ABSTRACT

The number of solutions involved in many algorithmic
composition problems is too large to be tractable with-
out simplification. Given this, it is critical that composi-
tion algorithms be able to move through different levels
of abstraction while maintaining a well-organized solu-
tion space. In this paper we present the following con-
tributions: (1) extended formalizations and proofs needed
to implement the chord spaces defined by Tymoczko [11]
and Callender et al. [2], (2) a generalized framework for
moving between levels of abstraction using quotient spaces
that can easily be integrated with existing algorithmic com-
position algorithms, and (3) an application of both to voice-
leading assignment.

1. INTRODUCTION

A major problem in the area of algorithmic composition
is the need for organized and easily traversable sets of
solutions, also referred to as solution spaces, which are
tractable in terms of both runtime and memory require-
ments. Many music-theoretic ideas are also not formal-
ized to the degree necessary to ensure correct implement-
ation of algorithms and accompanying data structures.
In this paper we address both of these problems by pre-
senting a general framework for organizing and traversing
harmony-related solution spaces. Our work builds on that
of Tymoczko [11] and Callender et al. [2], adding an addi-
tional layer of formalization necessary to create a gen-
eralized and extensible implementation. We then apply
our framework to the task of voice-leading assignment,
a common problem in music composition.

Consider the following situation: given a sequence of
chords intended for a soprano, tenor, and baritone, re-
write the same chords for three tenors while factoring in
additional constraints about each performer – perhaps one
of the performers is a beginner, requiring smooth voice-
leadings. This paper presents a set of algorithms and sup-
porting proofs to automate algorithmic composition and
arranging tasks such as above. Our approach utilizes two
important concepts: chord spaces [2][11] and musical pred-
icates.

A task such as outlined above will be referred to as a
voice-leading assignment. Our goal is to construct a
performable series of chords from incomplete information
about those chords, such as the pitch classes involved in
each. To assign a C-major triad to three voices, a spe-
cific C, E, and G must be chosen. This involves choosing
octaves for each pitch class and determining which pitch
should be assigned to each voice.

We use the term concrete chord to refer to chords with
no room for additional interpretation and the term ab-
stract chord when choices still exist. Voice-leading as-
signment is the process of turning abstract chords into
concrete chords. This is also representative of a larger
category of tasks in composition: moving between differ-
ent levels of abstraction in music. Particularly for large
problems, the solution spaces must be well-structured and
efficient to traverse.

Our approach to voice-leading assignment uses a type
of quotient space called a chord space [2][11]. Chord
spaces are a way to organize chords in musically mean-
ningful ways and provide a convenient, intermediate level
of organization between abstract and concrete chords. For
example, one such chord space groups chords based on
pitch class content, providing a useful level of abstraction
for voice-leading assignment. We use this space to turn a
sequence of abstract chords represented in terms of pitch
classes into a sequence of concrete chords. When finished,
each pitch class in each chord is assigned an octave and a
particular voice.

There are many other chord spaces that relate chords
in different ways. These can also be used with our algo-

rithm to perform variations on the voice-leading assign-
ment task, allowing the algorithm a greater degree of con-
trol over what musical features are generated. By simply
changing the chord space, our voice-leading assignment
algorithm can be generalized to make choices about pitch
classes and octaves.

Data-driven algorithms such as Markov chains have
been used to learn voice-leading behavior from collections
of examples [3][12]. Markov chains suffer from state ex-
plosion when addressing low-level features in music while
still capturing structure. Variable-length Markov models
[1] and probabilistic suffix trees [10] attempt to address
this problem, but are still prone to the same problem with
the large alphabets involved in musical problems. Chord
spaces [2][11] can help with this, since they allow gener-
ative problems to be broken into multiple steps, each at a
different level of abstraction.

Chord spaces, however, present a number of repre-
sentational issues. Being a type of quotient space, chord spaces are specified with equivalence relations. When combining two equivalence relations to produce a new one or when applying more than one equivalence relation to a set to produce a quotient space, it is not always easy to preserve properties like transitivity or to find an efficient representation for the relation or the quotient space. We present additional formalizations and algorithms to address these types of representational issues for chord spaces.

Finally, algorithms that perform well at large-scale compositional tasks, such as those presented in the work of Ebcioglu [6] and Cope [4, 5], tend to have application-specific aspects to the implementations that make generalization and reuse difficult. In this paper, we aim to achieve a general result that has application to any algorithmic composition problem that requires moving between multiple levels of abstraction to find a solution. We show one such application to voice-leading assignment. Our approach utilizes chord spaces to organize the solution space and controls voice-leading behavior (the exact up/down movement in each voice from one chord to the next) through the use of musical predicates, which can easily be constructed to follow common music-theoretic principles, such as prohibiting voice crossing and parallel motion.

2. CHORD SPACES

We represent chords as vectors of integers, where every integer represents a pitch. Middle C, (C,4), is the integer 60. (C#,4) is 61, and so on. Vectors are written as \( \vec{v} \) to refer to a single vector and \( \langle x_1, x_2, \ldots, x_n \rangle \) to show all of the vector’s elements. A chord is represented as a vector of integers. The set of all possible chords of length \( n \) (having \( n \) voices) is \( Z^n \). \([a, b]^n\) is the set of all vectors \( \langle x_1, \ldots, x_n \rangle \) where \( a \leq x_i \leq b \). We write sequences of vectors using the notation \( [\vec{v}_1, \ldots, \vec{v}_n] \). The notation \( 1^n \) denotes the vector of length \( n \) whose elements are all 1.

