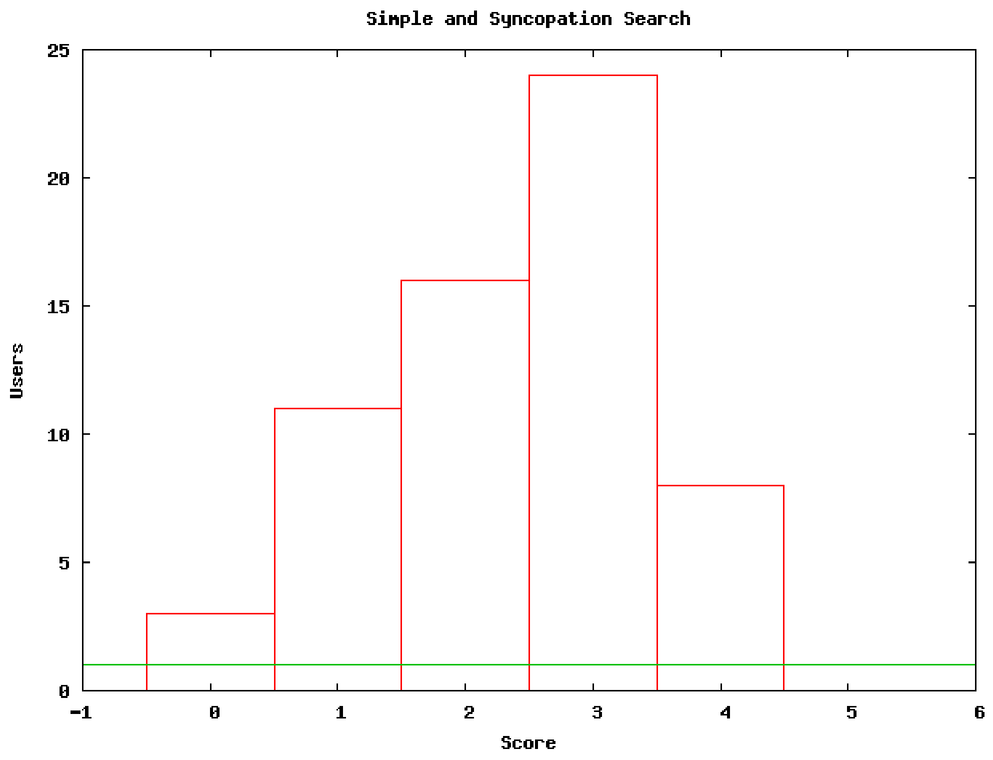

Figure 34. Histogram for Simple Search and Syncopation Search

measured average in the pairing of Relational Search and Syncopation Search are very close. This is further reason to believe that under such a large legal input set Syncopation Search is behaving very closely to Simple Search, producing output that is very similar. However, the low t value does not let us state with statistical certainty the extent to which Relational Search generates more pleasing rhythms than Syncopation Search.

