Bringing AI Bach

Four-Part Voice Leading Generator from Bassline
Using CSPs, Hill-Climbing, and Backtracking DFS

Joanna Chung, Handong Park, Luran He

I. PURPOSE

For generations, audiences and musicians alike have enjoyed the works of classical composers, including those of the most well-known Baroque composer, Johann Sebastian Bach. Part of Bach's most famous repertoire includes the many chorales he wrote for four voices. However, we know that Bach's own chorale writing was mostly constrained by the voice-leading conventions of his day, which followed compositional rules for writing chorus pieces with four parts: bass, tenor, soprano, and alto. Knowing that these rules existed, we ask the main motivational question for our project – could an AI agent, equipped with the right set of rules and principles by which Bach and other baroque composers abided, create a beautiful Bach chorale for four voices if given a solo bassline?

II. DEVELOPING PROJECT PURPOSE AND SCOPE THROUGH RESEARCH

To develop the goals of our project and our project scope, we needed to conduct research to understand the problem and what work had been done on it already. Upon researching the problem of generating chorales specifically in the style of Bach, we found that work had been done by a project called DeepBach (Hadjeres and Pachet, n.p.), which had used Gibbs-like sampling and deep learning through neural nets to generate Bach-style chorales with great success (Vincent, n.p.). But we didn't find substantial work with implementing Bach-like chorale voicing through the use of constraints and constraint satisfaction, which was surprising, since we had enough background knowledge, every member of our team being an avid musician, to suggest that voice-leading principles were well-defined enough to be implemented as constraints.

Upon seeing what work had been done in the field, we continued to explore the musical possibilities that might be enabled by using constraints and constraint satisfaction, by researching the classical principles that governed Baroque voice-leading. In order to fully understand what principles were needed in order to create Bach-like works, we needed to seek out a foundational list of what those principles were and research the importance of each principle. We worked with Glee Club Resident Conductor Nathan Reiff and student composer Brandon Snyder to research the most important rules that governed Baroque composers' four-part choral writing. These principles were (Reiff, Snyder):

- At any time, the four voices' notes must together form a complete triad chord of three notes, within the key of the piece (C Major for our purposes).
- The root of each chord should be doubled (except in cases of inverted chords where another note must be doubled).
- Two voices may not move parallel to each other in octaves or in fifths.
- The chord progression must follow one of the predetermined possibilities allowed by classical musical theory.
- Our phrases should end with either a plagal (IV-I) or perfect (V-I) cadence, meaning that certain basslines that did not allow for such final cadences would have no solutions to begin with.
- The upper three voices must be properly spaced (within one octave of each other).
- Not all motions can be similar. Some oblique or contrary motion is required.
- There shouldn't be any excessively large jumps.
- Our voices must not be forced to extend outside of the typical voice range for each part.

For more details, a documentation of our interview with Brandon can be found in Appendix 4. Once we determined these rules through the interviews we conducted with these experts, we were able to determine what the scope of our final project should be and what the proper implementation of our project should look like.

III. REFERENCES

Hadjeres, Gaetan, and Pachet, Francois, et al. "DeepBach: a Steerable Model for Bach Chorales Generation." *34th International Conference on Machine Learning*. Volume 2 (17 Jun 2017). <https://arxiv.org/pdf/1612.01010.pdf>. 12 Dec 2017.

Reiff, Nathan. Personal Interview. 8 Nov 2017.

Snyder, Brandon. Personal Interview. 14 Nov 2017.

Vincent, James. "Can you tell the difference between Bach and RoboBach?" *The Verge,* 23 Dec 2016.<https://www.theverge.com/2016/12/23/14069382/ai-music-creativity-bach-deepbach-csl>. 8 Dec 2017.

IV. PROJECT SCOPE

After realizing that the principles that we needed in order to generate Bach-like chorales were fairly restrictive and well-defined (and checkable), we decided to generate our Bach-like chorales by encoding the principles as constraints in a Constraint Satisfaction Problem.

