

Heuristic Counterpoint

Timothy Ambroggi

Advisor: John Dougherty

Haverford College

April, 2004

Abstract

This paper will introduce and formalize the problem of generating 16th-century Fuxian counterpoint using search algorithms. Several algorithms will be analyzed with respect to the problem domain, and the advantages of heuristics will be considered for each algorithm. The paper culminates with an in-detail description of the algorithms in question, and how they might be implemented.

Steps to Parnassus

In 1725, Johann Joseph Fux published a treatise on music theory entitled, *Gradus ad Parnassum*, told through the imaginary interaction between a hypothetical student and his fictional music teacher. The student, Josephus, wishes to be taught how to compose polyphonic 16th-century counterpoint, a style of music writing in which the composer juxtaposes two or more musical lines so as to provide harmonic relief while at the same time retaining linear autonomy within each part. The teacher, Aloysius¹, reluctantly agrees to teach him, and over the course of the text lays out sets of rules for the student to follow when composing. The direct value of the text as a compositional tool is

¹ Aloysius is widely understood to be based on the 16th-century composer Palestrina, whose style of counterpoint is codified in Fux's treatise.

questionable, but it was one of the first conscious and effective attempts to systematize and organize the elements of a certain musical style.

Before progressing any further, let us outline the basics of Fux's method. Central to every piece of 16th-century counterpoint is a fundamental line called the *cantus firmus* (fixed melody). The composer chooses a cantus firmus, and then writes one or more lines of *counterpoint*, which are to be sung at the same time as the cantus, providing a harmonic backdrop. The counterpoint is also supposed to be a line of music in and of itself, rather than notes chosen solely on the basis of their relationship to the cantus. The choice of such notes is the primary task of the composer, and the focus of Fux's treatise. Concerns such as which notes are considered consonant against the cantus, how often leaps should occur in the various parts, and the relative motions of parts to each other are all taken into account. Rhythm, while an important element of music, is viewed rather simplistically as an even subdivision of the duration of the cantus firmus duration, with ties between notes when appropriate.

It would be impractical to attempt to encapsulate the whole of Fux's 100-page treatise into a paper of this scope, refer to Alfred Mann's English-language translation of the text. Instead of a full synthesis, I will instead explain some of the more universal central themes using modern notation:

1. *Diatonicism*: In traditional Western music, pitches are considered to have distance from other pitches. These pitch-distances are called *intervals*, and can be measured in discrete units called *half-steps*². A *scale* can be defined by a pair (I,

² Historically, like with many measurement systems, there has been some debate as to what the definition of a half-step is and should be. At the time of Palestrina, on whose music Fux's treatise is based, the prevalent definition was based on the Pythagorean tuning system. In this system, a half-step is defined as the distance between any two frequencies with a ratio of 256:243, or ~ 1.053498 . While this may seem

r), where I is a set of intervals and r is the root note. The C major scale, for instance, can be defined as $I = \{0, 2, 4, 5, 7, 9, 11\}$, and $r = \text{middle C} = 48$.³ Membership in a scale is defined by $(x-r)(\text{mod } 12)$, taking the interval between a given note x and r , and testing whether or not it is in the scale interval set I . Fux considers the Greek *modal* system for writing counterpoint, where in addition to the major scale, the scales which start on different degrees of the major scale (known as *modes*) are also valid scales for composing music with. The actual note membership set for these modes is the same as the membership set for the major scale, but the musical relationships of the notes are different. For any scale S , the set of modes of S , denoted M_S , is the set of scales given by

$$M_S = \{(S+i)(\text{mod } 12), r_S+i \mid \forall i \in I_S\}$$

For a note to be *diatonic*, it means that the note is a member of the scale in which the piece is composed.

2. *Harmonic Consonance*: Consonance is a term applied to a set of notes to determine whether or not they will be aesthetically pleasing to the listener. Obviously, a major factor in determining which note pairs will be consonant is the listener's musical preferences. Fux allows a set of intervals which he considers to be representative of the musical sensibilities of the 16th century. He claims that if the interval between two notes is in this set, then the notes are consonant. When considering more than two notes, a set of N notes is considered to be consonant if

arbitrary, it is grounded in acoustical physics. For the purposes of this paper, however, it will be advantageous to abstract away these details of this definition for the sake of theoretical simplicity. For further information on tuning systems and the motivations behind them, I recommend Peter Frazier's "The Development of Musical Tuning Systems", <http://www.midicode.com/tunings/index.shtml> .

³ Middle C is the 48th note on a standard piano keyboard. For the purposes of permitting mathematical operations on musical notes in this paper, this is the convention I will use.

all intervals between pairs of notes within the set are consonant. An important distinction exists between modern and 16th century music theory; while to modern theorists, the “chord” is the fundamental unit of harmony, to Fux it is the interval. For vertical (“chord”) consonance, Fux’s rules effectively outline the following consonance table⁴:

Name	P1	m2	M2	m3	M3	P4	T	P5	m6	M6	m7	M7	P8
							T						
Distance	0	1	2	3	4	5	6	7	8	9	10	11	12
Consonant	Y	n	n	Y	Y	n	n	Y	Y	Y	n	n	Y

Larger intervals are not necessarily dissonant, but they “should be avoided”. The consonance of larger intervals is the interval distance modulus 12.

