CS182 Final Project: Four Part Harmony

Matthew Deutsch and Jarele Soyinka

December 13, 2016

https://github.com/JSoyinka/CS182-Final-Project

1 Introduction

In 1725, a music theorist named Johann Fux codified the set of rules of thumb that Baroque composers used to create good-sounding harmonies. Fux's theories are still taught in almost all introductory music theory courses, and a common assignment is to write music that follows Fux's rules. Since many of these rules are quite objective, it is possible to codify his prescriptions as a CSP.

We sought to produce an output that matches or is indistinguishable from harmonies written by real Baroque composers. We attempted to do this by modeling Fux's prescriptive rules as constraints in a CSP. In order to account for many of Fux's rules which are explicitly meant to be occasionally ignored, we encode these as soft constraints, and seek to modify our familiar CSP algorithms to account for such soft constraints.

Our final product ended up being an adaptation of the min-conflicts algorithm, as discussed by Russel and Norvig (2011) [3]. As we will discuss later in the paper, this algorithm gives us the largest opportunity to make use of our soft-constraints parameter while still maintaining a sufficient solving speed. Zweben et al (1992) [4] take an extensive look at various procedures that can be used within min-conflicts, such as iterative repair, simulated annealing, and random permutations. Many of these algorithms performed extensively well in certain constraint satisfaction problems, however they either required a consistent understanding of how variables and constraints are related to one another or didn't fully make use of the extensive information that was provided when our program was run.

Thus, in the creation of a general purpose CSP, we made the decision to utilize a greedy version as it allowed us the most flexibility in terms of designing our music chord variables and constraints. Many of the published algorithms dealt with solving a specific type of problem efficiently, so in tackling our solving algorithm we started from scratch. We were creating a CSP solver in the broadest sense of the term. There would be no predefined variable, domain, or constraints. All of these elements would be initialized as the program was run. Our job was to then to create an efficient and holistic algorithm that would solve the problem regardless of what was inputted.

2 Background and Related Work

The reason we were inspired to do this project in particular was due to this question being a common (and remarkably tedious) homework assignment among introductory music theory courses. While these problems can be solved by a human in a reasonable amount of time, it is quite difficult for most students to follow every rule of harmony all of the time. With one constraint alone - the avoidance of ”forbidden parallel motion” - one must check each pair of notes in each pair of chords to see whether two voices are moving in the same direction in one of these forbidden manners. Ultimately even checking to see whether the harmony you've written follows the rules is difficult and tedious work for a human. Throughout these assignments, we would often think, ”This is a job for a computer.”

One of the sources of inspiration was the Emily Howell project. Stanford Professors Selfridge-Field and Sapp (2016) [2] discuss the project extensively. Emily Howell is an artificial intelligence that composes original music that is indistinguishable from authentically created music in a blind test. David Cope, Emily's creator, cites his effort to fit in with algorithmic composition as a source of inspiration for his software [1]. We used the success of Emily Howell and it's inspiration as a proof of concept for our project. Unfortunately, as Selfridge-Field and Sapp explain the generative algorithms for the software aren't publicly available, and take different approaches than the focus of our project.

3 Problem Specification

Given a key and a chord progression from the user, we sought to make a program to produce notes for a four-part chorus to sing that follow the basic rules of four-part harmony. This means giving each singer a note to sing at each time step while avoiding the pitfalls of bad voice-leading. These pitfalls include:

- Ensuring the soprano and the alto voices are within an octave of each other.
- Ensuring the alto and the tenor parts are within an octave of each other.
- Avoiding parallel fifths and octaves, which occur when two voices are a fifth/octave apart, and remain so in the next timestep.
- Ensuring that the seventh is always immediately resolved to the root.