Given an initial bassline in C Major, we had a search space that included all possible ways to fill up the notes for the remaining three voices, which were the tenor, alto, and soprano. If we took each of our voice-leading principles and encoded them each as a single constraint, we could search all of the possible voicings to find a way of voicing all of the chords with a given bassline so as to not violate any of these constraints. Once we implemented each principle as a constraint, we wanted to find an efficient way of finding a voicing that solves the CSP. We wanted to assess our AI's performance by looking both at the aesthetic value of the solutions that were generated and the speed and accuracy with which we found results, knowing that there were a great deal of ways to voice each chord in a musical phrase and quite a few possible chord progressions for any given bassline.

Yet solving the CSP for a solution was not our only goal. Once we were able to find a solution for a given bassline, part of our project would also involve investigating whether heuristics could actually improve the solutions that we generated. As part of our work, we hoped to utilize stochastic hill-climbing (within the bounds of our constraints) on heuristics to minimize the distances between notes for the same voice and thus "smooth out" voice lines so that individual voices did not have to jump large distances between chords over the course of a chorale.

Finally, in order to fulfill our project goal of using AI to generate Bach-like chorales from just basslines, we also wanted to implement features in our programs that would allow users to not only generate four-part chorales from basslines, but also generate their related musical scores and audio outputs. Doing so would allow both musicians and general audiences alike to read our AI's music or listen to it, and compare our AI output to Bach's works.

V. ALGORITHMS AND DATA STRUCTURES USED

So how did we go about actually finding a solution, given a bassline? Well, we worked in Python to implement the following overall plan:

1. Use Backtracking Search to solve for a chord progression that can fit our bassline given our constraints on possible chord progressions. If neither is possible, inform the user of failure.
2. Use Backtracking Search to solve for a sequence of voiced chords, using the chord progression from step 1, that violates none of the voice-leading constraints we have implemented. If no solution found, inform the user of failure.
3. Return the solved chords, which we can then convert to a musical score and a MIDI audio file using a Python package called Abjad.

A. Backtracking Search

Backtracking search is the canonical approach to solving CSP's, as we discussed in class. We imagine that we're performing a depth-first search (DFS), but for each node we check to see if any constraints are being violated. If we find out that some constraints are violated, we know that there can be no goal state in any descendent of that node, so we backtrack early. Here's the process, in detail:

1. For each variable:
    a. For each possible filling of the variable:
        i. If constraints can be fulfilled:
            1. If the assignment completed, return the completed assignment.
            2. Otherwise, continue to the next variable.
        ii. If constraints cannot be fulfilled:
            1. Backtrack to the previous variable.
2. Return failure.

For our problem, we have two CSP's: Choosing the chords for every bassline note, and choosing how to fill in the upper voices given the chord and the bass note. Both CSP's have constraints that are primarily *local*, in that variables are primarily constrained by the values of nearby variables. As such, we chose to fill in the voicing from front to back, and we chose to fill in the chords from back to front only to conveniently specify our preference for ending in authentic or plagal cadences.

For the problem of filling in voicing, we try more preferential voice-doubling first as according to classical music theory, and proceed to less preferential voice-doubling if need be in order to generate a solution (if more preferential voice-doubling does not work).

B. Hill Climbing Search

In terms of implementing hill climbing to find solutions that had better aesthetic value by having voice lines that were "smoother" with less distance between notes, we used the following hill-climbing algorithm (given a preset limit on the number of repetitions to try):

1. Solve the CSP for the given bassline as before and store the results and the heuristic value for the initial solution.
2. Choose a random chord in the CSP to re-voice.
    a. Generate a possible re-voicing of the notes in the chord for the top three notes.
    b. Check whether replacing the old version of the chord with the re-voiced version yields a better heuristic value. (If not, throw it out and return to step 2.)

   c. Check only the constraints that involve the newly voiced chord. (If any constraints are violated, throw the new chord out and return to step 2.)
   d. If both checks succeed, replace the old solution with the new solution including the re-voiced chord, and repeat step 2 until the number of total repetitions reaches the preset limit.

