CSCI 4511W Final Project Report

Broden Wanner

May 11th, 2020

Abstract

In order to address the problem of using artificial intelligence to compose music, researchers have used the approach of Constraint Programming, which solves combinatorial problems using variables, constraints on those variables, and partial information. Early attempts used constraints defined in first-order logic while later attempts focus more on heuristic search with cost functions. This paper will explore a more direct approach of implementing constraints in explicitly defined terms and modelling the problem as a Constraint Satisfaction Problem with using Python, which differs in other approaches that mainly use Lisp, in order to create simple harmonies based off of roman numerals. Different constraints willed be evaluated based on time to the solution and the subjective quality of it.

1 Introduction

Recently, there have been efforts by members of the AI community to build creative Artificial Intelligences, especially ones for making music. Popular examples include Google's *Magenta*[18] and Pierre Barreau et al.'s *Aiva*[3]. Although they do not disclose how their software works, it appears that they use Deep Learning techniques to train their software on existing music to generate new music, trying to discover how machine learning can be used creatively. However, it can be incredibly difficult to leverage this type of machine learning to create music because of its complex, multi-faceted, and rigid structure. In order to sound appealing to our ears, these machine learning models require coherently modeling structure at many different timescales, which adds to the difficulty of the problem [8]. Western Music as we know it uses discrete notes to produce and communicate music (music from other cultures, such that from Nepal, sometimes uses continuous representations, but we will not address this). And most musical composition follows established forms based on the concepts of chords and tonality. Therefore, instead of viewing AI musical composition as a machine learning-based problem trained on existing music, we can view this as a search problem that finds melodies and harmonies based on our established conventions of music.

This project aims to solve the problem of AI music composition by modeling it specifically as a constraint satisfaction problem (CSP) [17]. Here, we will apply CSP techniques to music using conventions from what is called the "Common Practice Period" (ca. 1600-1910). Applying Constraint Satisfaction to this music is a very natural idea, because of its harmonization rules which are constraint-like stated [20]. Music theory textbooks give rules like "no parallel fifths", "opposite motion between two voices", and so on. They are written in a declarative way. Given a set of roman numerals (explained later in the report), a key, and a set of voices, this CSP help to create voicings that are consistent with the rules of of Western Classical Music. Once this is done, it can be used to create novel melodies and harmonies by creating variations of each voice that each add interesting qualities to the music. However, this project will focus solely on creating the voicings of the harmonies.

The project will focus on using a CSP to create harmonies based on the rules of Western music theory. In a given key, each chord can be assigned a Roman numeral, an inversion, and a quality, and there are specific rules about which chords should follow other chords. Each variable in the CSP can be each note for each part and each successive chord in a composition and the domains will contain the possible notes, numerals, inversions, and qualities for the chords. The constraints will be the various rules of harmony and melody from the Common Practice Period. Then, I will use arc heuristic techniques and searching to find

4-part voicing solutions based on a partial assignment of the roman numerals. However, for constraints that are not absolute (for instance, even though leaps of a 7th are eschewed, they can still be used), it will be difficult to use backtracking search to find more diverse melodies based on these *preference constraints*, as Russel defines them [17]. And because the constraints used in the problem usually involve more than two variables, we need to use generalized arc consistency instead of faster methods such as AC3. The nature of the constraints makes it nearly impossible to reduce to binary constraints.

To implement the CSP, Python will be used with the music21 library, which was developed at MIT and provides various classes and methods for music-theoretical analysis and programming. The library also supports exporting to MIDI output and MusicXML. The Aima Python Library from the Russel textbook [17] will be used to define the CSP and as a model for the constraints. Objectively, the results of the experiments will be measured based on time to solution, arc heuristics used, and adherence to the rules of the Common Practice Era of Western Music. But because music is mainly a subjective human experience, the output is also listened to directly and evaluated. The overall goal of the project will be to evaluate the best constraints to use when creating harmony using a constraint satisfaction problem.