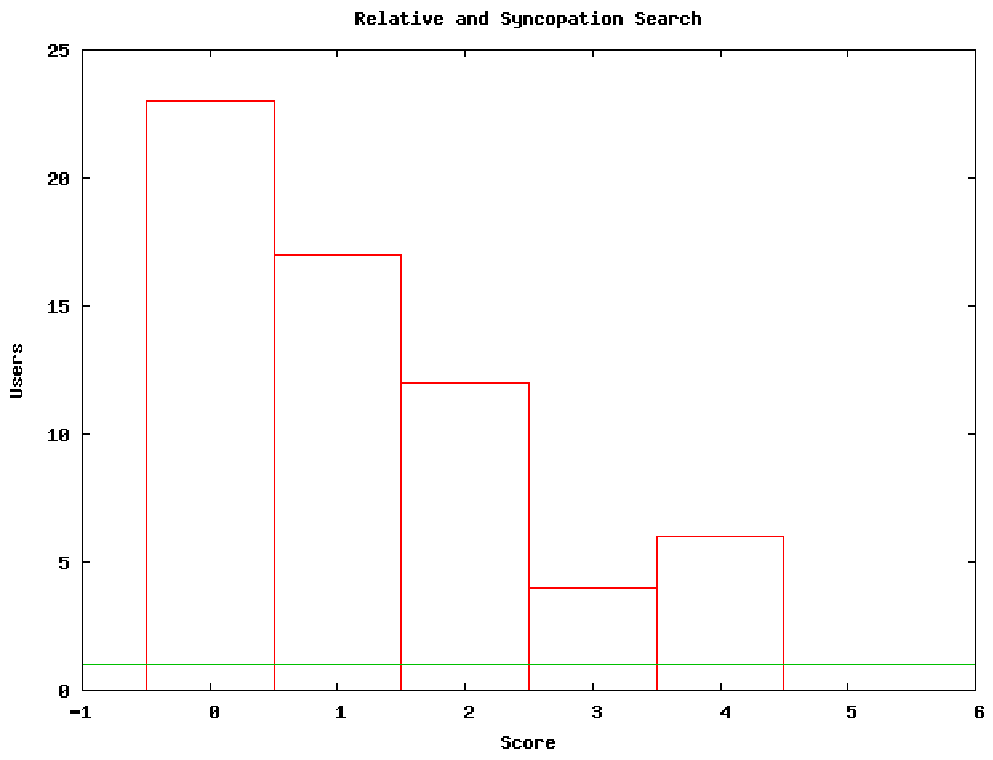

Figure 35. Histogram for Relational Search and Syncopation Search

7 Conclusion

We have shown that modified Markov look up algorithms can generate rhythms in highly constrained search environments where the set of legal events is likely to be highly limited and each event is expected to be present in the output. Using relationships between events rather than events themselves gives a larger set of possible output strings, thereby rendering an algorithm more useful to a pedagog.

We have also shown how syncopation can be used as a utility metric in ordering nodes within depth first search trees. Because syncopation is frequently employed in the same patterns across genres and composers (see section 3.6) it provides a natural ordering on rhythm events within rhythm strings.

Conclusions as to which algorithm is "best" are hard to draw without looking at a specific context and specific input set. In some sense, the best algorithm should be selected by the user based on desired output. Relational Search does the best job of producing well distributed events for all input sets while Simple Search can generate distributions for events that adhere to user specified weights. Concrete Search generates rhythms that most closely resemble the original data set even in a backtracking environment.

In our personal opinion the rhythms generated by Syncopation Search most closely resemble what we conceive of as well crafted rhythms, though this opinion is not represented in the survey we administered.

7.1 Future Work

The potential for further research in the domain of constrained composition environments is limitless. We shall therefore restrict the discussion to specific improvements to the algorithms presented. We will also discuss future additions to the étude generation software in which the algorithms were implemented.

7.1.1 Data Sets

The first obvious improvement to the Markov chain algorithms is a more dynamic data set from which to build the tables. The Bach chorales function as an excellent data set with which to test our methods but more interesting data sets should in theory produce more interesting music. The algorithms presented could easily be expanded to use tables built for a specific voice within a data set. For example, the lead violin within the Beethoven String Quartets would provide a very different rhythm string signature than the Cello part, and either part would be of benefit to an experienced pedagog.

7.1.2 Rest in Rhythms

Just as an ability to dial an amount of syncopation is desired within the composition software, we would like to give the user and ability to dial the amount of rest present within the output. Silence is a critical component in all music and one that is often very challenging for novice music readers to execute properly. Furthermore, rest in music is a critical component at implying syncopation. Syncopation is also a skill with which novice music readers struggle and would be of enormous benefit to music students. At present, the étude generating software does nothing to account for rests in output.

7.1.3 Syncopation Heuristic

The use of syncopation as a heuristic for generating rhythms is very promising. I would like to see it combined with Markov tables in such a way that further ordering on events in the search tree occurs when multiple candidates are of equal utility. The Markov tables themselves could be constructed in such a way that included a syncopation measurement in the table structure. That is to say that at each event e_i preceded by events $e_{i-1}...e_{i-n}$ we compute the syncopation value at e_i and add it as a parameter to event $e_i...e_{i-n}$.