A quotient space formed by applying relation \( R \) to set \( S \) is denoted \( S/R \). A chord space is a quotient space formed from \( Z^n \) and an equivalence relation on chords (vectors). The quotient space formed from \( Z^n \) and an equivalence relation \( R \) is referred to as \( R \)-space. Two elements of the same equivalence class under \( R \) are said to be \( R \)-equivalent. The notation \( E(x, S/R) \) denotes \( x \)'s equivalence class under \( S/R \) which is \( \{ y \in S \mid y \sim_R x \} \).

Relations that form musically-meaningful partitions of \( Z^n \) are useful for automated composition tasks. One way to construct musically-meaningful equivalence relations is to exploit existing concepts in music theory, such as the ideas of pitch class and transposition. Tymoczko and Callender et al. introduce several such relations on chords, each based on some concept in music theory [11][2]. Each relation is defined over vectors in \( \mathbb{R}^n \), which uses the same mapping of whole numbers to pitches but also allows for microtones (a pitch of 60.5 would be a microtone just above middle C). All of the relations are applicable to our discrete pitch space, \( \mathbb{Z}^n \), as well. We make use of three of these equivalence relations:

- **Octave equivalence, \( O \).** Chords belong to the same equivalence class if they have the same vectors of pitch classes: \( \vec{v} \sim_O \vec{v} + 12i, i \in \mathbb{Z} \). For example, \( \langle 0, 4, 7 \rangle \) and \( \langle 12, 4, 7 \rangle \) are \( O \)-equivalent; they are both C-major triads where the voices have the pitch classes C, E, and G respectively.

- **Permutation equivalence, \( P \).** Chords with the same multiset of pitches belong to the same equivalence class under this relation. \( P \) can be defined using the symmetric group of order \( n \), \( S_n \) (the set of all permutation functions for \( n \) elements): \( \vec{v} \sim_P \sigma(\vec{v}), \sigma \in S_n \). For example, \( \langle 0, 4, 7 \rangle \) and \( \langle 4, 0, 7 \rangle \) are \( P \)-equivalent.

- **Transposition equivalence, \( T \).** Chords with the same intervallic content belong to the same equivalence class. For example, \( \langle 0, 4, 7 \rangle \) and \( \langle 1, 5, 8 \rangle \) are \( T \)-equivalent. The relation is defined in [2] as \( \vec{v} \sim_T \vec{v} + e1^n, e \in \mathbb{R} \), but we further constrain the definition by requiring \( e \in \mathbb{Z} \) for the remainder of this paper to be consistent with our discrete interpretation of pitches.

The \( O \), \( P \), and \( T \) relations can be used individually or combined to produce additional equivalence relations. Two equivalence relations, \( R_1 \) and \( R_2 \), can be combined to make a new equivalence relation using the join operation, \( R_1 \vee R_2 \) [8]. We will use the notation \( R^+ \) to denote the transitive closure of relation \( R \). For two equivalence relations, \( R_1 \) and \( R_2 \), where \( R_1 \cap R_2 \) is the composition of the two relations:

\[
R_1 \vee R_2 = (R_1 \cdot R_2 \cup R_2 \cdot R_1)^+ \tag{1}
\]

The join operation is commutative, such that \( R_1 \vee R_2 = R_2 \vee R_1 \). For simplicity, we will abbreviate \( R_1 \vee R_2 \) as simply \( R_1 R_2 \). For two points \( x, y \in S \):

\[
x \sim_{R_1 R_2} y \iff \exists z \in S, x \sim_{R_1} z \sim_{R_2} y \tag{2}
\]

The join operation can be used to combine the \( OPT \) relations to produce four other equivalence relations described in [2][11]: \( OP \), \( OT \), \( PT \), and \( OPT \):

- **Octave and Transposition equivalence, \( OT \).** \( \vec{v} \sim_{OT} \vec{v} + 12i + c1^n, i \in \mathbb{Z}^n, c \in \mathbb{Z} \) (or \( c \in \mathbb{R} \) for microtonal systems). Chords in the same equivalence class have the same intervallic structure when represented as vectors of pitch classes. For example, \( \langle 0, 4, 7 \rangle \sim_{OT} \langle 13, 5, 8 \rangle \).

- **Octave and Permutation equivalence, \( OP \).** \( \vec{v} \sim_{OP} \sigma(\vec{v}) + 12i, i \in \mathbb{Z}^n, \sigma \in S_n \). Chords in the same equivalence class have the same multiset of pitch classes. For \( n = 3 \) voices, \( OP \)-space contains an equivalence class for all C-major triads, another for all C-minor triads, and so on.
Permutation and Transposition equivalence, \( PT, \ \tilde{v} \sim_{PT} \sigma(\tilde{v} + c1^n), \sigma \in \mathbb{S}_n, c \in \mathbb{Z} \) (or \( c \in \mathbb{R} \) for microtonal systems). Chords in the same equivalence class share the same intervallic structure of their multisets of pitches. For example:

\((0, 4, 7) \sim_{PT} (5, 1, 8)\).