This hill-climbing algorithm is roughly based on the hill-climbing methods we learned in class. The two aspects of the algorithm here that are unique are the following - first, we are climbing within our constraints for musical voice-leading after having solved the chord progression problem, rather than just trying any possible notes to voice. This allows us to avoid trying completely random voicing assignments that have no chance of working and save time, so that we're only varying voicings within chords that we already know will fit the bassline. Next, instead of checking all of the constraints over and over again for all chords, whenever we re-voice a single chord, we only check those constraints that our new chord is involved in. This is a major cost savings that allows us to try revoicing chords at a much faster pace than if we had to try every chord for every constraint each time we changed one chord.

As for our heuristic, we decided to calculate heuristics measuring "note distances" (the size of the intervals that voices have to jump between chords) for the upper three voice lines, and then try to minimize those distances. Namely, we ultimately decided to use the squared distance traveled by voices. (Using absolute distances proved to be less effective after some initial work, so we chose to focus on squared distances for the purposes of the project.) To calculate total squared distance, we did the following:

1. Start with a running total of 0 for total distance.
2. For each pair of two neighboring chords, do the following:
   a. Look at each pair of subsequent chords in a voicing.
   b. Calculate the interval that each voice has to jump between the two chords.
   c. Add the square of the size of the interval to the running total.
3. When we run out of pairs, return the final running total.

This heuristic was our way to measure how "smooth" each voice line would be, where a smoother voice line would have less big jumps and move in smaller-sized intervals between chords.

VI. ANALYSIS, EVALUATION AND CRITIQUE

In order to test how well our Baroque AI would do, we decided to test its abilities on basslines that Bach himself used in his chorales. We repeatedly tried different basslines to see how our

CSP solver gave similar or different results from that of Bach himself, using these results to gauge the kinds of solutions that our CSP would return as chorales. We utilized excerpts from Bach chorales which we transposed into C Major for testing.

First of all, in order to ensure that all the preliminary parts of our implementation were working, we did comprehensive unit testing on both the data structures we implemented to store notes and the constraints that we implemented to represent the musical voice-leading rules that our four-part voicing solutions had to satisfy. Using Python's unittest package, in 'unit_tests.py', we made unit tests that tested the fundamental parts of both of these aspects in order to ensure that all of work would generate correct solutions to voice-leading problems.

In terms of results, for all of the 6 to 15-note basslines that we created by transposing and excerpting from Bach (limiting ourselves only to cases where our basslines would allow for either a plagal or perfect cadence at the end of the phrase), our implementation successfully found a four-part voicing that would work for each bassline. Here is one example of a side-by-side comparison of Bach's version (condensed by removing non-chord tones and other rhythmic gestures):

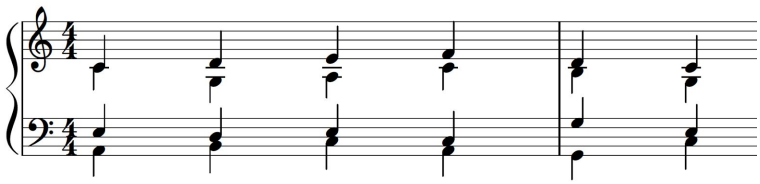


Figure 1: Bach's voicing, excerpted, transposed, and condensed from BWV 318

Here is the voicing that our CSP implementation generated:

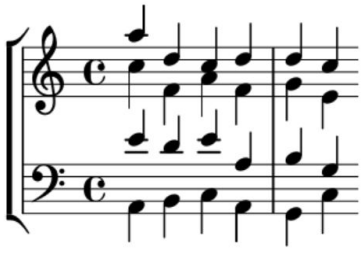


Figure 2: Our initial CSP solution given Bach's bassline from above - some jumps within voice lines.

While the above solution was relatively smooth, sometimes we would get lots of jumping between chords even though we implemented an extra constraint that eliminated any jumps that were greater than a fifth for any line. As a result, we also tested our hill-climbing algorithm on the same basslines that we used to test our initial CSP solutions. Here's an example of the result

of hill-climbing for 1000 tries on minimizing squared distances after starting from the initial voicing above:

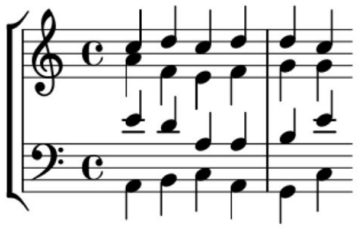


Figure 3: Hill-climbing for 1000 repetitions on total squared distance actually makes the lines smoother!