2 Background

Music generation is already a well-established research area in field of artificial intelligence. It is overall a notoriously difficult problem to solve, and there are many different domains within music generation that computer music systems examine. Many examples include the type of music to be generated, such as creating a single melodic line or complex polyphony, the style to compose the music in, such as Baroque or Jazz, and the way to incorporate pseudo-human expression into the music. Since the 1950s, different computational techniques related to Artificial Intelligence have been used for musical composition, including grammatical representations, probabilistic methods, deep learning and neural networks, symbolic rule-based systems, constraint programming and evolutionary algorithms [15]. Pre-computer methods existed for algorithmically composing music such as species counterpoint or Schoenberg's 12-tone technique. Using these pre-established techniques and harmonization rules that dominate Western Music, we can strive to create music using artificial intelligence. However, incorporating some degree of creativity into the music is incredibly difficult in itself. In order to narrow the focus of this project, we will only be examining a small subset of the wide array of literature on AI musical composition—that of Constraint Satisfaction Problems (CSPs).

One of the first applications of using a constraint satisfaction problem for musical composition is Ebcioğlu's fifth-species counterpoint music writer [7]. Species counterpoint in Common Practice Period Music (ca. 1600-1900) specifies a rigid set of rules for two-part voice writing. Fifth-species (or florid) counterpoint combines every rule from the preceding four species to write a melody in a rhythmically free style that matches a given Cantus Firmus (the bass line). Implemented in Lisp, he translated the rules in composable Boolean functions that say whether or not a given melody follows all the rules. There is an enumeration algorithm that will then output one-at-a-time all the melodies having the property corresponding to a given rule. It uses a search strategy that influences the order of enumeration of the melodies so that more desirable melodies are generated first. Ebcioğlu found that the given rules were incredibly insufficient and thus had to construct numerous other rules to make an acceptable output. Approximately 8 years later, he addressed the problem of writing four-part chorales in the style of Bach using a system called CHORAL that distilled his compositional style into about 350 rules to guide harmonization and melody generation [6]. He invented a custom logic language called BSL (Backtracking Specification Language) that had optimizations for CSP techniques such as backjumping and cutset ordering to make the problem of music composition tractable. The methods Ebcioğlu uses lie more in the area of Constraint Logic Programming (CLP), which uses first-order logic to represent constraints for CSPs.

Following Ebcioğlu's research, numerous constrain logic programming or constraint satisfaction systems have been developed for harmonization or counterpoint. Ovans and Davison created a CSP system that takes input from the user who guides the search process [11]. Their approach models the task of composing contrapuntal music as a CSP and illustrates the potential of separating generative (search) from restrictive (constraints) knowledge in expert systems. They used arc consistency to make their algorithm more efficient in resolving constraints. Phon-Amnuaisuk used a similar style to Ebcioğlu to harmonize chorales in the style of Bach using CLP, yet it was innovative in the way in which the user can add additional rules to control the process more explicitly [14]. In contrast to other CSP or CLP problems solved previously, Anders and Miranda presented a formal model of Schoenberg's guidelines for compelling chord progressions that are self-contained and do not depend on some human input for partial assignment [2]. This solves one the most pressing problems for musical composition according to Pachet and Roy [13]. It also generalizes Schoenberg's rules to make harmonic progressions that are more applicable in other situations (not just for 4-voiced Bach chorales). Novelly, Anders and Miranda model Schoenberg's explanation of these rules as constraints between chord pitch class sets and roots instead of as constraints on scale degree intervals. This allows for the introduction of more chord tones instead of the usual four.

Deviating from the common problem of harmonization and melody-writing with CLP and CSPs, other researchers have delved into new ways to look at music composition with artificial intelligence. Zimmermann presents a modelling system called COPPELIA that generates music on the basis of the structures, goals, and contents of given multimedia presentations [21]. Previous approaches stress the technical aspects of applying constraints and do not examine the concrete role of the constraints, yet the composer is usually more focused on what they want to convey than with theoretical rules (unless they are music theorist of course). The system, implemented in Lisp, has two components called AARON (which derives a certain musical framework on the basis of a given *intentional* structure) and COMPOzE (which derives concrete audible data from the framework delivered by AARON). AARON takes a "storyboard" to configure abstract parameters such as the ambience as a function of time and then generates a harmonic progression and sequence of directives. COMPOzE actually does the composition by taking the progression and directives and adding individual notes and melody. Ventureing out from Western Music, Chemillier and Truchet applied constraint programming to two different sets of music: a style of Central African harp music and Ligeti textures [4]. Instead of backtracking search, they used heuristic search, which was a fairly important advancement in musical CSPs.