Octave, Permutation, and Transposition equivalence, \( OPT, \ \tilde{v} \sim_{OP} \sigma(\tilde{v} + 12\tilde{z} + c1^n), \tilde{z} \in \mathbb{Z}^n, \sigma \in \mathbb{S}_n, c \in \mathbb{Z} \). Chords in the same equivalence class have the same intervallic structure of their multisets of pitch classes, capturing the notion of chord quality. For example, \((0, 4, 7) \sim_{OPT} (0, 3, 8)\), where \((0, 4, 7)\) is a C-major triad and \((0, 3, 8)\) is an A-flat-major triad. This can be seen as follows:

\((0, 4, 7) \sim_{OPT} (12, 4, 7) \sim_T (8, 0, 3) \sim_P (0, 3, 8)\)

2.1. Normalizations and Representative Subsets

To construct and use a chord space under relation \( R \), it must be possible to test the \( R \)-equivalence of two chords. Points \( a \) and \( b \) are related under \( R \) if \((a, b) \in R \). However, if \( R \) is large (possibly infinite), such a space can be computationally intractable. Indeed, because \( \mathbb{Z}^n \) is infinite, all of the spaces formed by \( O, P \), and \( T \) are also infinite.

To address this problem, we take the following computational approach: for a set \( S \), relation \( R \), and quotient space \( S/R \), rather than enumerate the entire equivalence class of an element \( s \in S \), we compute a representative point of that equivalence class. We refer to the set of all representative points as the representative subset of \( S/R \), which we denote by \( S_R \). If a function \( f : S \rightarrow S_R \) has the property that every point in \( S \) is \( R \)-equivalent to exactly one point in \( S_R \), it is called a normalization. More formally:

**Definition 1.** \( S_R \subseteq S \) is a representative subset for \( S/R \) if and only if \( \forall x, y \in S \), there is only one \( z \in S_R \) such that \( x \sim_R z \).

**Definition 2.** \( f \) is a normalization for the quotient space \( S/R \) whenever \( \forall x, y \in S, f(x) = f(y) \iff x \sim_R y \).

For example, a normalization for \( O \), denoted as \( normO \), is straightforward:

**Algorithm 1.**

\[ normO((x_1, \ldots, x_n)) = (x_1 \mod 12, \ldots, x_n \mod 12) \]

If \( normO(x) = normO(y) \) we know that \( x \) and \( y \) belong to the same \( O \)-equivalence class.

A normalization for \( P \) is similarly straightforward:

**Algorithm 2.** \( normP(\tilde{v}) = \text{sort}(\tilde{v}) \)

where \( \text{sort} : \mathbb{Z}^n \rightarrow \mathbb{Z}^n \) arranges a vector's elements in ascending order. Thus if \( normP(x) = normP(y) \) we know that \( x \) and \( y \) belong to the same \( P \)-equivalence class.

Proofs that both \( normO \) and \( normP \) are normalizations for their respective spaces are trivial and are omitted.

Consider now \( OP \)-equivalence. A suitable normalization algorithm can be defined as:

**Algorithm 3.** \( \text{normOP}(\tilde{v}) = \text{normP}(\text{normO}(\tilde{v})) \)

In other words, \( \text{normOP} \) is the composition of the normalizations for \( P \)-equivalence and \( O \)-equivalence; i.e. \( \text{normOP} = \text{normP} \cdot \text{normO} \). Also note that \( \text{normOP} \) returns chords falling within:

\[ S_{OP} = \{ (x_1, \ldots, x_n) \in [0, 11]^n, x_1 \leq x_2 \leq \cdots \leq x_n \} \]

For example, the chords: \((0, 16, 7)\) and \((12, 7, 4)\) are \( OP \)-equivalent because:

\[ \text{normOP}((0, 16, 7)) = \langle 4, 0, 7 \rangle = \text{normOP}((12, 7, 4)) \]

In this case, the \( OP \)-equivalence class is that of C-major triads: chords containing the pitch classes C, E, and G.

That \( \text{normOP} \) can be defined as the composition of the normalizations for \( O \) and \( P \) is a special case. Given normalizations \( f_1 \) and \( f_2 \) for relations \( R_1 \) and \( R_2 \) respectively, there is no guarantee that \( f_1 \) and \( f_2 \) can be simply composed to create a proper normalization for the new equivalence relation.

Indeed, even the order of composition is important. If we were to reverse the order of composition in the definition of \( \text{normOP} \), i.e. apply the normalization for \( P \)-space first (sort) and then the normalization for \( O \)-space (take all pitches \( \mod 12 \)), points that should be \( OP \)-equivalent are mapped differently, thereby breaking one of the requirements for a representative subset (that all \( OP \)-equivalent points must map to the same representative point). For example, recall from above that \((0, 16, 7)\) and \((12, 7, 4)\) are \( OP \)-equivalent. But if we reverse the order of composition, let’s call it \( \text{normOP}' \), we have \( \text{normOP}'((0, 16, 7)) = (0, 7, 4) \), whereas \( \text{normOP}'((12, 7, 4)) = (4, 7, 0) \).

Equivalence relations can have more than one normalization, and different normalizations may be needed under different circumstances. This is easily seen for \( T \)-equivalence.