In terms of the performance of our hill-climbing algorithm, we found that from time to time (as in the above case) it generally was able to smooth out voice lines compared to our initial CSP solutions by a small amount. Numerically speaking, for most runs of our hill-climbing algorithm using squared distance as our heuristic function, the algorithm reduced the total distances that individual voices had to jump from note to note. However, in certain cases where our original CSP solution had smooth voice lines already, hill-climbing didn't do anything to change our solutions (although it was effective in cases where our original CSP had jumpy voice lines).

Similarly, we found that hill-climbing didn't always do that much to aesthetically improve the solutions we obtained through just solving the CSP, partly because our CSP solutions were already pleasing to begin with. Even though our heuristic minimized note distances on paper, the results that we found through hill-climbing had relatively similar aesthetic appeal to our initial solutions, with many cases where the results of hill-climbing were only negligibly smoother than our initial results. (We also had similar difficulties when we used other heuristics, such as total absolute distance rather than squared distance between notes.) Given more time, we would try to develop other heuristics that could further aid our hill-climbing search in finding more aesthetically pleasing voicings and smoother voice lines.

One of the best parts of our CSP solver through backtracking search was that it was able to run in a quick and efficient manner. We had feared that due to the sheer size of the search space in which we were looking for solutions, with a branching factor of around 200 and thus on the order of 10^{18} solutions for a typical 8-note bassline, we might have trouble finding working voice-leading solutions in a timely manner.

Yet it turned out that given a 6-15 note bassline, our implementation could find a working solution (if it existed) in around 10-20 milliseconds on average. To find this, we ran our implementation on 6 different basslines through 6000 total trials, and used the total time it took to run 6000 trials to find the average time that it took to find any single one of the 6000 trial

solutions. Repeating this process several times gave us average times that ranged between 10 and 20 milliseconds per solution, which was very fast. (We had average trial times of 10.2, 12.4, 12.7, 12.8, and 17.5 milliseconds per voicing after running "timings.py" five times, with 6000 trials each.)

In addition, our hill-climbing algorithm was also relatively quick and efficient. We set our hill-climber to check 1000 random re-voicings for each bassline, and had it try 60 basslines (10 tries for each of the 6 basslines). It took an average of 94 milliseconds for each total hill-climbing run for a given bassline, and around 0.1 milliseconds for each individual revoicing attempt. The key reduction in time spent trying each individual revoicing was due to the fact that with each revoicing attempt, we only check the constraints that involve the proposed new chord (and not all chords). Thus, even though we have to calculate the heuristic function for each new voicing attempt, we save more time on checking each voicing compared to the initial solving of the problem, rather than solving the whole problem over again every single time - which turns out to be an effective strategy as shown by the decreased time to try each new attempted voicing.

Thus, as it turned out, the problem was much easier to solve quickly than we had initially anticipated (even if hill-climbing wasn't as aesthetically successful). While we had worried that finding solutions to the problem would be difficult, it turned out that we didn't need to backtrack as much during our search as we might have expected. We found (as evidenced by just how quickly our searches ran) that backtracking is simply not required very frequently because the constraints aren't very constraining; this is great considering that the constraints produce results that are aesthetically pleasing and fit fundamental Baroque voice-leading principles. And our stripped-down constraint checking for hill-climbing also allowed us to also hill-climb for smoother voice lines quickly (even if they weren't as aesthetically exciting as hoped). See Appendix 3 for the rest of our generated scores, which form the rest of our results of testing!

In conclusion, by implementing these voice-leading principles as a CSP using backtracking search rather than through other methods, we were able to quickly find solutions for inputted basslines. While hill-climbing on a distance heuristic was less effective in improving the aesthetic quality of our chorales (partly because the initial solutions were already quite good), the effectiveness of our CSP solver showed us that encoding Baroque voice-leading principles as constraints in a CSP is a strong and effective way to generate surprisingly aesthetic chorales from a bassline.