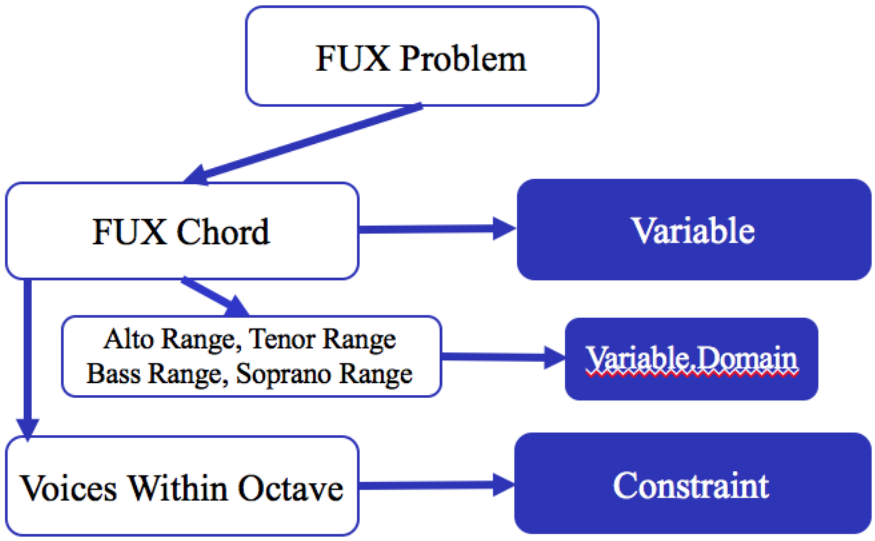
If any one of these constraints is not met, the music is bad for lots of reasons music theorists care about, but I suspect the details of why such things are bad is beside the point of this report.

As output, we give both a list of the chords and the notes used within them, which could be taken by another application and turned into sheet music, and we also give a midi file of what the final product sounds like.

4 Approach

The first task for our project was defining the classes in our CSP. Our deliberation led to four separate classes eventually simplified to three (another fourth class was added as we reached our

goal of soft constraints). We had our first three classes which were the Problem, Variables, and Constraints. Each Problem contained a hash map of Variables, and an array of Constraints. Each Variable had an id used for identification and an array who's elements represented it's domain. Each Constraint contained a list of variable id's that signified which variables the constraint applied to and an executable block of code, that represented the actual constraint.



The fourth class which was eventually moved to the Problem class was the Solver, which simply contained our solving algorithm. Our solving algorithm evolved as such: We started with a simple backtrack search solution which for solvable problems would come to a conclusion 100% of the time, albeit very slowly. In order to deal with larger problems efficiently, we implemented a forward checking algorithm which, in a time-efficient manner, allowed us to test on large problems similar in scope to our FUX problem.

Algorithm 1 Backtrack Algorithm

procedure BACKTRACK(*assignments*)
  if current *assignments* is a solution of CSP **return** *assignments*
  var → next unassigned variable
  for *val* in var's domain
  → *assignments*[*var*] = *val*
  →**if** *valid*(*assignments*)
  →→ *solution* = *backtrack*(*assignments*)
  →→ **return** *solution*
  return *False*
end procedure

Still this wasn't the most ideal situation to apply our soft constraints algorithm. If we chose to apply soft constraints every time we pruned, there would be a chance we would prune all the valid solutions, and not know for a long time. The same would occur if we applied soft-constraints as we selected the next available var. These problems were generally mitigated with the implementation of min-conflicts. Originally we solved min-conflicts with random permutations, but

Algorithm 2 Forward Checking Algorithm

procedure FORWARDCHECK(*assignments*)
vars in unsasigned variables
→**for** *constraint* in applying constraints
→→ **for** *val* in *var's* domain
→→→ **if** *val* doesn't satisfy *constraint* then $prune(var)$
end procedure

after switching to the most-constrained value we saw much better results. All this was done using a greedy algorithm that optimized the least constraining value.

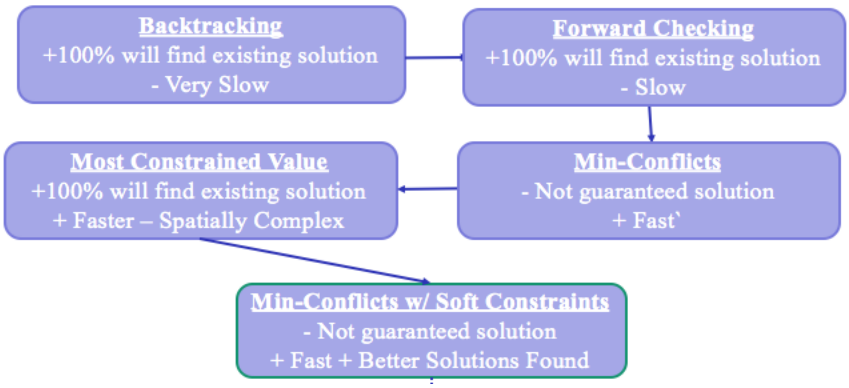
Algorithm 3 Min Conflicts Algorithm

procedure MINCONFLCITS(*assignments*)
if *assignments* is a solution of csp **return** *assignments*
var → a randomly chosen variable
if *var* is null **return** false
$assignments[var] = val$ → the *val* for *var* that minimizes conflicts
return $minconflicts(assignments)$
end procedure