**Theorem 1.** Let \( F = \{ f_1, \ldots, f_n \} \) be the set of functions \( f_i = (x_1, \ldots, x_n) = (x_1 - x_i, \ldots, x_n - x_i) \). An algorithm \( A : \mathbb{Z}^n \rightarrow \mathbb{Z}^n \) is a normalization for \( \mathbb{Z}^n \) iff, for all \( \bar{x} \in \mathbb{Z}^n \), it applies the same \( f_i \) to all members of \( \bar{x} \)'s equivalence class, \( E(\bar{x}, \mathbb{Z}^n/T) \).

**Proof.** Recall that \( \bar{x} \sim_T \bar{y} \iff \exists c \in \mathbb{Z}, \bar{x} = \bar{y} + c1^n \). Two chords are \( T \)-equivalent if they have the same intervallic structure.

- Let \( \bar{x} = (x_1, \ldots, x_n), \bar{y} = (y_1, \ldots, y_n) \).
- Let \( \bar{x}' = (x_1 - x_i, \ldots, x_n - x_i), \bar{y}' = (y_1 - y_i, \ldots, y_n - y_i) \) for some \( i \in \mathbb{Z} \).
- If \( \bar{x} \) and \( \bar{y} \) have the same intervallic structure (the definition of \( T \)-equivalence), then \( \bar{x}' \) and \( \bar{y}' \) will be equal and the \( i \)-th element of both \( \bar{x}' \) and \( \bar{y}' \) will be 0. We therefore have that \( c = x_i - y_i \) and \( \bar{x} \sim_T \bar{x}' \sim_T \bar{y}' \).
- If \( \bar{x} \) and \( \bar{y} \) are not \( T \)-equivalent, then the intervallic structures are different and \( \bar{x}' \neq \bar{y}' \).
Therefore, $A(\bar{x}) = A(\bar{y}) \iff \bar{x} \sim_T \bar{y}$.

Corollary 1. The following function:

$$\text{normT}(\langle x_1, \ldots, x_n \rangle) = \langle x_1 - x_1, \ldots, x_n - x_1 \rangle$$

which subtracts the first element of a vector from all other elements, is a normalization for $\mathbb{Z}^n/T$.

The order-of-composition problem also exists for $OT$-equivalence. The normalizations for $O$ and $T$ can be composed in one order to produce a proper normalization for $OT$, but not the other. This can be summarized as a more generalized theorem:

Theorem 2. If relations $R_1$ and $R_2$ have normalizations $f_1$ and $f_2$, respectively, with ranges $S_1$ and $S_2$, respectively, such that $S_1 \cap S_2 \neq \emptyset$, then if $f_3 = f_1 \cdot f_2$ has the range $S_3 = S_1 \cap S_2$, then $f_3$ is a normalization for $R_1 \lor R_2$.

Proof. From definition\[1] $S_1$ and $S_2$ are representative subsets for $R_1$ and $R_2$ respectively. Every point in a representative subset for some relation $R$ belongs to a different $R$-equivalence class. When applying additional equivalence relations to a set, the number of equivalence classes can never increase. If an element is in $S_1 - S_2$, it must be $R_2$-equivalent to an element in $S_1 \cap S_2$. Similarly, elements in $S_2 - S_1$ must be $R_1$-equivalent to elements in $S_1 \cap S_2$. $S_3$ is therefore the only set of points which are not $R_1R_2$-equivalent to each other. Therefore, every point outside of $S_1 \cap S_2$ will map to only one point in $S_3$, meeting the requirements for $f_3$ to be a normalization and $S_3$ to be a representative subset for $R_1R_2$.

Corollary 2. The function $\text{normOT} = \text{normO} \cdot \text{normT}$ is a normalization for $\mathbb{Z}^n/O\cdot T$.

Proof. Recall that $\bar{x} \sim_{OT} \bar{y} \iff \bar{x} = \bar{y} + 12\bar{T} + c\bar{I}$.

- $\text{normT}$’s range is $S_{T1} = \{(0, x_2, \ldots, x_n) \in \mathbb{Z}^n\}$.
- $\text{normO}$’s range is $S_O = \{0, 1\}^n$.
- $\text{normOT}$’s range is $S_{OT} = \{(0, x_2, \ldots, x_n) \in \mathbb{Z}^n\}$.

$S_{OT} = S_O \cap S_{T1}$, therefore $\text{normOT}$ is a normalization for $OT$.

Corollary 3. Algorithm\[3] normOP, is a normalization for OP-space.

Proof.

- $\text{normO}$’s range is $[0, 1\}^n$.
- $\text{normP}$’s range is $S_P = \{\bar{x} \in \mathbb{Z}^n | \bar{x} = \text{sort}(\bar{x})\}$
- $\text{normOP}$’s range is $S_{OP} = \{\bar{x} \in [0, 1\}^n | \bar{x} = \text{sort}(\bar{x})\}$

$S_{OP} = [0, 1\}^n \cap S_P$, therefore $\text{normOP}$ is a normalization for $OP$.