APPENDIX 1: RUNNING OUR IMPLEMENTATION

Our code can be found and downloaded from this public Github repository:
https://github.com/luranhe/cs182-project/

Please be sure to download or git clone the entire repository so that all of the below files are available in the same folder where the code is being run. In addition, please read all of the instructions included below in this appendix before running.

Here is a description of the files that comprise the repository - please keep them all in one folder in order to run our code.

constraints.py	Coding of the music compositional rules and other subjective constraints according to the Western music tradition in the Baroque era.
examples.py	Excerpts transcribed from Bach chorales which can be used to give examples of generated solutions. (One example was made up by us, and the other five are condensed, transposed, and excerpted from real Bach chorales.) These were found from Peter Billam's "Forty Bach Chorales" which can be accessed at: www.pjb.com.au/mus/arr/us/satb_chorales.pdf
hill.py	Hill-climbing search with heuristic and random perturbations, as described in the above writeup.
prelims.py	Generates the preliminary bank of chords and chord combinations/inversions for each bass note. Contains our implementations of basic data structures for individual music notes and basic functionality involving notes.
score_generator.py	The user interface file which can be run in Python (once the repository is cloned so that all of the other files are also present in the same folder) to start up a UI that will allow the user to generate PDF files of musical scores and MIDI audio files for any of the six example basslines that we've preset. Ex. "python score_generator.py" **NOTE: In order to run this file, you must have the Abjad package installed along with all of its dependencies in order to generate the PDF and MIDI files. See below instructions and/or http://abjad.mbrsi.org/installation.html#install-abjad for details. (You may also need viewers**

	for PDF and MIDI files, as well as XServer for remote clients.)
search.py	Uses backtracking DFS to search through the preliminary chord bank and test against the constraints until a solution which doesn't violate any constraints is found.
timings.py	A file to time processing/search speed of how quickly our CSP solver finds four-part voicing solutions, as well as how quickly our hill-climbing algorithm can search for solutions that decrease the total distance that voices travel. Takes around five minutes to run, and outputs the total time for testing and average time to find each solution in both strategies. **NOTE: In order to run this file, you must have TQDM installed, which is a progress-tracking package in Python. See https://github.com/tqdm/tqdm for details.**
unit_tests.py	Running this file in Python will run the comprehensive unit tests we programmed that test our implementations of prelims.py (data structures and chord-generation functions) and constraints.py (voice-leading rules).

In order to be able to generate PDF scores and audio files, please follow the step-by-step instructions here to download Abjad, as well as all of the dependencies listed: **http://abjad.mbrsi.org/installation.html#install-abjad**
Note that without Abjad and all of these dependencies (such as LilyPond), one cannot generate any of the PDF or audio files that we successfully created through our work.
If on a Linux subsystem or SSH'ing in (any remote client), please ensure that XMing or some similar support is on to see any graphical outputs (the PDF and audio files) that are generated by our work.

In order to run the user interface, please run **score_generator.py** in Python (ensuring that the file is located in the same folder as all of the other files listed above). **Again, this can only be run after the Abjad installation instructions (and dependency installations) are completed.**
Running score_generator.py should start up a user interface, which first prompts the user for their choice of bassline, and then prompts the user to choose whether to hill-climb or not. If not

hill-climbing, then one PDF score should be generated and then one MIDI file should be generated. If hill-climbing, one PDF score and MIDI file should be generated for the initial solution, and then one PDF and MIDI should be generated for the new solution after hill-climbing was completed. Note that you might have to close the PDF windows and MIDI file windows to have score_generator.py proceed to each following step.

If you wish to run timings.py to also try our timing tests for our algorithms, make sure you've installed TQDM (which is a progress indicator package for Python). Instructions on that can be found here: **https://github.com/tqdm/tqdm**

Finally, note that since our chord progression finder is non-deterministic and our hill-climbing method is also non-deterministic, the solutions generated may vary each time our algorithms are run and may vary from the results we obtained above in our report.

APPENDIX 2: CODE & DISTRIBUTION OF RESPONSIBILITIES

Again, our entire public Github repository for the project can be found at https://github.com/luranhe/cs182-project/

All three of our group members were significantly throughout each part of the project, but we each took leadership of specific parts.