3. *Melodic Consonance:*

- a. *Diatonic Steps:* Fux defines an effective musical line as one that follows a set of constraints pertaining to melodic motion. The first of these claims that motion of a diatonic step should be used whenever possible. A diatonic step is the distance between any two adjacent notes in a scale. This may be any number of half-steps, but is generally 1 or 2 half-steps. The reasoning behind this is that if the notes do not move by the smallest steps possible (while staying in the scale), the sense of an individual musical “line” will be lost.
- b. *Jumps:* Although motion by diatonic step is generally preferable, jumps of more than a step are sometimes desirable, provided that they move by step in the

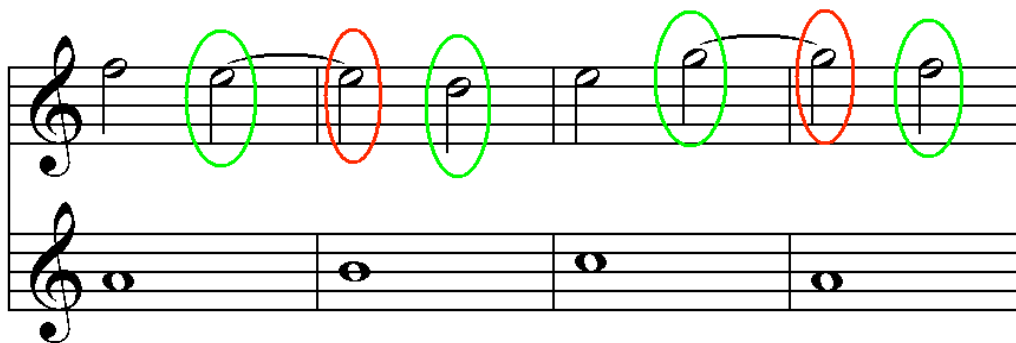
⁴ For an explanation of what the interval names represent, see Walter Piston’s “Harmony”.

opposite direction of the jump immediately after the jump, they are of an interval no greater than an octave, the target of the jump is a diatonic note, and the interval of the jump is consonant. The basic consonance table for melodic motion is:

Name	P1	m2	M2	m3	M3	P4	T	P5	m6	M6	m7	M7	P8
							T						
Distance	0	1	2	3	4	5	6	7	8	9	10	11	12
Consonant	Y	Y	Y	Y	Y	n	Y	Y	Y	Y	n	n	Y

There are additional concerns, such as resolution of the leading tone (7th scale degree), frequency of jumps, and the possibility of consecutive jumps under certain circumstances. However, to enumerate these myriad exceptional circumstances is unnecessary for the purposes of this paper.

4. *Suspensions*: Fux devotes a substantial portion of his text to the usage of suspensions, notes which are held through a note change in the cantus firmus, creating a temporary dissonance. The notion of a suspension suggests that under the correct circumstances, we can violate the rules of consonance. Again, the rules for suspension are quite involved, centering around the idea that the note must be consonant when struck, becoming dissonant upon the striking of the next note of the cantus, and then becoming consonant again on the next note of the line. The following example shows two suspensions, with the consonant intervals circled in green and the dissonant intervals circled in red.



Without suspensions, this could be written using entirely consonant intervals, as shown in the following example.



As with rules for melodic jumps, it is both laborious and unnecessary to enumerate all the rules and exceptions regarding the use of suspensions.

5. *Relative Motion:* Any interval can be considered to have a direction. The second note may be higher in pitch than the first note, in which case it is called upward motion. If the second note is lower in pitch than the first, it is called downward motion. If the note stays the same, then the notes are said not to move. Many of the more complicated rules are concerned with the relative motions of two simultaneous intervals. The three kinds of motion are parallel motion (lines move in same direction), oblique motion (one line does not move), and contrary motion (lines move in opposite directions). In general, contrary motion is preferable to oblique motion, which is preferable to parallel motion. However, the consonance

rules suggest that parallel stepwise motion will preserve consonance over time.

Thus, the rule of contrary motion should not be overlooked in its importance.

6. *Vocal Ranges*: Another important consideration in the writing of tonal counterpoint is the vocal range of a human singer. In short, the limitations of the human voice should be represented in any given line of counterpoint. A vocal range $V \subseteq N$ (where N is the set of possible notes) is defined by a high note $h \in N$, a low note $l \in N$, where $h \geq l$. The range is the set of all chromatic notes between h and l which are members of the scale S . The traditional parts are soprano, mezzo soprano, alto, tenor, baritone and bass. The following diagram shows the low and high notes for each of the 6 basic vocal ranges.



Courtesy of the *New Harvard Dictionary of Music*

(<http://www.library.yale.edu/cataloging/music/vocalrg.htm>)

The different parts are thus constrained to choose their notes from only one of these vocal ranges.

7. *Part Crossing*: Crossing the lines of two parts is considered poor part-writing by Fux. David Huron rationalizes this observation on the basis of rules of music perception, once again suggesting that the rules are attempting to optimize the aesthetic sensibilities of the time. Thus, the range of valid notes is limited by the pitches of notes in adjacent voice parts, since any note that lies beyond will be a case of crossing parts.