Numerous general-purpose constraint programming systems have been proposed and implemented for musical composition, which are languages and environments that are used to program the constraints. Pachet and Roy created the BackTalk system for CLP, which is a canonical integration of obects, in the sense of object-oriented programming (OOP), with a finite-domain CSP [12]. The main algorithms in BackTalk are a generalization of AC-3 to n-ary and functional constraints for arc-consistency and a parameterized enumeration algorithm. The MusES system was a project to represent knowledge about basic harmony in Smalltalk (an object-oriented, dynamically typed reflective programming language) to study the adequacy of various representation techniques to capture the essence of musical structures. The MusES approach is based on the remark that only three types of operations are important with intervals: (1) computing the interval between two notes, (2) computer the top note given the bottom one in a a chord, and (3) computing the bottom note given the top note in a chord. The salient quality of Backtalk is its use of object-oriented programming instead of logic programming to represent and generate music. An earlier systems was Courtot's CARLA, which was a CLP system written in Prolog for generating polyphonies with a visual GUI and an extendable type system designed to represent relationships between different musical concepts [5]. The goal was to provide a model which tries to represent the way a composer structures their knowledge but in a computer-aided environment. Agón's OpenMusic is a visual language for Computer Assisted Composition (CAC) based on CommonLisp and CLOS [1]. Although it does not explicitly provide automated musical composition using CSPs, it makes it very easy to do so, and some of the following papers use it in this way. PWConstraints (PatchWork Constraints) is a rule-based, visual programming language that allows the user to create rules from many different viewpoints (harmony, melody, etc.) [9]. A search problem is defined in two steps: a search space definition with search variables and variable domains, and a set of rules operating on that search space. PWConstraints was able to relatively easily handle problems in polyphonic composition through a subsystem (score-PMC). Another general-purpose system was called Situation [15], written in OpenMusic. It was more flexible than PWConstraints and implemented more optimizations in its search procedures.

Rueda et al. created an expirimental language for algorithmic musical composition that integrated concurrent objects and constraints called *PiCO* [16]. They proposed having constraints and objects as primitive notions and seamlessly integrating in a constraint system using ρ-calculus. Concurrent processes make use of a central constraint store to synchronize a knowledge base using ASK and TELL operations of the constraint model. *PiCO* makes it easy to represent complex, partially defined objects (like musical structures) in a compact way. Using a front-end written in Cordial, they created a visual language for ease of use like so many other musical composition contraint languages. A similar concept to *PiCO* was *ntcc*, another language for constraint programming with primitive notions of constraints and objects for defining concurrent systems [15]. *ntcc* was proposed as an alternative to *PiCO* that could be used for "more expressive" algorithmic musical composition and could generate more varied rhythmic patterns [10]. However, both of these languages have fallen out of favor and are mostly used for teaching programming language design.

In the more recent years of Constraint Programming and musical composition, we see a markedly different approach to the problem. The first example is with OMClouds, another purely visual, general-purpose constraint system used for algorithmic composition written in OpenMusic [19]. Instead of using explicit constraints using CLP, it implemented all the constraints as cost functions. An adaptive tabu search was used to find the solution with the minimal cost instead of using backtracking in the usual CSPs. With this, users could avoid the problem over overconstraining the problem, but it could not be guaranteed to navigate the search space as tabu search is not complete [17]. Later, Truchet and Codognet published a better approach to the adaptive search problem in algorithmic composition by defining more specific problems and domains in musical composition using OMCLouds [20]. The specific problem areas include harmony, rhythms, melodies, musical analysis, and diatonic sequences. The adaptive search algorithm then works as follows: Compute errors of all constraints and combine errors on each variable, select the variable X with highest error and evaluate costs of possible moves from X, if no improvement exists, mark X tabu for a certain number of iterations else select the best move using the min-conflict heuristic and change the value of X accordingly.