Algorithm 4 Most Constrained Variable Algorithm

procedure MOSTCONSTRAINEDVAR(*assignments*)
$MostConflictedVars$ → empty array
$max\_conflicts$ → 0
for *var* in *assignments*
→ $conflicts$ → count applying constraints
→ **if** $conflicts$ == $max\_conflicts$
→→ $MostConflictedVars$ append *var*
→ **if** $conflicts$ greater than $max\_conflicts$
→→ $MostConflictedVars$ = $[var]$
return $MostConflictedVars$
end procedure

As mentioned earlier, several of the procedures discussed by Zweben et al were rendered subpar for the final implementation of our general purpose algorithm. For instance, Yuval Baror (2016), discusses the process in which iterative repair solves n-queens. Although it is less spatially complex, the time cost for solving a large, randomly assigned CSP were unreasonable and often suffered from a stack overflow. A similar circumstance occurred for simulated annealing. Thus, although the procedures were evaluated for the final implementation of our solver, they weren't tested as extensively.



5 Experiments

We have two main algorithms to test: backtracking with forward-checking and min-conflicts with soft-constraints.

Initially, we found that running backtracking search on our problem always resulted in similar results. The domain of each chord was initially ordered - that is, chord values with lower notes occurred earlier in our domain list than chord values with higher-pitched notes. This resulted in solutions that had a couple of properties.

1. They were deterministic. If you inputted the same key and the same chord progression, then you got the same result as an output.
2. They were low. Our solver was always able to find solutions that involves all four parts singing close to the bottom of their ranges. Examples of this can be heard in ordered_backtracking(1-3).midi.
3. Certain chord progressions are impossible. Namely those in which there is a chord with a seventh followed by a chord without a root.

Point (3) is an inescapable reality of the problem we are working on. Many chord progressions simply cannot be made into a valid four-part harmony. All of the failures we found were due to having chords that contain a leading tone (3, 5, 7) followed by a chord that did not contain the root (the ones that do contain the root tone are 1, 4, and 6). So if you type in a random chord progression, there is a chance that the program will return ”false” instead of the music you requested. It’s not the fault of the algorithm, it is literally an impossible chord progression to complete in a valid manner. A more intelligent program might loosen some of the hard constraints of four-part harmony and try to find a partial solution, but we decided to remain as strict as Fux in this case.

Point (1) led us to randomize the ordering of our domain before running our solver, to get more interesting variation in the music our program produced. Examples of this can be heard in unordered_backtracking(1-3).midi. We prefer listening to these, as they are usually better spaced and prettier, with a more differentiated soprano part, which gives rise to a more differentiated melody. As such, randomizing the order of the domains happens automatically when you call our program from the command line.

Point (2) implies (without proving) a very important fact about our solution space: there exist many solutions to our problem. The constraints are not so constraining in general that solutions are rare - we almost always find a solution very early in our domain space. This implies that we may be able to use a local search algorithm to find solutions fairly quickly.

So we did, and the next thing we implemented was a min-conflicts algorithm. One thing we noticed immediately was that occasionally the algorithm stalls. It must be getting into deep ruts that never allow it to find a solution that satisfies all of our hard constraints. However, most of the time, if you give it a valid chord progression, it will finish if you run it more than once - the longer the chord progression, the more likely it was for min-conflicts to find itself in a rut it could not escape. For the short chord progression 1 4 5 1 in the key of C, our min-conflicts algorithm found a solution in 24 out of 30 trials. For the longer chord progression of 1 4 5 1 6 4 5 1, min-conflicts succeeded in only 20 out of 30 trials, a near doubling of the failure rate. For the excessively long progression 1 4 5 1 6 4 5 1 4 5 1 6 4 5 1, min-conflicts succeeded only 13 times out of 30 trials. Still, its ability to quickly (in less than a few seconds) find a solution makes it a compelling option.

Backtracking with forward-checking, however, remained a viable option time-wise even for the obscenely long chord progression given above. By repeating this progression, we can get even larger problems, and we found that backtracking search had no problem with these long chord progressions even with input sizes of 57 chords, a massive harmony that no teacher would ever assign a student as homework. As far as solving the problem of four-part harmony is concerned, backtracking search is all we need for all problem sizes that show up in the real world.