8. *Parallel and Direct Perfect Intervals*: There is a final important concept in writing counterpoint, which is the relationship between the perfect intervals $P = \{P1, P4, P5, P8\}$, and relative motion. The aural properties of the perfect intervals make them a common point-of-exception in music theory. In particular, when two lines are moving in the same direction, the destination interval cannot be one of the perfect intervals. If the source interval is a perfect interval as well, this is called a “parallel perfect interval”. If the source interval is an imperfect interval ($x \in S-P$), this is called a “direct perfect interval”. Both direct and parallel perfect intervals are exceptions to the above vertical consonance table, and are no longer acceptable vertical consonances.

Advantages of the Problem Space

Fux’s approach is governed by a set of rules which vary in how strictly they must be followed. Assuming adherence to these rules, music written will not only be technically acceptable to the critical eye, but aesthetically pleasing to the ear. *The New Grove Dictionary of Music and Musicians* suggests reasons for the continuing popularity of Fux’s treatise in teaching the fundamentals of music composition: “Since the break with tradition that occurred around 1910, the custom of continuing to teach counterpoint in the Fuxian manner is generally justified by arguing that it is pedagogically necessary to discipline musical thought by means of exercises in ‘dead material’. (No other style can be codified to the same extent as can the technique of Palestrina.)” Additionally, it implicitly provides a formal definition of aesthetic beauty, for the later assessment of the success of such an endeavor. To a scientist in the field of algorithmically-generated music, this is an

appealing claim. Because Fux's rules outline a way to systematically generate tonal music, 16th century counterpoint has become a very popular testing ground for music composition algorithms.

Formalizing the Problem

To formalize the counterpoint problem, it is sufficient in this case to define both the problem space and a decision algorithm for whether or not a given solution satisfies the problem. The counterpoint problem is in a class of problems called constraint satisfaction problems (CSP, from Russell/Norvig). A constraint satisfaction problem (CSP) is a problem in which there are a set of constraints, and a candidate solution in the problem space is a solution if and only if it satisfies these constraints. In the case of the counterpoint problem, there is a cantus firmus against which all solutions are constrained. The solution space is bounded by all possible sets of $m-1$ strings of notes, each of length n , where n is the length of the cantus firmus and m is the number of lines in the piece, including the cantus firmus⁵. If the candidate solution satisfies all the constraints put forth as the basis of Fuxian counterpoint, then it is a solution to the problem. Otherwise it is not a solution. Pseudocode for this algorithm might look like the following:

```
function isASolution(candidate) :
    if(    isNotDiatonic(candidate))
        return false;
    else if(    badMelodicConsonance(candidate))
        return false;
    else if(    badHarmonicConsonance(candidate))
        return false;
```

⁵ Because the cantus firmus is already known, only $m-1$ lines must be considered.

```

        . . .
    else if(    parallelPerfectIntervals(candidate))
        return false;
    else return true;
end function;

```

This algorithm simply screens the candidate solution to make sure that it satisfies each of the constraints.

It is important to stress the difference between this decision algorithm, which determines if a solution is valid, and other algorithms that might generate such valid solutions. The above `isASolution` function is simply a verification algorithm, and cannot be used to generate candidate solutions. However, it might be possible to write an algorithm to find valid solutions that first generated all possible solutions in the problem space, and then ran the above decision algorithm to screen for which of those candidate solutions were valid solutions. Note that by this definition of the problem, there is no emphasis on finding more than one valid solution that satisfies the constraints. Once a single solution is found, then problem is solved. Thus, an algorithm that always finds the same valid solution technically satisfies the problem.

The Problem with Blind Search Algorithms

If one were to imagine the counterpoint problem as a search problem, in which the search space is all possible solutions, there are two simple approaches that would surely yield high-scoring results, both of which fall under the class of *generate-and-test* algorithms. The underlying idea is that the algorithm generates all possible solutions, and then tests these solutions to find which ones are valid solutions.

The first of these is the depth-first search. Consider a single note cantus firmus, which can be any of the 88 notes on a keyboard. Now consider a line of counterpoint, in

which there are 88 choices of note to juxtapose against the cantus note. If an additional note is added to the cantus firmus, an additional note must also be added to the counterpoint. This can be represented as a tree diagram in which each node is a possible successor note, with an empty start node. This would yield, for an n -note cantus firmus, 88^n possible solutions to search through. A simple depth-first traversal would provide all of these solutions, and would allow for finding valid solutions by way of a test function called on each leaf node to see whether it is a valid piece of counterpoint against the cantus firmus. This same technique could work for an m -voice piece of counterpoint, in which there are $m-1$ distinct melodies against the cantus. This could be represented as 88^{m-1} transitions-per-node, each of which is a set of $m-1$ potentially identical successor notes. Again, leaf nodes would be tested against Fux's rules, and as long as a valid solution exists, it will be found (since every element of the search space will be tested).