The application of constrain programming to algorithmic musical composition is a well-studied area of artificial intelligence at this point in time, having many languages developed just for its purposed. They have been used to solve a wide range of problems from harmony-writing to melody-writing to monophony to polyphony to expressiveness to rhythm. Early attempts at solving the problem looked more at explicit constraints in CLP. More recently, we have had advancements using a more heuristic approach to the constraint systems.

3 Methods

As mentioned in the Introduction section, this project aims to create two to four voiced harmonies using constraint satisfaction problems with constraints based on the rule of Western Music. This section will first outlined terminology used in this project and then the overall method for analyzing and solving the problem.

3.1 Terminology

This section will define terminology used to describe concepts of Western Musical Theory used in this project. A basic understanding of notes and the staff in Western Muisc is desirable but not required. First, one should be familiar with the general concept of a *voice* in Western Music, which is an independent part that may be played at the same time as other voices (one could think of it as different instruments or parts in a band). This project focuses on using two to four voices, which is the most common in Classical Music. *Harmony* describes the vertical alignment of the notes in each voice, whereas *melody* or *counterpoint* focuses on the horizontal layout of each voice. An *interval* between two notes is simply the distance between each note on the staff. All intervals have a distance and quality that describe them, but they will not be enumerated here. For example, Figure 1 shows an interval of a perfect fifth. A *chord* is a group of three or more notes played at the same time, and these chords too have qualities based on the intervals between the notes in them (but we will not enumerate them here).

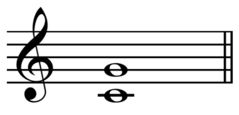

Figure 1: Interval of a Perfect Fifth

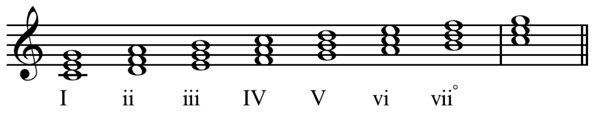

Figure 2: Roman Numerals in C Major

Central to Common Practice Music is the idea of a *tonic* or a note that is heard throughout a piece and is elaborated on by the music (hence, why it is called *Tonal Music*). Associated with the tonic is a *scale*, which is generally defined as a group of notes that are arranged by ascending or descending order of pitch based on the tonic. There are also different qualities of these scales, two of the most common ones are *major* and *minor*. The combination of a tonic and quality forms a *key*. Examples of keys include C Major, D Major, Bb minor, and so on. Composers would write pieces based on these keys centered around some tonic and having a major or minor quality. They would even name their pieces after them (like Beethoven's *Symphony No. 7 in A Major*). In order to accurately describe the harmonies that could be used in a given key, Roman Numeral notation was invented. Each scale degree has a triad (group of three notes) associated with it and is labelled with its respective roman numeral, as shown in Figure 2. Each of the constraints used in the problem will used concepts outlined here to define them.

3.2 CSP Formulation

This project uses a CSP representation of the task. A CSP has three general parts, as defined by Russell: a set of variables, a set of domains with one for each variables, and constraints on the variables that dictate allowable values on the variables [17]. Each constraint consists of a *scope* of a tuple of variables involved in the constraint and a *relation* that defines the values the variable can take on. A CSP is solved when some assignment of values to all the variables is made that is consistent with all the constraints on the variables. CSPs can have *discrete, finite* domains or *continuous* domains.

The success of composing music using a CSP depends largely on the way variables and domains are used. There are generally two approaches to defining them: vertical or horizontal. To think about it vertically is to think purely of the harmony and nothing of the actual voicing (i.e. which type of chord should follow each other type of chord). To think of it horizontally is to think of each note in each voicing and which specific note should come next. This project will use the horizontal approach to create voicing for the harmony since we are concerned with the specific notes in the voices. Thus, this means that every note in the piece that will be composed by the CSP must be a variable. As an example, for a four-voice composition with two measures at four beats per measure, there there must be $4 \cdot 2 \cdot 4 = 32$ variables. The formulation of the domains and the constraints then follow from this formulation of the variables.