An algorithm that orders its events using a Markov table constructed in this way would have to maintain a computation of the syncopation value at each step in the search tree and pass it as a parameter to the look up procedure. The set of candidate rhythms and corresponding utilities could then be further ordered based on additional output considerations, or simply placed into the heap of tree nodes untouched.

7.1.4 Polyphony

The last major component I'd like to see brought into the rhythm generating algorithms is a deliberate effort to create cogent rhythms when multiple voices are present. The issues involved with polyphony are normally thought of as harmonic and melodic issues but the potential for rhythmic and metric dissonance within polyphony is great if not composed with care. An algorithm that generates rhythms in a polyphonic environment would have to do so with respect to other voices that may already be composed, or would have to compose multiple voices simultaneously. It seems unlikely that the rhythm generating algorithms presented here could be directly mapped but my hope is that the basic techniques, especially the use of relationships between events, could function as a building block for polyphonic music generation.

8 Rhythm Event Selection Source Code

The following source code is the Java method used in all the algorithms we implemented. The method first normalizes the weights (computed in algorithm specific methods) by sorting the nodes on their weights then summing the weights for the individual nodes. Each node's probability is then set to its previous weight divided by this sum. A random number is generated and the first node with a probability less than or equal to this random number is selected as the new choice for the given step in the search iteration. The choice's weight is then set to an arbitrary constant so that it moves to the head of the queue.

```
protected void makeChoice(PriorityQueue<SearchTab> lastElement) {
  assert !lastElement.isEmpty();
  if (lastElement.size () == 1){
    return;
  }
  Vector<SearchTab> aux = new Vector<SearchTab>();
  Vector<SearchTab> q = new Vector<SearchTab>(lastElement);
  lastElement.clear();
  Collections.sort(q, new SearchTab.GreaterThan());
  Iterator<SearchTab> iter = q.iterator();
  SearchTab choice = null;
  while(iter.hasNext()){
```

```
  aux.add(new SearchTab(iter.next()));
}

BigFraction sum = BigFraction.zero;


for(int i = 0; i < aux.size(); ++i){
  sum = sum.add(aux.get(i).getUtility());
}

for (int i = 0; i < aux.size(); ++i){
  aux.get(i).setUtility(aux.get(i).getUtility().divide(sum));
}

Collections.sort(aux, new SearchTab.GreaterThan());
for (int i = 1; i < aux.size (); ++i){
  aux.get(i).setUtility(
      aux.get(i).getUtility().add(
          aux.get(i - 1).getUtility()));
}



BigFraction roll = new BigFraction(
  Globals.random.nextInt(99) + 1, 100
  );

for (int i = 0; i < aux.size() && choice == null; ++i){
  if (roll.lessEqualTo(aux.get(i).getUtility())){
    choice = aux.get(i);
```

```
      }
    }

    assert choice != null;
    choice.setUtility(new BigFraction (10000));
    iter = q.iterator();
    while(iter.hasNext()){
      SearchTab tab = iter.next();
      if(!tab.equals(choice)){
        lastElement.offer(tab);
      }
    }
    lastElement.offer(choice);
  }
```