This solution poses many potential problems. First and foremost is the worst-case complexity of the algorithm, which is

$$O((88^{m-1})^{n+1} - 1) = O(88^{(n+1)*(m-1)}) = O(e^{(n+1)*(m-1)*\ln(88)})$$

If we have a 16-note cantus firmus and a 4-part texture, this yields $681472^{17}-1$ nodes to expand in the worst case. This would involve some $1.474e+99$ operations, optimistically assuming that to expand and evaluate any node takes only one operation. Thus in the worst case, with $1.0e+9$ operations per second, it will take until well after the sun becomes a supernova to find a solution. Furthermore, the fact that the only valid solutions are at leaf nodes diminishes the efficiency of this algorithm⁶. The space

⁶It should be noted here that the inefficiency could be considered to be insignificant. In a complete tree, the ratio of leaf nodes to total nodes is $(b^n):(b^{n+1}-1)$. It is effectively $b^n:b^{n+1}$, which is roughly $1/b$ of the nodes. In situations where the base $b = 2$, this inefficiency is more pronounced, where almost half of the expanded nodes are non-leaf nodes. In this particular situation, though, the base is 88^{m-1} . Given that

complexity of the algorithm will be $O(88^n) = O(n)$ in the worst case. However, while this solution has a desirable space complexity, the time complexity and inefficiency of the solution make it unacceptable.

A more efficient algorithm might be an iterative generate-and-test. In this algorithm, the candidate solution is represented as a number in base 88^{m-1} , where each radix represents a note in the piece. Starting with the number 0, the candidate is evaluated, and returned as a solution if it satisfies the problem's definition of a solution. If it is not a satisfactory solution, it is incremented and re-evaluated. The time complexity for this is $O((88^{m-1})^n) = O(88^{n(m-1)}) = O(e^{n(m-1) \ln(88)})$. This is a slightly better time complexity, and the space complexity is $O(n)$, since only one length n candidate solution needs to be stored in memory at any given time. Also, the inefficiency of the depth-first search is no longer an issue. Unfortunately, while slightly better in many regards, this solution is still of infeasible time complexity, and thus unacceptable.

The Heuristic Algorithm

While the complexity of the search space may be daunting, there is no reason that it must be approached by an algorithm which takes so little into account regarding the properties of the search space. The above definitions of depth-first search and generate-and-test are both in a class of search algorithms known as *blind* or *uninformed* searches (Russell and Norvig,). This class of algorithms provides a good starting point and

traditional counterpoint is written for between two and eight voices, this suggests that only between 1/88 and 1/40867559636992 of the notes are redundant. For many practical applications, > 98% efficiency is seen as quite efficient.

benchmark against which can compare other search techniques that may or may not be more suitable for the search space.

The other major class of search algorithms is the *heuristic* or *informed* search (Russell, Norvig), in which information about either the search space or the state of the search algorithm can be used to augment the process of finding a solution. Imagine that during the depth-first search, at each node expansion, instead of choosing nodes in some arbitrary order, a node is chosen based on its proximity to the current node, with the closest node expanded first, etc... This variation on the algorithm is motivated by the rule of stepwise motion. While the solution must still be checked at the leaf, the idea is that the solution will be found sooner because of the use of information about the partial solutions. Another variation might be to evaluate partial solutions, only expanding those nodes that satisfy the constraints for their segment of the solution. Provided that the validity of a complete solution depends on the validity of partial solutions, and that the number of valid partial solutions is sufficiently smaller than the total number of nodes, this variation should significantly reduce the average time it takes to find a valid solution.

These variations, while different in their own right, both hinge on the idea of a rule-of-thumb. This rule of thumb is called a *heuristic*, the etymology and history of which can be found in Russell and Norvig's *Artificial Intelligence: A Modern Approach*. They describe the word heuristic as “most often used as an adjective, referring to any technique that improves the average-case performance on a problem-solving task, but does not necessarily improve the worst-case performance.”⁷ (Russell/Norvig, p. 94) In addition to this definition, this paper treats heuristic as a noun, referring to a function

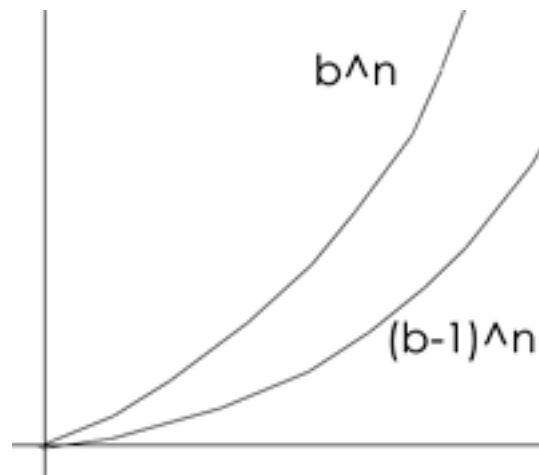
⁷ Analysis of average-case performance is beyond the scope of this paper, since it relies heavily on experimental results.

from a candidate solution to a value representing heuristic information for use in a search algorithm. One example of a “heuristic search algorithm” is called a *best-first search*, in which a heuristic is evaluated over all expandable nodes, giving each node a value for that heuristic. The node with the best score is expanded first, and the second best node is expanded second, and so forth. In the above example the heuristic might be $d-f(x)$, where $f(x)$ is some distance function of the nodes in the current node and the nodes in the next node, and d being the maximum possible distance returned by $f(x)$. Thus the closest nodes would be expanded before those that are further away. While there is no guarantee that this will produce better or faster results, experiments have shown that, when chosen carefully in a way that utilizes actual data about the problem, best-first searches often display consistent improvement over blind search techniques.