The domains for each of the variables will then be the set of *musical notes* that each *note variable* can take on (here we make a distinction between *musical notes*, which are the values that the *note variables* variables can take one). In theory, this could mean that each variable could take on over 88 different values (the number of keys on a piano). For a basic two-measure piece with four voices, that already means that at least $32^{88} \approx 2.84 \times 10^{132}$ values would have to be checked. However, in reality, depending on the instrument, no range spans over that many notes. Typical choral pieces for instance only ask the singers to span at most an octave (8 notes). Therefore, using common practice assumptions, we can reduce the domains of each variable simply to the range of that voice. Building off of that, because we are using roman numerals and keys, we can greatly reduce the number of musical notes in each domain by only allowing each variable to take on values for the roman numeral and key associated with it.

The constraints for this problem are already well-defined in every music theory textbook. However, the problem lies in translating them into code. As mentioned earlier, because of the nature of the way the constraints are defined, it is incredibly difficult to formulate binary constraints, which Russell says are desirable to make search go quickly [17]. Table 1 contains the important constraints, their scope, and their definition in more abstract terms. (Not all are shown because there are over 20 constraints so far). Most of them are based on the intervals between adjacent voices on adjacent beats or simply on two notes in a voice.

The most important constraints are the *noParallelOctaves* and *noParallelFifths* constraints, which are rules that must never be violated in Western Music. It should be noted that the length of the scopes for each of these constraints changes depending on the number of voices to be composed. For instance, two voices requires the *noParallelOctaves* constraint to involve four notes (two notes per beat) while four voices would require 8 (four notes per beat).

Name	**Scope**	**Description**
differentNotes	Adjacent notes in one voice on adjacent beats	No two adjacent notes in a voice should be the same in order to add variety
noParallelOctaves	All notes in all voices in adjacent beats	No parallel octaves between any voices in adjacent beats
noParallelFifths	All notes in all voices in adjacent beats	No parallel fifths between any voices in adjacent beats (unless it is from a diminished fifth to a perfect fifth)
requireRootAndThird	All notes in one beat	Chord must include the root and the third for each roman numeral
ltResolvesUp	Adjacent notes in one voice	If it occurs, the leading tone (scale degree 7) of the key must resolve up by step
leapsUnderFifth	Adjacent notes in one voice	No leaps over a fifth in all voices (except the bottom one, for which it is an octave)
noVoiceCrossing	All notes in adjacent beats	No voice lines can cross, meaning that no voice can go below or over another and vice versa
isPAC	All notes in penultimate and final beats	Assert that the last two beats create a Perfect Authentic Cadence

Table 1: Important Constraints

3.3 Generalized Arc Consistency Algorithm

The Generalized Arc Consistency (GAC) Algorithm in the figure for Algorithm 1 makes an n-ary CSP arc consistent. It is based off of the AC3 algorithm but accepts CSPs with constraints that involve numerous variables in its scope. It returns true if the CSP is consistent and false otherwise. It also uses a function called NEW-TO-DOS (not shown) which returns a new set of to-do's to be examine based on assigning variable *var* in constraint *const*.

Algorithm 1 Generalized Arc Consistency Algorithm

```
function GAC(csp, domains, toDo)
    local variables: toDo, a set of (var, constraint) pairs
    if toDo is NIL then
        toDo ← all combination of variable and constraint pairs
    else
        toDo ← copy of toDo
    while  dotoDo is not empty
        (var, const) ← REMOVE-FIRST(toDo)
        otherVars ← every other variable in the scope of const besides var
        newDomain ← {}
        if LENGTH(otherVars) is 0 then
            for val in domain of var do
                if const holds when var = val then
                    add val to newDomain
        else if LENGTH(otherVars) is 1 then
            other ← first variable in otherVars
            for val in domain of var do
                for otherVal in domain of other do
                    if const holds when var = val and otherVar = otherVal then
                        add val to newDomain
                        break
        else
            for val in domain of var do
                if const holds when var = val and works for any variables in otherVariables then
                    add val to newDomain
        if newDomain is not the domain of var then
            domain of var ← newDomain
            if newDomain is empty then
                return (false, domains)
            addToDos ← NEW-TO-DOS(var, const) - toDo
            toDo = toDo ∪ addToDos
    return (true, domains)
```

3.4 Domain Splitting Algorithm

This is an algorithm used to get an actual solution for the CSP by calling GAC on half of each domain until it gets down to one value. It returns a solution to the CSP or false if it is inconsistent.