8.1 Concrete Search Weighting Source Code

The Concrete Search Java class takes as a parameter the tree constructed so far ($P(T)$), the length of G consumed so far, the length of G remaining, the subset of user supplied rhythm events L‘ and a few other objects of extraneous importance. A subsequence of the tree up to the length of a user supplied “order” is constructed and its rhythm events are concatenated into a string representation that serves as the key into the Markov Tables. The entire set of potential next events are returned along with their probabilities. The method then iterates through the set of next events and for each event, if it is located in L‘, it is added to the set of child nodes.

```
  protected Vector<SearchTab> createTreeNode(
      Vector<PriorityQueue<SearchTab>> searchTree,
      BigFraction consumed,
      BigFraction left_over, SortedSet<BigFraction> less_than,
      Voice voice, Section section) {
    Vector<SearchTab> node = new Vector<SearchTab>();
```

```
Iterator<BigFraction> iter = less_than.iterator();
int size = searchTree.size();
if(size == 0){
  while(iter.hasNext()){
    BigFraction next = iter.next();
    SearchTab tab = new SearchTab(next);
    tab.setUtility(new BigFraction(1));
    node.add(tab);
  }
}
else if(size == 1){
  BigFraction oneAgo = searchTree.get(size - 1)
    .peek().getDuration();
  RhythmTemplate.TableEntry[] table = this.getOrder1()
    .get(oneAgo.toString());
  if (table == null){
    return node;
  }
  for(int i = 0; i < table.length; ++i){
    if(less_than.contains(table[i].getEvent())){
      SearchTab tab = new SearchTab(table[i].getEvent());
      tab.setUtility(table[i].getProbability());
      node.add(tab);
    }
  }
}
else{
  int n = size > this.getOrder() ? this.getOrder() : size;
  BigFraction [] fractions = new BigFraction[n];
```

```
      for(int i = 0; i < n; ++i){
        int place = size - n + i;
        fractions[i] = searchTree.get(place).peek().getDuration();
      }
      String key = "";
      for(int i = 0; i < n - 1; ++i){
        key += fractions[i].toString() + " , ";
      }
      key += fractions[n - 1].toString();
      RhythmTemplate.TableEntry[] table = this.getTable(n).get(key);
      if (table == null){
        return node;
      }
      for(int i = 0; i < table.length; ++i){
        if(less_than.contains(table[i].getEvent())){
          SearchTab tab = new SearchTab(table[i].getEvent());
          tab.setUtility(table[i].getProbability());
          node.add(tab);
        }
      }
    }

    return node;
  }
```

8.2 Relational Search Weighting Source Code

The Relational Search Java class operates similarly to the Concrete Search class. However, instead of using the rhythm events themselves to construct the table key it divides each event by the previous event and uses this value to construct the table key.

```
protected Vector<SearchTab> createTreeNode(
      Vector<PriorityQueue<SearchTab>> searchTree,
      BigFraction consumed,
      BigFraction left_over, SortedSet<BigFraction> less_than,
      Voice voice, MusicForm.Section section) {
    BigFraction total = consumed.add(left_over);
    Vector<SearchTab> node = new Vector<SearchTab>();
    String key;
    int size = searchTree.size();
    if (size == 0){
      Iterator<BigFraction> iter = less_than.iterator();
      while(iter.hasNext()){
        BigFraction next = iter.next();
        SearchTab tab = new SearchTab(next);
        tab.setUtility(new BigFraction (1 + Globals.random.nextInt(99)));
        node.add(tab);
      }
    }
    else if (size == 1){
      BigFraction oneAgo = searchTree.get(size - 1).peek().getDuration();
      RhythmTemplate.TableEntry [] table = this.getOrder1();
      for(int i = 0; i < table.length; ++i){
        BigFraction next = oneAgo.multiply(table[i].getEvent());
        if(less_than.contains(next)){
          SearchTab tab = new SearchTab(next);
          tab.setUtility(table[i].getProbability());
          node.add(tab);
        }
      }
```

```
}
else{
  int order = size > this.getOrder() ? this.getOrder() : size;
  key = "";
  BigFraction[] fractions = new BigFraction[order];
  for(int i = 0; i < order; ++i){
    fractions[i] = searchTree.get(size - order + i).peek().getDuration();
  }

  for(int i = 1; i < order - 1; ++i){
    key += fractions[i].divide(fractions[i - 1]).toString() + " , ";
  }
  key += fractions[fractions.length - 1]
    .divide(fractions[fractions.length - 2]).toString();

  RhythmTemplate.TableEntry[] table =
    this.getOrder(order).get(key);

  BigFraction oneAgo = fractions[order - 1];

  if (table == null){
    return node;
  }
  for(int i = 0; i < table.length; ++i){
    BigFraction next = oneAgo.multiply(table[i].getEvent());
    if(less_than.contains(next)){
      SearchTab tab = new SearchTab(next);
      tab.setUtility(table[i].getProbability());
      node.add(tab);
```

```
        }
      }
    }
    this.logTreeNode(node);
    return node;
  }
```