Another application of heuristics in the search process is called pruning, which effectively sets a lower bound threshold that is used to determine which nodes should be expanded. After each node is heuristically evaluated, its score is measured against a threshold, and if the score falls below the threshold it is removed from the set of nodes that can be expanded. The advantages of this technique involve what is called the *branching factor* of the search tree. The branching factor is the average number of nodes that can be expanded at each non-leaf node of the tree, and is central factor in the time and space complexity of many search algorithms. In the case of the search tree described above, there will be 88^{m-1} possible nodes to expand at each non-leaf node, so the branching factor will be 88^{m-1} . Now, considering the number of constraints and the strictness of the problem description, it is likely that very few of these nodes will be valid successors. However, they must still all be expanded, in the worst case. The idea behind

pruning is that the algorithm can rule out those nodes which are least likely to produce valid complete solutions, without expanding them and their children, and that by doing so, the branching factor is reduced.

It is extremely difficult to predict the extent to which pruning will affect the branching factor because the benefits depend largely on the probability that a node will be invalid, the value of the threshold, and the quality of the heuristic. Furthermore, if the threshold is too high or the heuristic too inaccurate, no valid solutions will be found, even though they might exist. However, in the counterpoint problem, the branching factor contributes the base b of the complexity, $O(b^n)$. Thus, if just a single node can be pruned from the tree at each non-leaf node without sacrificing the accuracy of the algorithm, the complexity will improve. Intuitively, the complexity of b^n is less than that of $(b-1)^n$, since there is no constant coefficient which can be multiplied by $(b-1)^n$ that will always bound b^n for all n .



More formally, the complexities of two algorithms can be compared as a ratio. Let C be the ratio of algorithm a to algorithm b , both functions of n . As n approaches infinity, C will do one of three things: approach 0, approach infinity, or equal some

constant value. If C equals some constant, then the two functions are in the same complexity class. If C approaches 0 or infinity, then the complexity of a is less-than the complexity of b , or greater-than the complexity of b , respectively. In this case, $C = b^n / (b-1)^n = (b/(b-1))^n$. Since $b > b-1$ and $b > 0$, $b/(b-1)$ will always be greater than 0. Therefore, as n approaches infinity, C approaches infinity. Therefore, the $O(b^n) > O((b-1)^n)$.

While analysis of the practical benefits of pruning in this problem space would require experimental results, it has been shown that pruning can offer significant benefits to applications of heuristic algorithms.

Heuristic Algorithms in the Counterpoint Problem

Before continuing on with an analysis of the various heuristic algorithms and techniques one might use to solve the counterpoint problem, it is necessary to refine the problem somewhat to incorporate the idea of a heuristic. It is possible to view the constraint-satisfaction problem proposed above as an optimization problem. Instead of giving each candidate solution a binary score of “good” or “bad”, they might be given a gradient score between 0.0 and 1.0. In this relaxed version of the problem, a 1.0 still satisfies most, if not all of the constraints, and a 0.0 satisfies few, if any of the constraints. This increased precision makes the use of heuristic algorithms feasible within the problem space, as well as allowing for a finer granularity with which to classify potential solutions the counterpoint problem. Such a heuristic should give consistently high marks to candidates which constrain to the rules of Fuxian counterpoint, and high marks to the contrapuntal music of Palestrina, on which Fux’s treatise is based.

The more complicated problem lies in finding such a heuristic that can effectively provide this gradation. Trivially, it could be a function that increments the score by $1/c$ for every constraint satisfied. However, there is a problem inherent in this, which is that the above constraints are not wholly separable as unrelated factors. In fact, conformance to some constraints might make it impossible not to constrain to another constraint. For instance, pieces with mostly of oblique and contrary motion will have fewer parallel perfect intervals. As nice as it might be to imagine these constraints as discrete bundles, there are countless tiny overlaps which might skew the score in the favor of one constraint over another, as the benefits of satisfying a certain constraint might imply the satisfaction of several others. As a result, aspects of a good solution, such as harmonic consonance, might be lost. The intuitive fix for this imbalance is to provide each constraint's score with a coefficient that weights it more heavily if it is not equally represented. The question then becomes how much weight to place on a given constraint. If that knowledge were available, however, the overlap would no longer be a mystery, and it would possible to write a new set of constraints that do not overlap.

This was the decision reached by David Cope, David Eisenstat, and myself⁸: a set of relaxed, yet separable constraints could be used as a heuristic function, where each constraint accounted for $1/c$ of the final score. We performed preliminary experiments, and found that a genetic algorithm using relaxed and separable constraints as its heuristic found solutions of high quality faster than using either the weighted or un-weighted overlapping constraints. The constraints settled on were as follows:

⁸ This was discussed during the 2003 Workshop on the Algorithmic Composition of Music at University of California, Santa Cruz.

1. *Stepwise Motion*

This is a relaxed version of the melodic consonance constraint. It scores based on how closely the piece tends towards consonant stepwise motion. In our implementation, the intervals are first placed into equivalence classes and then scored based on which equivalence class they are in. The classes are:

Intervals	{0,1,2}	{3,4}	{5}	{6}	{7}	{8}	{9}	{10}
Score	10	7	6	5	4	3	2	1

Intervals that are not horizontally consonant or are greater than 10 half-steps are given a score of 0. This rewards for stepwise motion, but does not punish greatly for a leap. Also, the filter on horizontal consonance removes the balancing conflict of stepwise motion versus horizontal consonance by folding them into the same constraint.