Algorithm 2 Generalized Arc Consistency Algorithm

```
function DOMAIN-SPLITTING($csp$, $domains$, $toDo$)
    ($consistent, newDomains$) ← GAC($csp$, $domains$, $toDo$)
    if $consistent$ is false then
        return false
    else if size of all domains is 1 then
        return solution with all variables assigned the one domain value
    else
        $var$ ← first variable in $variables$ where $|newDomain| > 1$
        ($dom1, dom2$) ← PARTITION-DOMAIN($newDomain$ of $var$)
        $newDoms1 \leftarrow newDomains \cup \{\{var : dom1\}\}$
        $newDoms2 \leftarrow newDomains \cup \{\{var : dom2\}\}$
        $toDo$ ← NEW-TO-DOS($var$, NIL)
        $res1$ ← DOMAIN-SPLITTING($csp$, $newDoms1$, $toDo$)
        $res2$ ← DOMAIN-SPLITTING($csp$, $newDoms2$, $toDo$)
        return $res1$ or $res2$
    return true
```

4 Design

Each of the algorithms, the CSP, and the constraints will be implemented in Python with the help of the music21 library for the music theory aspects and the Aima Python Library for the CSP helpers and solvers. To test the effectiveness of the combination of certain constraints and number of voices in the quality and speed of a solution, we will test the CSP solver using a standard 8-beat measure with a simple descending fifths sequence for the roman numerals for all parameters. This ensures that the time and quality to a solution depends not on the complexity of the roman numerals. The parameters that will be varied are the number of voices (2, 3, or 4), the ranges of each of the parts (an octave, a 12th, or two octaves), and a combination of the constraints. The solutions will then be evaluated on metrics such as time to solution, number of checks made for variables, and subjective sound appeal. Table 2 contains the variables to test and their values and Table 3 contains all the metrics to be used to measure the solution.

Parameter	**Possible Values**
Voices	2, 3, 4
Ranges	octave, 12th, two octaves
Constraint combinations	all, no *isPAC*, none

Table 2: Paramters

Metrics
Time in seconds to solution
Number of checks
Subjective sound quality

Table 3: Metrics

5 Results

The following are figures showing the various parameters vs. the number of checks required for the CSP to be solved. This information conveys the same data that the time to solution does, and thus time will not be shown in order to conserve space.

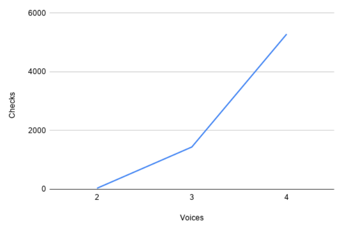

Figure 3: Voices vs. Checks

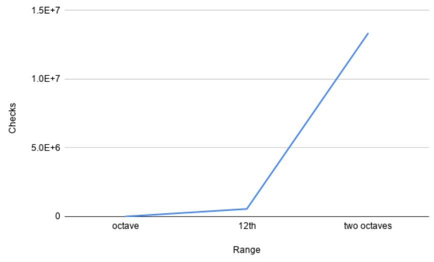

Figure 4: Range vs. Checks

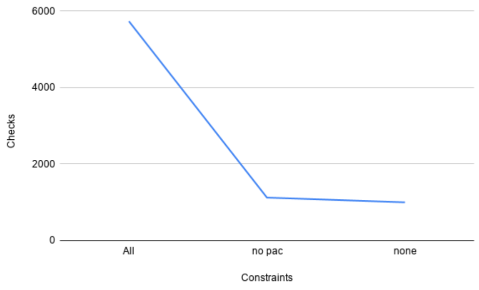

Figure 5: Constraints vs. Checks

6 Analysis

The code used to generate this was over 500 lines long and can be found at in the UMN GitHub repo at https://github.umn.edu/wanne036/ai-harmonizer, which will be made public after the due date for viewing. hen, Figure 6 shows a solution produced from the domain splitting using a range of one octave, 4 voices, and all constraints. The roman numerals for the descending fifths sequence are shown below for reference. A sample of the audio from the excerpt can be found at https://z.unm.edu/aiharmonizersample1. Overall, the solutions produced with domain splitting had a great sound quality (vocal voices were used to evaluate each part). However, there were still some voice leading errors in the parts. Parallel octaves were especially prevalent. It may have been the way the *noParallelOctaves* constraint was written that resulted in the errors. Moreover, it was obvious that increasing the size of the search space definitely increased the number of checks (and thus, the time) to get a solution.