8.3 Syncopation Search Weighting Source Code

Node weighting in the Syncopation Search algorithm is a more complicated process. The Syncopation Search Java class holds two helper classes that put weights on the nodes depending on whether or not syncopation should or should not be emphasized. The helper classes are selected by the following method in the Syncopation Search class.

```
private NodeSelector chooseSelector(
      Vector<PriorityQueue<SearchTab>> searchTree,
      Vector<MeasureBeat> beats, BigFraction phraseLength) {
    Vector<BigFraction> durations = new Vector<BigFraction>();
    for(PriorityQueue<SearchTab> queue : searchTree){
      durations.add(queue.peek().getDuration());
    }
    BigFraction sync = this.syncopationMeasurer.measureSyncopation(
        durations, beats).divide(new BigFraction(searchTree.size()));
    BigFraction quarterPhraseLength =
      phraseLength.divide(new BigFraction(4));
    BigFraction accume = accumulateSearchTree(searchTree);
    if(accume.lessEqualTo(quarterPhraseLength) ||
        accume.greaterEqualTo(
          quarterPhraseLength.multiply(new BigFraction(3)))){
      return this.demphasizor;
    }
    else{
      sync = sync.divide(new BigFraction(searchTree.size()));
      if(sync.lessEqualTo(this.syncopation)){
        return this.emphasizor;
      }
      else {
        return this.demphasizor;
      }
    }
  }
```

The method first computes the syncopation measurement as per section 3.6. Then it computes how much of L has been consumed and chooses the weighting class that de-emphasizes syncopation in the following cases:

- The sum of all rhythm events in the search tree is less than or equal to $\frac{L}{4}$.
- The sum of all rhythm events in the search tree is greater than or equal to $\frac{3*L}{4}$.
- The syncopation measurement is greater than or equal to a user supplied syncopation constant.

Otherwise the method selects the weighting class that emphasizes syncopation.

The code for the weighting classes is shown below. The search tree computed so far and $C(Q_i)$ are passed as arguments to the method. A third object passed contains the Java class responsible for computing syncopation and a data structure of musical beats representing the set of beats over which syncopation will be measured. In both classes the syncopation measurement is taken over $P(T)$, then for each rhythm event in Q_i the syncopation is computed as though the rhythm event were to become $C(Q_i)$.