2. *Consonance*

This is a relaxed version of the consonance and suspension constraints. First of all, it checks the intervals between all pairs of notes that sound simultaneously, and scores them based on the rules of harmonic consonance. If an interval fails the harmonic consonance test, however, it will still receive between 25-75% of the maximum score if the interval resolves from dissonance to consonance by stepwise motion. This is only a rough estimation of a suspension, and does not fully incorporate all the rules of suspensions. However, it is a simple heuristic that I feel captures the central purpose of the suspension, which is that dissonance is legal so

long as it resolves by step. It is important to acknowledge that struck dissonance becomes legal in this implementation. A more authentic implementation might be to check for struck dissonance, and to only apply the dissonance exception to those intervals which are truly suspensions.

3. *Contrary Motion*

Inclusion of contrary motion as a primary constraint is largely to safeguard against candidates which ensure stepwise motion and consonance by simply moving all lines in parallel. It encapsulates the tendency towards contrary motion, which in turn avoids parallel perfect intervals and encourages proper resolution of leaps.

It is more effective to implement the rest of the knowledge about the solution space through simple filters. For a given line, simply assign a vocal range and a scale, and do not allow any chromatic/out-of-range notes in that line to be chosen in the first place. This way, by limiting the data structure, a large number of nodes can be pruned away. This covers the constraints of vocal range, and diatonicism. If part-crossing should occur, it is possible to re-order the notes such that the parts no longer cross, by assigning notes of higher pitch to the line with the higher voice range. If two lines have the same voice range, one of the lines can be arbitrarily assigned to include the notes of higher pitch. Through the three relaxed constraints and limitations on the data structure, we found a heuristic that can map closely to the original constraints, with little or no overlap between the different constraints.

Comparison of Algorithms

What follows is a list of some common search algorithms, and how they might be used to solve the counterpoint problem.

The first algorithm is the *depth-first search*, which was discussed earlier. A stack can be used to implement a depth-first search. To expand a node of the tree, each of the successor nodes are added to the stack. The node expanded is always that node at the top of the stack. If the top element of the stack is of length n , where n is the length of the cantus firmus, it is compared against the constraints and no successor nodes are expanded⁹. If this leaf node is a valid solution, it is returned as such and the search is complete. While this is treated as a blind search, it is still possible to incorporate a random element to it, by having it place successor nodes onto the stack in pseudo-random order, causing sibling nodes to be evaluated in any order. This might remove a bias towards certain kinds of solutions that would emerge as a by-product of some other ordering. For instance, if the nodes are expanded in order from lowest pitch to highest pitch, then the algorithm might always find lower-pitch solutions when solutions with higher pitch exist.

A *best-first search* could be implemented using the same algorithm, with the alteration that the nodes would be added to the stack in order of how well they scored, with the lowest-scoring nodes being added to the stack first. This would mean that the nodes with the best score would be expanded first, and because of the nature of this stack-based implementation, the best children of the best nodes would be expanded before

⁹ This could be seen as a depth-limited depth-first search, in which the search depth is limited to some constant number. However, since there is only one depth that contains valid solution nodes, this has a different motivation than other depth-limited searches, which are often just bounding an infeasible or infinite time-complexity.

any of the best node's siblings. Another, often more effective, implementation is to use a priority queue as the underlying data structure for the best-first search. This way, when nodes are added to the priority queue, they have some absolute ranking relative to all other nodes expanded so far, and the globally "best" node will always be expanded first. This is an implementation of *back-jumping*, a technique where if the score of a given branch begins to decrease beyond the score of its ancestors, it is abandoned and the algorithm "jumps back" to some point in the tree at which the score was still increasing.

Best-first search is known as a *hill-climbing algorithm*. Hill-climbing means that at any potential branch, the algorithm selects which to take based on whichever has a higher heuristic evaluation. Going back to the underlying metaphor, the algorithm is similar to a blind man climbing a hill by always walking in the direction of steepest incline. The problem with this algorithm, though, is that the man cannot tell if he has reached the absolute top of the hill, or some local peak. In the case of the counterpoint problem, these local maxima are simply evaluated against a threshold. If a maximum is sufficiently high, then it is a valid solution, though it may not be the best solution. By pushing the threshold higher, the solutions will be guaranteed to be of higher scores. However, if the threshold is too high, it is possible that no valid solutions will be found.

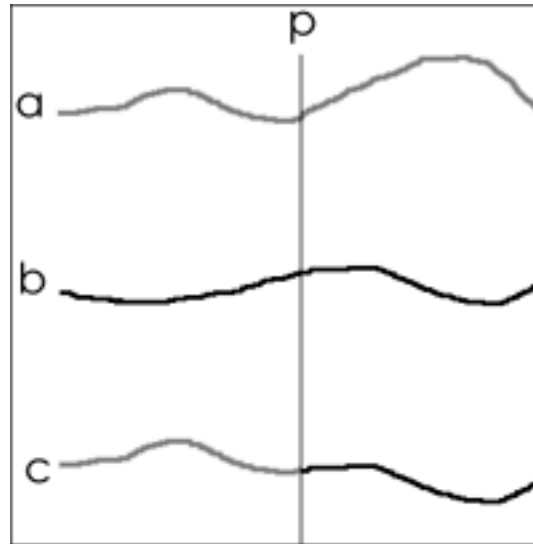
There is another hill-climbing algorithm worth looking at, which differs from the above best-first search primarily in its choice of data structure. (Here, hill-climbing indicates that the algorithm is inspired by the same underlying metaphor.) Instead of representing the search space as a tree structure, it is possible to represent it as a graph structure, with each node being an n -length solution, where n is the length of the cantus firmus. The transitions could be defined in any number of ways, but, in this