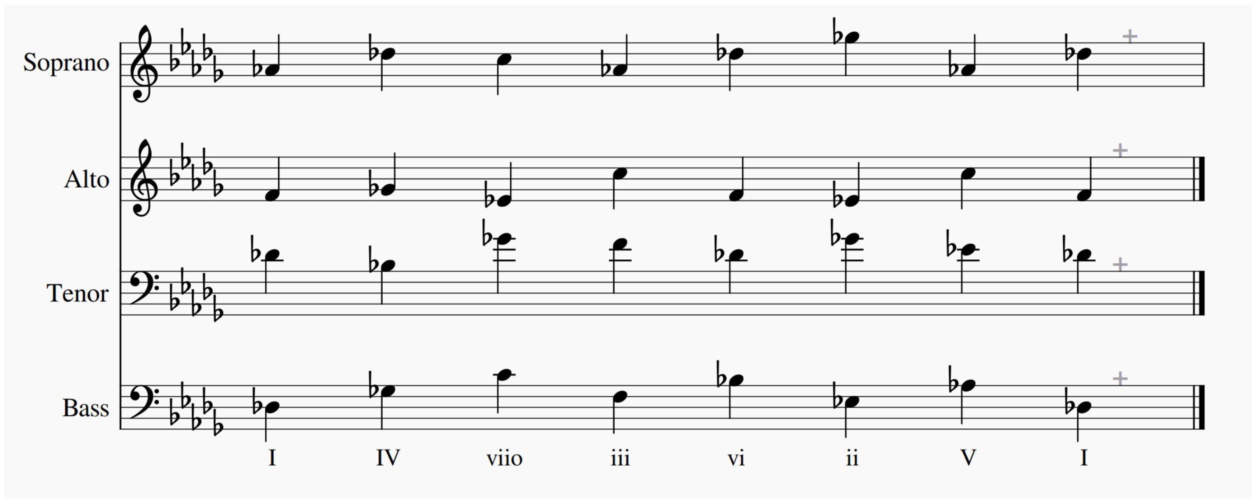

Figure 6: Solution with one-octave range, 4 voices, and all constraints

Something interesting about the results was that the *isPAC* constraint placed on the last two beats of the parts seemed to drastically increase the search space. Perhaps, it was because it involved so many variables. Another interesting fact was that removing all constraints did not necessarily remove the time to find a solution. This was probably because the DOMAIN-SPLITTING algorithm still needed to search through the space to find *some* solution. However, the sound quality with no constraints led to the solver choosing the same note for every beat, which was probably the first solution it saw when searching. All in all, more work needs to be done in developing the constraints and evaluating them on larger pieces in order to see which ones have the best effect.

7 Conclusion

In order to address the problem of using artificial intelligence to compose music, researchers have used the approach of Constraint Programming, which solves combinatorial problems using variables, constraints on those variables, and partial information. Early attempts used constraints defined in first-order logic while later attempts focus more on heuristic search with cost functions. This paper explored a more direct approach of implementing constraints in explicitly defined terms and modelling the problem as a Constraint Satisfaction Problem with using Python in order to create simple harmonies. Different constraints were evaluated based on time to the solution and the subjective quality of it.

More work needs to be done in creating these constraints to make better harmonies for multiple voices. In the future, one could create another CSP that can add variety to the parts based on the result for the simple harmony CSP. For instance, one can add passing tones (notes that connect an interval of a third in a part) to add interest. Or one could use a partial assignment of a melody and create the harmony based off of that instead of just roman numerals. Another approach would be to create a cost for each constraint and use a heuristic search approach such as adaptive search. All in all, it is still viable to create simple harmonies using a generalized CSP.