A normalization also exists for $PT$:

Algorithm 4. $\text{normPT} = \text{normT} \cdot \text{normP}$

The proof that this is a normalization also follows directly from theorem\[2]

Finding a normalization is not always the most efficient way to check for equivalence class membership. An example of this is OPT-equivalence. While a convenient representative subset of OPT-space exists, the set of all vectors in $[0, 1\}^n$ whose first element is zero and whose intervals between elements are sorted in ascending order $[2][11]$, it is not easy to normalize chords into this subset of $\mathbb{Z}^n/OPT$. The reason for this is illustrated by the points $\langle 0, 2, 7 \rangle$ and $\langle 0, 5, 7 \rangle$, which are related by:

$$\langle 0, 5, 7 \rangle \sim_O \langle 12, 5, 7 \rangle \sim_P \langle 5, 7, 12 \rangle \sim_T \langle 0, 2, 7 \rangle$$

The point $\langle 0, 5, 7 \rangle$ should, therefore, be normalized to $\langle 0, 2, 7 \rangle$ under the conventions of our representative subset. However, we cannot use any of the normalizations discussed so far to accomplish this. $\langle 0, 5, 7 \rangle$ will be mapped to itself with normP, normO, and normT. The same thing happens with $\langle 0, 2, 7 \rangle$ as well. Therefore, we have two choices: create one or more new normalizations, or use another algorithm to test whether two chords are OPT-equivalent.

We have chosen to use another algorithm that, although makes use of the $O$, $P$, and $T$ normalizations, does not define a normalization for the $OPT$ relation itself. This algorithm returns true if and only if two chords are OPT-equivalent.

Algorithm 5. optEq(\bar{x}, \bar{y}) =

1. Let $\bar{x} \leftarrow \text{normOP}(\bar{x}), \bar{y} \leftarrow \text{normOP}(\bar{y})$.
2. Let $S_T = \{\text{normPT}(\bar{v} + 12\bar{T}) | \bar{v} \in [0, 1\}^n \land \bar{v} \neq 1^n\}$.
3. If $\bar{x} \in S_T$ then return true, otherwise return false.

Two chords are OPT-equivalent if some stacking of their pitch classes have the same multiset of intervals. $S_T$ includes all possible sorted intervallic structures of $\bar{y}$’s pitch classes. We have that $\bar{x} \in [0, 1\}^n$, $\bar{x}$ is sorted, and $S_T$ includes all sorted, OPT-equivalent vectors to $\bar{y}$ within the range $[0, 1\}^n$. Therefore $\bar{x} \in S_T \iff \bar{x} \sim_{OPT} \bar{y}$.

3. GENERATING CHORD PROGRESSIONS

Finding a normalization for a chord space is important because it is one way to determine which chords belong to the same equivalence class. Many progressions will share the same pattern of representative points when their chords are normalized, so the process of rewriting harmony for $n$ voices using chord spaces can be modeled as a three-step process:

1. Normalizing the chords in the original progression to find a path through the representative subset of $\mathbb{Z}^n/R$.
2. Generating all solutions that share the same normalized path.
3. Choosing a solution with the desired characteristics.
In the case of our example for three tenors, $R$ would be OP and the desired characteristics would include specific ranges for each voice.

A sequence of equivalence classes represents many possible concrete progressions, each with a unique path through $\mathbb{Z}^n$. In $OP$-space, choosing the specific shape of the path through a series of equivalence classes is analogous to choosing a voice-leading behavior. For a chord space, $\mathbb{Z}^n/R$ and a sequence of concrete chords, $X = [\vec{x}_0, ..., \vec{x}_n], \vec{x}_i \in \mathbb{Z}^n$, the set of chord progressions sharing the same sequence of equivalence classes as $X$ is:

$$\{ [\vec{y}_0, ..., \vec{y}_m] | \vec{y}_i \in E(\vec{x}_i, \mathbb{Z}^n/R) \}$$

(4)

We can further constrain this to some subset of $\mathbb{Z}^n$, limiting the range of each voice based on instrumental or performer constraints.

$$\{ [\vec{y}_0, ..., \vec{y}_m] | \vec{y}_i \in E(\vec{x}_i, S/R), S \subseteq \mathbb{Z}^n \}$$

(5)

Finally, we want solutions that exhibit desirable behavior, such as those having certain voice-leading characteristics (e.g. smooth voice-leadings, no voice crossings, etc.). Given a musical predicate, $H$, that defines what desirable behavior is for a chord progression, we can further narrow the set of candidate solutions:

$$\{ [\vec{y}_1, ..., \vec{y}_m] | \vec{y}_i \in E(\vec{x}_i, S/R), S \subseteq \mathbb{Z}^n \land H([\vec{y}_1, ..., \vec{y}_m]) \}$$

(6)

### 3.1. Musical Predicates

We define two types of musical predicates: pairwise and progression predicates. Pairwise predicates apply to pairs of chords, and progression predicates apply to a sequence, or progression of $m$ chords.