implementation, consider there to be a bidirectional “neighbor” transition between any two nodes that differ in the pitch of a single note. To climb the search space, choose an arbitrary node in the graph. Evaluate all neighbors of the node using some heuristic, and place these on a priority queue with priority represented as their score. Place the now-expanded node in a list of expanded nodes, and proceed to expand the node at the top of the priority queue. Repeat this expansion process, making sure not to add any nodes to the priority queue that have already expanded (that are in the expanded nodes list). Once a node is found that has a score at least as high as the threshold, it is returned as a solution and the search has ended. Because this is a graph-traversal algorithm, there is the problem of cycles to contend with, which is why the algorithm must keep track of which nodes have already been expanded. Unfortunately, this hurts the space complexity greatly, since in the worst case all the nodes in the search space would be stored in memory at the same time.

Another very popular algorithm that makes use of a graph structure is called the *genetic algorithm* (GA), which is based on a model of genetic evolution, aiming to generate solutions via a crude approximation of “survival of the fittest”¹⁰. The set of nodes is the same as in the above hill-climbing algorithm, but the transitions are much more unusual. There are two transition functions, “mutation” and “crossover”. In the counterpoint problem, let two nodes be bi-directionally connected by mutation if they differ in the pitch of a single note (like the neighbor function in the hill-climbing example. Crossover is more complicated, transitioning from two nodes to one. Let $\text{crossover}(a,b) = c$ if and only if there is some point p in c such that the notes before p in a and c are the same

¹⁰ The history and origins of the genetic algorithm are beyond the scope of this paper, and can be found in Melanie Mitchell’s “An Introduction to Genetic Algorithms”.

pitch, and the notes at and after p in b and c are the same pitch. The following diagram shows a crossover relationship between a , b , and c at p .



Unlike the previous three algorithms, genetic algorithms do not use a stack or queue to implement their algorithm. Rather, GA's work with a population, a set of candidate solutions. A simple implementation of a GA involves the generation of an initial population, usually by random selection of nodes from the search space. Then a "selection" function is run on the population, returning an individual (GA term for a candidate solution). The selection is based on the score of the individual, relative to the rest of the population, with some probabilistic bias to select individuals of higher score. The selection function is run again, returning another individual from the population. A point is chosen at random, and the crossover of the two individuals is generated. Then, based on a probabilistic constant of mutation, a mutation is performed on the individual and the individual is placed into a new population. The procedure is performed repeatedly, usually until the new population is of size equal to the original population. At this point, the original population is then destroyed and the new population is used to

yield another population. Each cycle is referred to as a generation, and after some number of generations the individual in the population with the highest score is returned as the solution. Often the GA will cycle through generations until a threshold score is met.

GA's pose many problems, algorithmically. Firstly, they only look at a very small percentage of the search space, so there is no guarantee that they will always find a solution, even if one exists. Second, the ideal choice of the evaluation heuristic¹¹, selection function, population size, initial population, and number of generations are all unknown at the time of GA experimental setup. This leads to multi-factor trial-and-error optimization of the experimental setup. It might be possible to use a hill-climbing algorithm to optimize some of the variables of the search space, but the selection and fitness functions would be much more difficult in this respect. Despite these problems, GA's are still widely used in AI research, and a wide array of selection, mutation, and crossover schemes have been devised and justified via experimental results¹².

Conclusion

I introduced and formalized the problem of generating 16th-century Fuxian counterpoint as a CSP with both strict and relaxed constraints, motivating the constraints on the basis of Fux's *Gradus ad Parnassum*. I explained the process of finding a heuristic with orthogonal constraints, again motivated by Fux's work and the music of Palestrina. The algorithms that I presented were depth-first search, iterative search, hill-climbing, and the genetic algorithm. I also explained the techniques of pruning and back-

¹¹ Known by GA researchers as the "fitness function", because it evaluates the "fitness" of an individual. In this paradigm, the environment in which the individual is evaluated is the cantus firmus line.

¹² See Melanie Mitchell's "An Introduction to Genetic Algorithms" for details on these different approaches.

jumping and the potential implementation of them. In my presentations of the various algorithms, I showed the time and space complexities where relevant, emphasizing the relative advantages of heuristics in the search process. Future work might involve experimentation and comparison of the methods outlined in this paper, assessing quality of solutions as well as average-case time complexity. Alternatively, these same ideas might be applied to a more complex musical problem, such as Bach chorales or similar constrained formal musical styles.

Bibliography

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall 2nd edition, 2002.

Cormen, Thomas H. et al. *Introduction to Algorithms*. MIT Press, 1990.