References

[1] Carlos Agón. “OpenMusic: un langage visuel pour la composition musicale assistee par ordinateur”. 1998PA066374. PhD thesis. 1998, 162 p. URL: `http://www.theses.fr/1998PA066374`.

[2] Torsten Anders and Eduardo Miranda. “A computational model that generalises Schoenberg’s guidelines for favourable chord progressions”. In: *6th Sound and Music Computing Conference* (Jan. 2009).

[3] Pierre Barreau et al. *Aiva: The Artificial Intelligence composing emotional soundtrack music.* 2020. URL: `https://www.aiva.ai/` (visited on 03/24/2020).

[4] Marc Chemillier and Charlotte Truchet. "Two Musical CSPs". In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming* (2001).

[5] Francis Courtot. "A constraint-based logic program for generating polyphones". In: *Proceedings of the International Computer Music Conference* (1990), pp. 292–294. URL: `http://hdl.handle.net/2027/spo.bbp2372.1990.082`.

[6] Kemal Ebcioğlu. "An Expert System for Harmonizing Four-Part Chorales". In: *Computer Music Journal* 12.3 (1988), pp. 43–51. ISSN: 01489267, 15315169. URL: `http://www.jstor.org/stable/3680335`.

[7] Kemal Ebcioğlu. "Computer Counterpoint". In: *Proceedings of the International Computer Music Conference* (1980). URL: `http://global-supercomputing.com/people/kemal.ebcioglu/pdf/Ebcioglu-ICMC80.pdf`.

[8] Curtis Hawthorne et al. "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset". In: 2019. URL: `https://arxiv.org/pdf/1810.12247`.

[9] Mikael Laurson. *PatchWork: A visual programming language and some musical applications.* Sibelius Academy, 1996.

[10] Carlos Olarte, Camilo Rueda, and Frank Valencia. *New Computational Paradigms for Computer Music.* Editions Delatour France, 2009. Chap. Concurrent Constraint Calculi: A Declarative Paradigm for Modeling Music Systems.

[11] Russel Ovans and Rod Davison. "An interactive Constraint-Based expert assistant for music composition". In: *Proceedings of the Canadian Conference on Artificial Intelligence* (1992), pp. 76–81.

[12] Francois Pachet and Pierre Roy. "Integrating constraint satisfaction techniques with complex object structures". In: (Jan. 1996), pp. 11–22.

[13] Francois Pachet and Pierre Roy. "Musical Harmonization with Constraints: A Survey". In: *Constraints Journal* 6 (1 2001), pp. 7–19.

[14] Somnuk Phon-Amnuaisuk. "Control Language for Harmonisation Process". In: Jan. 2002, pp. 155–167. DOI: `10.1007/3-540-45722-4_15`.

[15] Jose David Fernández Rodriguez and Francisco J. Vico. "AI Methods in Algorithmic Composition: A Comprehensive Survey". In: *CoRR* abs/1402.0585 (2014). arXiv: `1402.0585`. URL: `http://arxiv.org/abs/1402.0585`.

[16] Camilo Rueda et al. "Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language". In: *Constraints Journal* 6 (1 2001), pp. 21–52. DOI: `10.1023/A:1009849309451`.

[17] Stuart J. Russell. *Artificial intelligence: A Modern Approach.* 3rd ed. Pearson, 2016. Chap. 6.

[18] Tensorflow Team. *Magenta.* 2020. URL: `https://magenta.tensorflow.org/` (visited on 03/24/2020).

[19] Charlotte Truchet, Gérard Assayag, and Codognet Philippe. "OMClouds, a heuristic solver for musical constraints". In: Jan. 2003.

[20] Charlotte Truchet and Philippe Codognet. "Musical Constraint Satisfaction Problems Solved with Adaptive Search". In: *Soft Computing* 8.9 (2004). cote interne IRCAM: Truchet04a, pp. 633–640. URL: `https://hal.archives-ouvertes.fr/hal-01161219`.

[21] Detlev Zimmermann. "Modelling Musical Structures". In: *Constraints Journal* 6 (2001), pp. 53–83. URL: `https://doi.org/10.1023/A:1009801426289`.