First we define a pairwise predicate that rejects cases where voice crossing occurs. When voices cross, a permutation that sorts one chord’s voices will not correctly sort the other chord’s voices.

$$h_{\text{noCross}}(\vec{x}, \vec{y}) = \exists \sigma \in S_n \mid \sigma(\vec{x}) = \text{sort}(\vec{x}) \land \sigma(\vec{y}) = \text{sort}(\vec{y})$$

(7)

Similarly, we define pairwise predicates for avoiding parallel motion (when two voices move in the same direction by the same amount) and for limiting the maximum movement of each voice.

$$h_{\text{noPar}}(\langle x_1, ..., x_n, \rangle, \langle y_1, ..., y_n \rangle) = \forall i, j \in [1, n], i \neq j, x_i - x_j \neq y_j - y_j$$

$$h_{\text{maxPar}}(t, \langle x_1, ..., x_n \rangle, \langle y_1, ..., y_n \rangle) = \forall x_i, y_i, |y_i - x_i| \leq t$$

(8)

(9)

Finally, given a pairwise predicate, $h_{\text{pair}}$, a progression predicate can be created from it.

$$h_{\text{prog}}(h_{\text{pair}}, [\vec{x}_1, ..., \vec{m}]) = \bigwedge_{i=1}^{m-1} h_{\text{pair}}(\vec{x}_i, \vec{x}_{i+1})$$

(10)

### 3.2. Algorithms for Applying Predicates

Finding a chord progression satisfying certain predicates is analogous to the satisfiability problem in computer science, which is NP-complete for arbitrary formulas on Boolean variables [7]. Because of this, there is a tractability issue involved in finding candidate solutions that satisfy one or more potentially arbitrary predicates. If there are $k$ possible choices for each of $m$ chords in a progression, there are $k^m$ total possibilities. For a given quotient space $S/R$ and predicate $H$ we clearly need a more efficient method for finding solutions than generating all $R$-equivalent solutions, storing them, and then looking for cases satisfying $H$. An algorithm generating $H$-acceptable solutions must perform more aggressive pruning of the solution space.

One way to prune the solution space is through the use of predicates that operate on sub-progressions, such as pairwise predicates. An algorithm can apply the predicates while generating partial solutions. We present one such algorithm below for pairwise predicates. While this general strategy does not change the complexity class of the problem, it avoids computing and storing unnecessary progressions.

**Algorithm 6. pairProg**

1. If $m = 1$, return $E(\vec{x}_1, S/R) = \{ \vec{z} \mid \vec{z} \in S \land x \sim_R, \vec{x} \}$, otherwise continue.

2. Let $Y = \text{pairProg}(R, S, h_{\text{pair}}, [\vec{x}_2, ..., \vec{x}_m])$

3. Return $\{ [\vec{y}_1, \vec{y}_2, ..., \vec{y}_m] | \vec{y}_1 \in E(\vec{x}_1, S/R), [\vec{y}_2, ..., \vec{y}_m] \in Y, h_{\text{pair}}(\vec{y}_1, \vec{y}_2) \}$

Even when solutions are filtered using predicates, the work involved in traversing the entire set of solutions to locate desirable ones and even the number of desirable solutions can be intractable in situations involving many chords, many voices, and/or large ranges for the voices. Fortunately, in music, rules are not always strict, and so it may be sufficient to find a solution that mostly satisfies a set of predicates, even if some parts of the solution violate the predicates. We present an alternative, greedy algorithm for generating chord progressions that, while not guaranteed to find a solution satisfying a progression predicate, will attempt to satisfy a pairwise predicate when choosing each chord.

**Algorithm 7. Let $\vec{x}_i$ be the chord for which we wish to find a new $R$-equivalent member of $S \subseteq \mathbb{Z}^n, \vec{y}_i$. Let $\vec{y}_{i-1}$ be the previously chosen chord, choose($S$) be a function to stochastically select an element from a set, and $f(\vec{x}_i, \vec{y}_{i-1}, E(\vec{x}_i, S/R))$ be a fall-back method for choosing $\vec{y}_i$.**

1. Let $S_H = \{ \vec{y} \in E(\vec{x}, S/R) | H_{\text{pair}}(\vec{y}_{i-1}, \vec{y}) \}$

2. If $S_H = \emptyset$ then return $\vec{y}_i = f(\vec{x}_i, \vec{y}_{i-1}, E(\vec{x}_i, S/R))$. Otherwise, return $\vec{y}_i = \text{choose}(S_H)$. 


Algorithm 8.  
\begin{align*}
\text{greedyProg}(\vec{x}_1, \ldots, \vec{x}_n, S \subseteq \mathbb{Z}^n, R, \text{H}_{\text{pair}}, f) &= \{ \vec{y}_1, \ldots, \vec{y}_m \},
\end{align*}

where
\begin{align*}
\vec{y}_i &= \begin{cases} 
\text{choose}(E(\vec{x}_1, S/R)) & \text{for } i = 1 \\
\text{greedyChord}(\vec{x}_i, \vec{y}_{i-1}, S, R, \text{H}_{\text{pair}}, f) & \text{otherwise}
\end{cases}
\end{align*}

The main advantage to this approach is that the solution space is maximally pruned at each step. This allows the algorithm to operate on inputs that would cause tractability problems for \textit{pairProg}. The downside is that \textit{greedyProg} is not guaranteed to find predicate-satisfying solutions. It is possible to find a partial solution with no subsequent choices that satisfy the supplied predicate. In such a situation, there are three options: fail and return an error message, backtrack to try to find a better solution (analogous to lazy evaluation of \textit{pairProg}), or try another predicate. For our implementation, we chose the latter: a fall-back method for choosing a next chord is therefore required if we wish to ensure that \textit{greedyProg} produces a solution. In practice, it may be sufficient to have a result that mostly satisfies a predicate even if some chord transitions do not. Since this greedy approach does not require examining all possible solutions, it presents a more tractable option for larger-scale composition problems.