Huron, David, *Tone and Voice: A Derivation of the Rules of Voice-leading from Perceptual Principles*, Music Perception, Vol. 19, No. 1 (2001) pp. 1-64.
<http://www.music-cog.ohio-state.edu/Huron/Publications/huron.voice.leading.html>

Mitchell, Melanie. *An Introduction to Genetic Algorithms*. MIT Press/Bradford Books. 1996. Cambridge, MA.

Todd, Peter M. & Werner, Gregory M. “Frankensteinian Methods for Evolutionary Music Composition”

Horner, Andrew & Goldberg, David E. “Genetic Algorithms and Computer-Assisted Music Composition”. University of Illinois at Urbana-Champaign.

Gartland-Jones, Andrew & Copley, Peter. “The Suitability of Genetic Algorithms for Musical Composition”. *Contemporary Music Review*. Vol. 22, No. 3. 2003.

Polito, John & Daida, Jason M. & Bersano-Begey, Tommaso F. “Musica ex Machina: Composing 16th-Century Counterpoint with Genetic Programming and Symbiosis”. *Evolutionary Programming Vi: Proceedings of the Sixth Annual Conference on Evolutionary Programming*. 1997.

Fux, Johannes J. *Gradus ad Parnassum*, trans. Alfred Mann. *The Study of Counterpoint*. New York. 1971. W. W. Norton & Company, Inc.

Conklin, Darrell. “Music Generation from Statistical Models”. *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*. 2003.

Pachet, Francois & Roy, Pierre. “Musical Harmonization with Constraints: A Survey”. *Constraints Journal*. 2001. Kluwer Publisher.

Schottstaedt, B. “Automatic Species Counterpoint”. *Current directions in Computer Music Research*. 1989.

The New Harvard Dictionary of Music

The New Grove Dictionary of Music and Musicians ed. S. Sadie and J. Tyrrell London. 2001. Macmillan.

Appendix

Lisp Implementation of the Revised Heuristic Function

```
(defvar *consonant-intervals*)
(setq *consonant-intervals* #(t t t t t nil nil t t t nil nil))

(defun is-consonant-step (interval)
  (aref *consonant-intervals* interval))

(defun score-stepwise-motion (pitches)
  (let ((score 0)
        (this-pitch nil)
        (next-pitch (aref pitches 0))
        (interval nil))
    (loop for index from 1 to (1- (length pitches))
          do (setf this-pitch next-pitch)
              do (setf next-pitch (aref pitches index))
              do (setf interval (abs (- next-pitch this-pitch)))

              if (and (<= interval 10) (is-consonant-step interval))
                  do (cond ((<= 0 interval 2) (setf interval 0))
                          ((<= 3 interval 4) (setf interval 3)))
                  if (and (<= interval 10) (is-consonant-step interval))
                      do (incf score (- 10 interval)))
    score))

(defvar *consonances*)
(setq *consonances* #(t nil nil t t nil nil t t t nil nil))

(defun is-consonance (pitch-one pitch-two)
  (if (< pitch-one pitch-two)
      (aref *consonances* (mod (- pitch-two pitch-one) 12))
      (aref *consonances* (mod (- pitch-one pitch-two) 12))))

(defun is-step (pitch-one pitch-two)
  (or ;;;; (equal (abs (- pitch-one pitch-two)) 0)
        (equal (abs (- pitch-one pitch-two)) 1)
        (equal (abs (- pitch-one pitch-two)) 2)))

(defun score-consonance (lower-pitches upper-pitches)
  (let ((first-lower-pitch)
        (second-lower-pitch (aref lower-pitches 0))
        (first-upper-pitch)
        (second-upper-pitch (aref upper-pitches 0))
        (score 0))
    (loop for i from 0 to (1- (length lower-pitches))
          do (setf first-lower-pitch second-lower-pitch)
              do (setf second-lower-pitch (aref lower-pitches i))
              do (setf first-upper-pitch second-upper-pitch)
              do (setf second-upper-pitch (aref upper-pitches i))
              if (is-consonance second-lower-pitch
                                second-upper-pitch)
                  do (incf score 4)

              else do (if (is-consonance first-lower-pitch second-upper-pitch)
                          (progn (incf score)
                                 (when (is-step first-lower-pitch second-lower-pitch)
                                   (incf score))
                                 (when (is-step first-upper-pitch second-upper-pitch)
                                   (incf score))))))
    score))

(defun direction (origin destination)
  (cond ((> 0 (- destination origin)) -1)
        ((< 0 (- destination origin)) 1)
        (t 0)))
```

```

(defun score-contrary-motion (lower-pitches upper-pitches)
  (let ((score 0)
        (first-lower-pitch)
        (first-upper-pitch)
        (second-lower-pitch (aref lower-pitches 0))
        (second-upper-pitch (aref upper-pitches 0)))
    (loop for index from 1 to (1- (length lower-pitches))
          do (setf first-lower-pitch second-lower-pitch)
              do (setf first-upper-pitch second-upper-pitch)
              do (setf second-lower-pitch (aref lower-pitches index))
              do (setf second-upper-pitch (aref upper-pitches index))
              unless (and (equal (direction first-lower-pitch second-lower-pitch)
                                (direction first-upper-pitch second-upper-pitch))
                        (not (equal (direction first-lower-pitch second-lower-pitch) 0)))
                    do (incf score))
    score))

```