At the initial research phase, Handong and Luran took initiative to interview the Resident Conductor of Glee Club and learned a lot of insights about the approach and development of western music, while Joanna led the interview with Brandon Snyder, who provided most of the music compositional rules and methodology for note-selection that we have followed in the project.

In the problem environment development phase, Luran led development of the note class and building the "ingredient" notes for each acceptable chord given a bassline note. Joanna then led development for voicing the different notes in the acceptable octave range, and creating a generating function that branches all possible iterations of the "ingredient" notes into possible chords. Joanna led the midpoint check-in report.

In the constraint satisfaction problem implementation, Luran and Handong shared responsibility for coding different types of constraints, Luran led the development of a backtracking "n-step look-ahead" approach to mitigate for the large branching factor that could take formidable computational power, and he wrote the main executable file that integrated the different generators and constraints that would produce an output solution. Handong then transcribed,

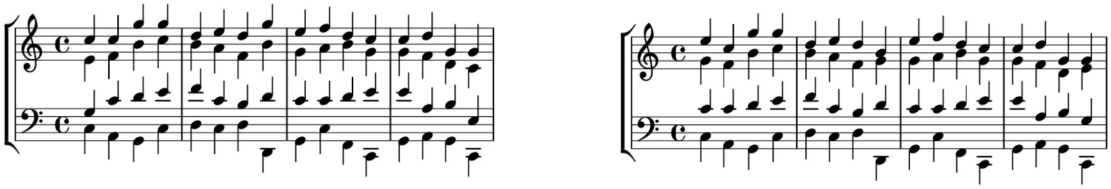
transposed, and condensed some emblematic Bach chorales and developed tests based on the different basslines. Joanna led programming for how to integrate the generated solutions into a form that can be seen and heard for testing purposes, programming the outputs to be compatible with Abjad and Lilypond, and having automatic generation of a pdf score and an audio file. Handong took leadership of creating the poster for the fair.

Handong then led the hill-climbing portion of the project and worked on developing comprehensive unit testing to ensure accurate functionality for the data structures for notes and functionality for generating inversions and checking all constraints. Luran created the user interface that users could use to choose which bassline they would like to use to try generating a solution, along with which algorithm (just solving the CSP or running hill-climbing) they would like to use. Joanna and Handong led the writing of the final report.

APPENDIX 3: RESULTS, CONTINUED (REST OF SCORES GENERATED)

For these results, each bassline was transposed and excerpted from a Bach chorale (referenced above each pair of results). Initial CSP solutions are on the left, with hill-climbing results on the right.

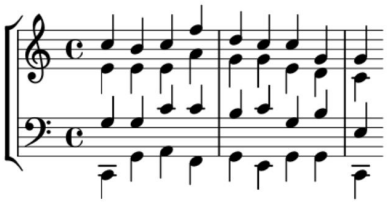
BWV 43: Ermuntre dich, mein schwacher Geist

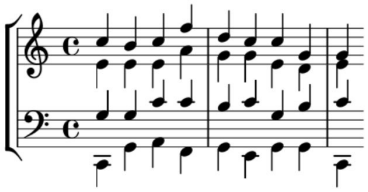


BWV 376: Lobt Gott, ihr Christen allzugleich

BWV 248: Von Himmel hoch da komm ich her

BWV 414: Uns ist ein Kindlein heut geborn





APPENDIX 4: SOURCE NOTES FROM INTERVIEW WITH BRANDON SNYDER

One Beginning Method for Harmonizing a Melody

Brandon Lincoln Snyder branlsnyder@gmail.com

The following is an outline of the problems and solutions for harmonizing a given melody with chords following basic functional harmony from the Western Classical tradition. Specifically, this outline will address melodies which are diatonic (remains within the major scale), and whose individual notes are all of the same rhythmic value. The harmonies utilized will be basic three-note triads, all remaining in the major scale. Once these solutions are in place in your AI, extending beyond the limits of the underlined terms, to more nuanced musicking is only a matter of adding nuance to your code.

Three Problems to Solve:

1. How to correspond the melody notes to chords.
2. How to create a sequence of chords that is “functional” (follows the conventions of Western Classical Music).
3. How to conventionally voice those chords. In other words, how to translate the idea of a “C major chord” to three harmonizing pitches.