3.3. Moving Between Levels of Abstraction

Problems of solution space size mentioned so far are more prominent for chord spaces with very large equivalence classes. For example, a given equivalence class in \textit{OP}-space can be larger than in \textit{O}-space. Similarly, \textit{OPT}-space will have some larger classes than \textit{OP}-space. However, before generating a final solution, we can move between levels of abstraction using only representative subsets at intermediate steps.

For example, it would be possible to find a path through a representative subset of \textit{OPT} space using predicates on chord quality and then, in a second step, transform that abstract path into one through $\mathbb{Z}^n$/\textit{OP} with predicates to shape the voice-leadings. If we were to try to find solutions in \textit{OPT}-space directly using a predicate for both chord quality and voice-leading behavior, the number of initial possibilities to explore would be much larger.

4. APPLICATION TO VOICE-LEADING

We implemented the algorithms described in the previous sections using Haskell, a lazy programming language that allows for a concise and elegant implementation.\footnote{Our implementation is available at the Yale Haskell Group website, http://haskell.cs.yale.edu/}

We present examples relevant to our originally described problem: re-writing existing chord progressions. Durations of chords shown in the examples do not change, since our algorithms only make decisions about the assignment of pitches to voices (not duration).

Our results show potential as a framework for some tasks in automated composition. We control specific solution choice with the use of predicates and we use chord spaces as a way to organize the solution space at different levels of abstraction. Our results also highlight several important issues in the representation of chord spaces for compositional tasks as well as problems with the tractability of searching a particular solution space under a set of predicates.

We present sample results using a three-voice input progression created by the authors, shown in Figure 1. This chord progression is then re-written using our algorithms and two different chord spaces: \textit{OP}-space and \textit{OPT}-space. The examples make use of the following predicates:

- $h_{\text{noCross}}(\vec{a}, \vec{b}) = \text{true iff no voice crossing occurs between } \vec{a} \text{ and } \vec{b}$.
- $h_{\text{noPar}}(\vec{a}, \vec{b}) = \text{true iff no parallel motion occurs between any pair of voices in } \vec{a} \text{ and } \vec{b}$.
- $h_{\text{max}}(\vec{a}, \vec{b}) = \text{true iff no voice moves more than 7 halfsteps from } \vec{a} \text{ to } \vec{b}$.
- $h_{\text{all}}(\vec{a}, \vec{b}) = h_{\text{noCross}}(\vec{a}, \vec{b}) \land h_{\text{noPar}}(\vec{a}, \vec{b}) \land h_{\text{max}}(\vec{a}, \vec{b})$
- $h_{\text{allProg}}(p) = h_{\text{prog}}(h_{\text{all}}, p)$

For the three-voice progression in Figure 1, two \textit{OP}-equivalent and two \textit{OPT}-equivalent progressions are shown. Figures 2, 3, and 5 utilize the range of $[36, 57]$ for each voice. Figure 4 uses a representative subset of \textit{OP}-space instead. For the sake of score readability, we chose to use a version of this progression transposed up by 48 halfsteps to lie within $[48, 59]$ rather than $[0, 11]^3$.

4.1. Greedy Algorithm Performance

Using the same input progression, $h_{\text{all}}$, and \textit{greedyProg}, we ran an experiment to estimate the success rate of \textit{greedyProg}. This progression demonstrates the tractability issues associated with the general problem of rewriting chord progressions. There are 11 chords in the input progression, and, within the range $[36, 57]^3$, there are 24 \textit{OP}-equivalent ways to choose each of the first, second, and third chords and 48 ways to choose each of the remaining eight chords. This gives a total of $24^3 \cdot 48^8$ possible \textit{OP}-equivalent solutions. Of those solutions, we used \textit{pairProg} to determine that 901728 of them satisfy $h_{\text{allProg}}$ within the range $[36, 57]^n$.

We ran 10000 trials with \textit{greedyProg} to find \textit{OP}-equivalent solutions to the input progression in Figure 1 within the range $[36, 57]^3$. We used $h_{\text{max}}$ as a fall-back method for \textit{greedyProg}. Under these conditions, 8165 of \textit{greedyProg}’s returned progressions satisfied $h_{\text{allProg}}$. 7406 of those $h_{\text{allProg}}$-satisfying progressions were unique solutions. Of the 1835 returned progressions that violated
h_{altProg}, an average of only 1.1 chord transitions per progression violated h_{alt}, with the highest number of h_{alt}-violating transitions being 2. We feel that greedyProg performed well in this experiment. However, the algorithm’s performance will be directly affected by the range of the voices and specific predicate used. Decreasing the range of each voice, for example, would increase the odds that greedyProg would become stuck and rely on its fall-back function.