The two classes differ in how the assign weights in the final loop. The syncopation emphasizing class simply uses the final syncopation measurement as the rhythm event's weight, resulting in nodes of higher syncopation values having a higher probability of selection in the node selection method. The syncopation de-emphasizing node subtracts the syncopation measurement from an arbitrary constant, resulting in nodes with a higher syncopation measurement being assigned a lower probability of selection in the node choice method.

```
static class EmphasizeSyncopation extends NodeSelector{
  public void orderChoices(Vector<PriorityQueue<SearchTab>> tree,
      Vector<SearchTab> lastElement, NodeSelectorArgs args){
    Vector<SearchTab> aux = lastElement;

    Vector<BigFraction> durations = new Vector<BigFraction>();
    for(PriorityQueue<SearchTab> tab : tree){
      durations.add(tab.peek().getDuration());
    }
    BigFraction syncopation = args.measurer.measureSyncopation(
        durations,args.beats);
    syncopation = syncopation.divide(new BigFraction(tree.size()));
    int index1 = -1, index2 = -1;
    for(int i = 0; i < args.beats.size() && index1 == -1; ++i){
      if(args.beats.get(i).consumed.equals(BigFraction.zero)){
        index1 = i;
      }
    }

    index2 = index1 + 1;

    Iterator<SearchTab> iter = aux.iterator();
    BigFraction distance = args.measurer
      .getDistanceMeasurement(args.beats.get(index1));
    while(iter.hasNext()){
      SearchTab tab = iter.next();
      BigFraction measurement = syncopation.add(
        args.measurer.syncopationFunction(tab.getDuration(),
          args.beats.get(index1), args.beats.get(index2), distance));
      tab.setUtility(measurement);
    }
  }
}

static class DeEmphasizeSyncopation extends NodeSelector{
  public void orderChoices(Vector<PriorityQueue<SearchTab>> tree,
      Vector<SearchTab> lastElement, NodeSelectorArgs args){
    Vector<SearchTab> aux = lastElement;

    Vector<BigFraction> durations = new Vector<BigFraction>();
    for(PriorityQueue<SearchTab> tab : tree){
      durations.add(tab.peek().getDuration());
    }
    BigFraction syncopation = args.measurer.measureSyncopation(
        durations,args.beats);
    syncopation = syncopation.divide(new BigFraction(tree.size()));
    int index1 = -1, index2 = -1;
    for(int i = 0; i < args.beats.size() && index1 == -1; ++i){
      if(args.beats.get(i).consumed.equals(BigFraction.zero)){
        index1 = i;
      }
    }

    index2 = index1 + 1;
```

```
      Iterator<SearchTab> iter = aux.iterator();
      BigFraction distance = args.measurer
        .getDistanceMeasurement(args.beats.get(index1));
      BigFraction ceiling = new BigFraction(100);
      while(iter.hasNext()){
        SearchTab tab = iter.next();
        BigFraction measurement = syncopation.add(
          args.measurer.syncopationFunction(tab.getDuration(),
            args.beats.get(index1), args.beats.get(index2), distance));
        tab.setUtility(ceiling.subtract(measurement));
      }
    }
  }
```

8.4 Syncopation Measurement

The code for pertinent methods in the Java class responsible for computing the syncopation measurements is shown below.

```
  public BigFraction measureSyncopation(Vector<BigFraction> durations,
      Vector<MeasureBeat> beats){
    int beatIndex = 0;
    BigFraction summation = BigFraction.zero;
    for(MeasureBeat beat : beats){
      beat.resetConsumed();
    }
    for(int i = 0; i < durations.size(); ++i){
      BigFraction duration = durations.get(i);
      MeasureBeat beat = beats.get(beatIndex);
      if(beat.durationRemaining().equals(BigFraction.zero)){
```

```
      beatIndex++;
      beat = beats.get(beatIndex);
    }
    BigFraction distance = this.getDistanceMeasurement(beat);
    BigFraction cost = this.syncopationFunction(
      duration, beat, beats.get(beatIndex + 1),
        distance);
    summation = summation.add(cost);
    beatIndex = this.consumeDuration(duration, beatIndex, beats);
  }

  return summation.divide(new BigFraction(durations.size()));
}

public BigFraction getDistanceMeasurement(MeasureBeat beat){
  BigFraction remaining = beat.durationRemaining();
  if(beat.getConsumed().lessEqualTo(remaining)){
    return beat.getConsumed();
  }
  else{
    return remaining;
  }
}

  public BigFraction syncopationFunction(BigFraction duration,
    MeasureBeat firstBeat,
    MeasureBeat secondBeat, BigFraction distance){
  if(firstBeat.consumed.equals(BigFraction.zero)){
    return this.getCase0Constant();
    }
    else if (firstBeat.getConsumed().add(duration)
      .lessEqualTo(firstBeat.getDuration())){
      return this.getCase1Constant().divide(distance);
    }
    else{
      if (duration.subtract(firstBeat.durationRemaining()).greaterThan(
        secondBeat.getDuration())){
        return this.getCase3Constant().divide(distance);
      }
      else{
        return this.getCase2Constant().divide(distance);
      }
    }
  }
```

8.5 Markov Table Creation

The pertinent code for the python program that constructs the Markov tables is listed below. Listed are two functions, one for the creation of the Relative Markov tables and one for the creation of the tradition Markov tables. In both cases, the functions step through an event list that is comprised of musical rhythms parsed from music xml files.