A. Understanding Problem 1

The melody comes from a major scale:

· Among the twelve unique pitches available in Western European harmony, there are seven that create a major scale.
· We will consider the degrees of the scale numerically on a mod7 system, in Arabic numerals.
· In a C major scale, the note C will be labelled “1”

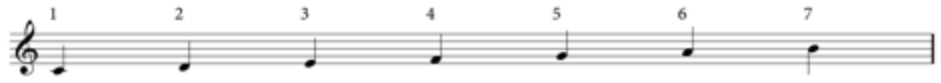


figure 1: Scale-degrees are represented in Arabic numerals

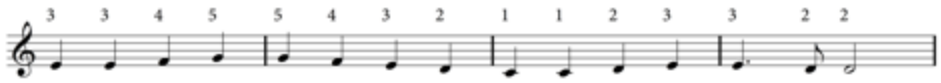


figure 2: Ode to Joy, labeled with scale-degrees

Chords = Harmony ; harmony comes from the major scale

- We will define a chord as a set of 3 notes in which:
  - Chord = [note , note + 2 , note + 4]
    - Remember, Arabic numerals denote scale degrees.
    - "C chord" = [1 , 3 , 5]
    - "D chord" = [2 , 4 , 6]
    - "B chord" = [7 , 2 , 4]
- <u>We will also consider chords in a given major scale in a mod7 system, but in roman numerals</u>

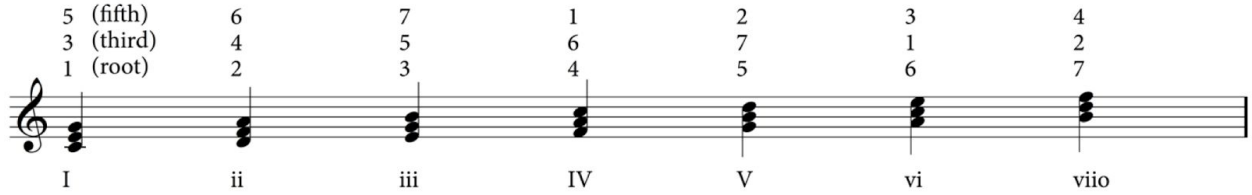


figure 3: Chords, labeled in roman numerals, consist of the root, third, and fifth.

With this knowledge, the solution to problem 1 is:

Any note that is a subset of a chord may be harmonized with that chord.

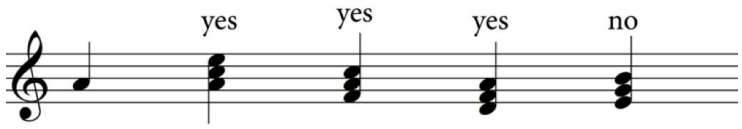


figure 4: the note 'A' cannot be harmonized with the chord [E,G,B] because 'A' is not an element in that chord's set.

B. Understanding Problem 2

Cadences

- A cadence is a short sequence of chords that resolve the harmonic and melodic tension.
- Your chord sequence should end with a cadence.
- Your chord sequence may also have cadences in the middle.

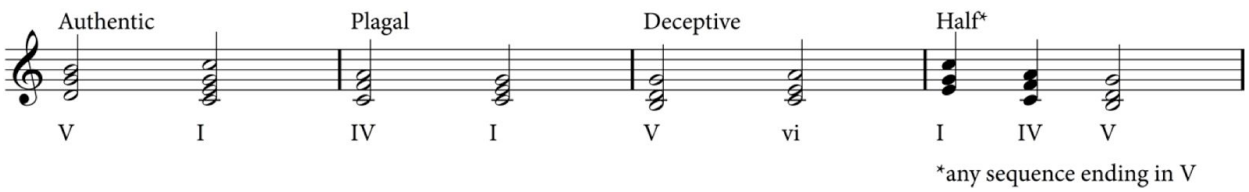


figure 5: 4 common ways a harmonic sequence is resolved. Note, a half cadence is simply the unresolved V chord. Any chord (within the rules of sequence) may precede it.

Sequence

- Your chord sequence must follow this diagram.