![Figure 1](image1.png)

**Figure 1.** A simple chord progression for three voices.

![Figure 2](image2.png)

**Figure 2.** A solution generated with greedyProg using [36,57]^{4}/OP, figure 1’s progression as input, and h_{alt}.

![Figure 3](image3.png)

**Figure 3.** A solution generated with pairProg using [36,57]^{4}/OP, figure 1’s progression as input, and h_{alt}.

### 4.2. Levels of Abstraction

The OPT-equivalent progressions to Figure 1, Figures 2 and 3 demonstrate use of different levels of abstraction for rewriting chord progressions. Figure 4 is an OPT-equivalent progression to Figure 1 using S_{OP}/OPT, where S_{OP} is a representative subset of OP-space lying within [48,59]^{3}.

Figure 5 is an OP-equivalent path to Figure 4 and an OPT-equivalent path to Figure 1. It is a randomly selected solution from the set generated by pairProg and h_{alt}. Because the input progression from Figure 1 only consists of major and minor trichords, its OPT-equivalent progressions are only required to share the same quality (major or minor) for each chord.

Although not exactly musical, the results from OP-space are promising in some ways. OP-space can produce musical results given appropriate predicates and a musical set of pitch classes as labels, but the same system can be generalized to make choices about more than just pitch class. Unfortunately, in our examples, OPT-space is too generalized to imitate the types of pitch class decisions humans might make, because the labels are only an indicator of chord quality. Ultimately, spaces with a level of generality somewhere between OP- and OPT-space may be most useful, giving an intermediate number of equivalence classes. For example, in a particular musical context, two OP classes may be functionally equivalent and therefore could be merged. Finding more appropriate, intermediate levels of abstraction may also require more complex spaces, with points holding more information than just pitches, such as home key. Additionally, this framework may benefit for including additional concepts such as Morris’s contour spaces for representing musical features more diverse than just chords.

![Figure 4](image4.png)

**Figure 4.** A solution generated with greedyProg with an always-true predicate and a transposed version of S_{OP}/OPT (4 octaves higher for readability)

![Figure 5](image5.png)

**Figure 5.** A solution generated with pairProg using [36,57]^{3}/OP, figure 1’s progression as input, and h_{alt}.

When using OP-space to choose voice-leadings for pitch classes, it may be also useful to learn a particular style or composer’s method of choosing voice-leadings from a data set. Similarly, equivalence relations for concepts such as chord substitutions and short progressions could be learned. Machine learning algorithms such as probabilistic suffix trees may aid in extracting relevant sub-progressions to learn concepts such as chord substitution for use in more complex quotient spaces of short chord progressions.

### 4.3. Representing Equivalence Relations

One shortcoming of our implementation is the need for single, Boolean tests for equivalence under multiple relations. Partitioning a set under multiple relations creates representational problems: either the space must be large enough that it contains all intermediate points needed to preserve transitivity, or functions must be developed to test for equivalence under multiple relations in one Boolean.
test. In other words, if we wish to compute \((S/R_1)/R_2\), we must find a suitable test, \(r\), for \(R = R_1 \lor R_2\) and then compute \(S/R\) directly instead of first finding \(Q = (S/R_1)\) and then computing \(Q/R_2\). As we have shown, this is possible for the \(OPT\) relations.

It is not always possible to find simple and concise formulas to combine multiple equivalence relations into single Boolean tests as we have done for all combinations of the \(O\), \(P\), and \(T\) relations. In the case of \(OPT\) equivalence, we relied on understanding of the music theoretic aspects of the relations to create a simple Boolean test. Equivalence relations learned from data sets could require storing and traversing the entire learned model to perform equivalence tests.

Storing an entire chord space is also problematic, since the number of chords grows exponentially with the number of voices. For large numbers of voices, it may be necessary to generate the quotient space on an as-needed basis. These issues must be addressed in order to pursue applications that involve large quotient spaces.

### 4.4. Algorithms for Satisfying Predicates

The size of the solution spaces demonstrated by our simple examples shows the need for intelligent methods of traversing these spaces without generating and storing each possible solution. With lazy evaluation in a language like Haskell, it is relatively easy to calculate the first solution that satisfies some predicates. However, choosing a random solution requires knowing how many solutions exist a priori.

The greedyProg algorithm avoids this issue by making stochastic choices as it generates the solution. However, the trade-off is the potential for becoming stuck, with no predicate-satisfying options for future choices. One possible improvement would be to incorporate back-tracking into an algorithm like greedyProg as a fall-back method when the algorithm becomes stuck. This type of approach would still allow for the selection of a random, predicate-satisfying solution without necessarily requiring generation of the entire solution space.

### 5. CONCLUSIONS

We have presented a general framework for rewriting chord progressions using chord spaces and musical predicates and added additional formalization to concepts important for implementing and using chord spaces. We have presented two algorithms for generating chord progressions within this framework. Our greedy algorithm shows potential for avoiding tractability issues associated with the size of musical solution spaces.

We foresee algorithms like those presented here being useful in tasks such as algorithmic composition of novel music and arrangement of existing composition. Particularly in the case of algorithmic composition, there is a need to address structure at different levels of abstraction. Chord spaces present one such method for moving between levels of abstraction, but other quotient spaces could be used as well, such as quotient spaces formed from sets of short progressions rather than sets of individual chords. In our ongoing research, we are investigating the construction of equivalence relations to form these more complex quotient spaces, as well as more efficient ways to store and traverse them.

### Acknowledgements

This research was supported in part by NSF grant CCF-0811665.

### 6. REFERENCES