The order of the Markov table to be mined is passed in as a parameter and for each step through the event list a sub list of size $order$ is constructed. The events of this sublist is then used to construct a key into a hashmap holding integers that count the number of times a given key has been observed. In the case of the Relational Table mining function, the key is constructed of the dividends between events in the event list.

```
def mineRelationalTable(voiceDict, table, order):
    eventLists = voiceDict.values()
```

```
if order == 1:
    topKey = "only table"
    if not table.has_key(topKey):
        table[topKey] = {}
    for eventList in eventLists:
        prev = eventList[0]
        for event in eventList[1:]:
            multipland = event.duration / prev.duration
            if not table[topKey].has_key(multipland):
                table[topKey][multipland] = 0
            table[topKey][multipland] += 1
            prev = event

else:
    for eventList in eventLists:
        prev = eventList[0:order]
        for event in eventList[order:]:
            multiplands = ([(prev[i].duration / prev[i - 1].duration)
                                for i in xrange(1, order)])
            lastMultipland = event.duration / prev[order - 1].duration
            topKey = " , ".join([str(m) for m in multiplands])
            if not table.has_key(topKey):
                table[topKey] = {}
            if not table[topKey].has_key(lastMultipland):
                table[topKey][lastMultipland] = 0
            table[topKey][lastMultipland] += 1

            prev.pop(0)
```

```
                prev.append(event)




def mineConcreteTable(voiceDict, table, order):
    eventLists = voiceDict.values()
    for eventList in eventLists:
        prev = eventList[0:order]
        for event in eventList[order:]:
            key = " , ".join([str(p.duration) for p in prev])
            if not table.has_key(key):
                table[key] = {}
            if not table[key].has_key(event.duration):
                table[key][event.duration] = 0
            table[key][event.duration] += 1

            prev.pop(0)
            prev.append(event)
```

8.6 Survey Sample Question

The image below is a sample question from the survey administered. The actual questions did not have the algorithm name as a title to the written out notation.

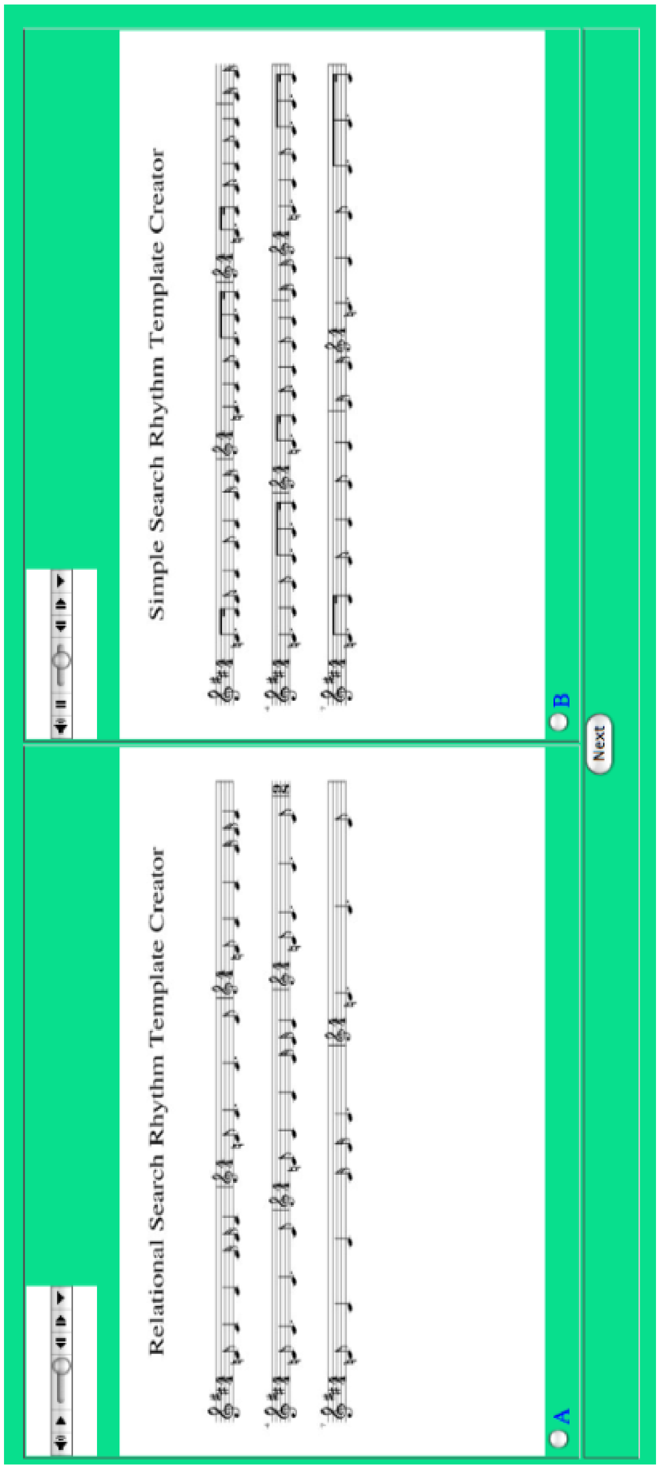
Relational Search Rhythm Template Creator
A
Simple Search Rhythm Template Creator
B
Next