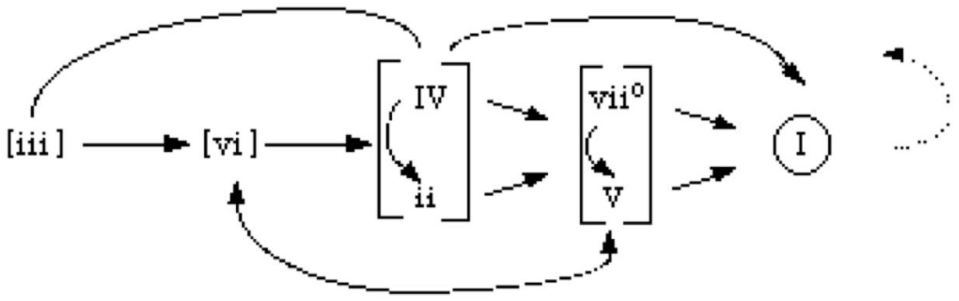


figure 6: Notice how all chords move towards I, and that it is very common for IV and ii (the first bracket) to move towards the vii and V (second bracket).

With this knowledge, the solution to problem 2 is:
Your sequence of chords must follow the proper sequence rules, as well as end in a cadence.

Additional Clarifiers:

- A chord may be repeated/sustained over 2+ melody notes.
- Given the rules of sequence allow, any given note can be harmonized in three different ways.
- There are many different ways to harmonize the same melody.

C. Understanding Problem 3

How to turn roman numerals (chords) into actual notes (aka, 4-part writing)

- You will need to make an instrument that can produce 4 tones at once. To actualize a chord and melody note, 4 distinct notes must sound at once.
    - These 4 voices will be labeled S A T and B (like a choir)

Roles for S A T and B: *think of each voice as its own melody!*

- S: melody. this is the first voice to be laid down.
- B: bassline. This is the second voice to be laid down. It is usually the root note

    - The bass can be a note other than the root note if it creates melodic contour in the bass voice.
- A and T: These two voices fill in the pitches of the chord not represented by the melody.

Voice Leading

- With 4 voices, and a 3-note chord, you will need to double one of the 3 notes. For now, always double the root note.
- When moving from chord to chord, A and T should travel as little intervallic distance as possible.
    - This will mean that chords are not always "stacked" in the same order
- When moving between chords, there should be no *parallel octaves, parallel fifths, or parallel unisons*.
    - A parallel fifth is when two voices have a spacing of 5 notes between each other for consecutive chords. Parallel octaves are the same idea, but when the voices are 8 notes apart. Parallel unisons, when the voices are 0 notes apart.

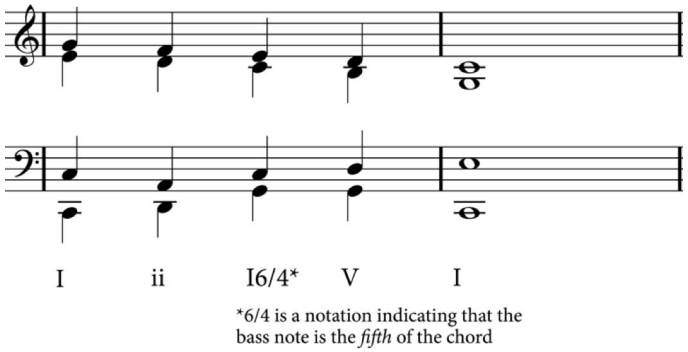


figure 7: Notice how each individual voice, S A T and B have a melodic contour. Large jumps between notes is minimized.

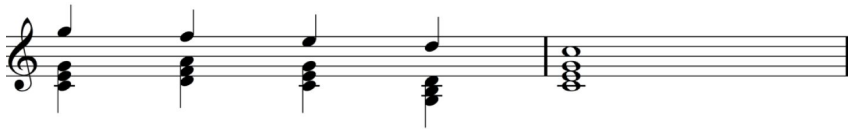


figure 8: This harmonization is bad for two reasons. 1) the B and A voices move in parallel fifths (i.e. they are 5 scale degrees apart for consecutive chords.) 2). The voices have huge jumps when moving from note to note. It is not 'melodic.'