That being said, only min-conflicts was able to successfully incorporate our soft-constraints, favoring solutions that had them to those that did not. An example of min-conflicts at work can be found in min-conflicts.mid.

6 Discussion

Luckily we were able to find solutions, which means that we were able to successfully satisfy Fux's constraints, and produce music. To quote Matthew after he first listened to it "It's actually kind of beautiful and very long... It sounds like something a baroque composer would have written." Jarele had no opinion as he is musically illiterate.

Counterintuitively, the final implementation of our solving algorithm was less suitable than our initial one. While usually one wants to use a min-conflicts algorithm to find solutions to large problem sizes more quickly than backtracking, the unfortunate reality of this problem is that as the input size grows, the chance that min-conflicts hangs gets unacceptably larger and larger, and backtracking is fast enough to handle the input sizes anyone would ever care about.

The best takeaway from the project in terms of algorithms was the idea that optimization occurs not just with better and faster theoretical time and space complexities, but also in terms of how it relates to your project. For complicated problems with very large input sizes and very few solutions, backtracking might be much less suitable, as iterating through each variable's domain takes exponential time in the size of the domain. However, for music where solutions are plentiful and the sizes are small (to a computer), this concern isn't as great.

That being said, only min-conflicts easily lends itself toward the allowance of soft-constraints. If we want to include some of the rules of Fux that were never meant to be followed 100 percent of the time, we must then use a local-search algorithm that allows us to find "good" solutions, not just "a" solution.

If we were to improve our work, we'd want to make sure we can narrow these solutions down to a greater degree. The addition of a weighting mechanism to our soft constraints would definitely help. Fux himself, agreed that certain soft requirements were more valuable than others, and if we were able to represent that we'd have an objective way creating a better solution. Unfortunately the algorithm for combining weights while also testing to see if original constraints are satisfied is a bit more computationally expensive and we didn't have enough time to explore the option to the degree we wanted. In a final version of our work, we would have a separate intelligence that adjusted the weights depending on feedback of how good the created music is.

A System Description

First, acquire a version of Ruby 2.0 or higher. We developed in Ruby 2.3 and cannot guarantee that it will work on lower versions of ruby. We use one ruby gem, namely "midilib".

Once both of these are on your machine, run our program by typing into the command line "ruby main.rb ¡algorithm¿ ¡key¿ ¡chord progression¿", where algorithm is one of the characters b and m. b will initiate backtracking search, while m will initiate min-conflicts. "key" is a string, such as the character C or F, which the harmony will be in. The chord progression is then given as a list of integers from 1 to 7, indicating which chords you want to occur in which order. For example, if you wanted the chord progression from Pachelbel's Canon in the key of G, and you wanted to use backtracking search to find it, you would type: ruby main.rb b G 1 4 5 1

It is also possible to independently run the NewCSP.rb file. Test cases are described at the bottom of the file, where we declare a problem object, add variables with an id and domain, add constraints with variables and a block of code, and add soft constraints with similar parameters. Then we can execute problem.backtrack or problem.min_conflicts to find a solution. Feel free to add any type of variable/constraint combination!

B Group Makeup

Cooperatively - Both Matthew and Jarele contributed to the writing of this final report, and high-level implementation of each section of the project. Implementation design and strategies were discussed at a high-level so both partners were aware of what outputs we would expect from one another which allowed us to work independently on the code.

Matthew Deutsch - Matthew was in charge of implementing the specific implementation of the problem as a CSP. He wrote Note and Chord classes to be variables in the problem, wrote the specific Constraints that the problem ended up using, and tied it all together into one big instance of the Problem variable. He also implemented the midi output for the problem and the command line interface.

Jarele Soyinka - Jarele was in charge of implementing the general purpose solver. He held final decision on the class organization of the Problem, Variables, Constraints, and Solver classes and corresponding methods. In addition he coded each implementation of the CSP solving algorithms. As depicted in the evolution of the solver algorithm graph.

References

[1] Ryan Blistein. Triumph of the cyborg composer, 2016.

[2] Craig Stuart Sapp Eleanor Selfridge-Field. From experiments in music intelligence (emmy) to emily howell: The work of david cope. *CS275a*, Spring 2016, 2016.

[3] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2003.

[4] Monte Zweben, Eugene Davis, Brian Daun, and Michael J Deale. Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1588–1596, 1993.