Figure 36.

References

[1] Feburary 2009.

[2] September 23.

[3] C. Ames. *Automated Composition in Retrospect*. Leonardo, 1987.

[4] J. Bharucha and P. Todd. Modeling the perception of tonal structure with neural nets. *Music Perception*, pages 1–30, 1987.

[5] J. Biles. Genjam: a genetic algorithm for generating jazz solos. In *Proceedings of the 1994 International Computer Music Conference.*

[6] J. Biles. Interactive genjam: Integrating real-time performance with a genetic algorithm. In *Proceedings of the 1998 International Computer Music Conference.*

[7] A. Camurri. Artificial intelligence and music: On the role of artificial intelligence in music research. *Journal of New Music Research*, 1990.

[8] D. Cope. *Computers and Musical Style*. A-R Editions, Inc, 1991.

[9] D. Cope. One approach to musical intelligence. 1999.

[10] D. Cope. Recombinant music. 1999.

[11] D. Cope. *The Algorithmic Composer*. A-R Editions, Inc, 2000.

[12] D. Cope. *Virtual Music: Computer Synthesis of a Musical Style*. The MIT Press, 2001.

[13] D. Cope. *Computer Models of Musical Creativity*. The MIT Press, 2005.

[14] R. D., editor. *The Harvard Dictionary of Music*. Harvard University Press, 1986.

[15] J. S. Douglas Eck. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, 2002.

[16] F. G. et. all. Mathematical measures of syncopation. 2008.

[17] W. G. H. George E.P. Box, J. Stuart Hunter. *Statistics for Experimenters*. Wiley, 2005.

[18] M. A. Hall. Selection of attributes for modeling bach chorales by a genetic algorithm. 1995.

[19] L. Hiller and L. Isaacson. *Experimental Music*. McGraw-Hill, 1959.

[20] D. Hornel and W. Menzel. Learning musical structure and style using neural networks. *Computer Music Journal*, 1998.

[21] M. F. Karsten Verbeurgt, Michael Dinolfo. Extracting patterns in music for composition via markov chains, 2004.

[22] H. Lincoln, editor. *The Computer and Music*. Cornell University Press, 1970.

[23] E. R. Miranda. *Composing Music With Computers*. Focal Press, 2001.

[24] J. A. Mooror. Music and computer composition. 1972.

[25] M. C. Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing. In *Connection Science*, 1994.

[26] M. T. Peyman Sheikholharam. Music composition using combination of genetic algorithms and recurrent neural networks, 2008.

[27] C. Roads. Music composition treks. *Composers and the Computer*, 1985.

[28] C. Roads. Reasearch in music and artificial intelligence. 1985.

[29] R. Rowe. *Machine Musicianship*. MIT Press, 2001.

[30] P. N. Stuart Russel. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[31] K. Swingler. *Applying Neural Networks: A Practical Guide*. London Academic Press, 1996.

[32] I. Xenakis. *Formalized Music*. Indiana University Press, 1971.