Markov Decision in Tonal Music Generation

David Benoit
University of Massachusetts Lowell
Department of Computer Science
david_benoit@student.uml.edu

Brian Chiang
University of Massachusetts Lowell
Department of Computer Science
brian_chiang@student.uml.edu

ABSTRACT

Author Keywords
put author keywords here

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: Miscellaneous—*Optional sub-category*

INTRODUCTION
The objective of this project was to explore applications of Markov Decision Process (MDP) to the generation of tonal music. The system creates a Markovian state space based on a user-supplied harmonic progression, and applies an algorithm based on MDP and value iteration. The intended result is for the system to approximate the most suitable voicing of each chord in the progression, such that the created musical excerpt contains smooth melodic lines which interact with each other correctly based on a Schenkerian musical analysis of the excerpt.

Definitions
Definitions of terms used in this paper.

MDP

MDP State

MDP State Space

Factored MDP

Value Iteration

Music Theory

Schenkerian Musical Analysis/Theory

Harmony

Harmonic Progression

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2009, April 4 - 9, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-246-7/09/04...$5.00.

Chord Voicing

Voicing Progression

Melodic Lines

Stepwise Motion

Melodic Leaps

Tonic

Convolution Kernel/Matrix

PROJECT DESCRIPTION
The state space comprised of all possible arrangements of voicings in a chord progression can be represented as the cross product of the sets containing —-.

Code
This project was implemented using the Racket programming language. It's musical facilities were built using the RSound-Composer library, developed by David Benoit in parallel with this project. RSound-Composer is a library which provides a high-level, music-theory based API for Racket's RSound library, including data structures for use in music theory representation, and procedures which perform common operations on musical data. It also provides automatic scheduling and playback of created music. RSound is a sound library written by John Clements.

All of the code which implements the project-specific functionality for Markov Decision in Tonal Music Generation is contained within three files, mdp-state.rkt, mdp-reward.rkt, and mdp-main.rkt. All other files at the root of GitHub repository are either documents or legacy code (code which is no longer used by the project).

mdp-state.rkt

This file provides an implementation of Markovian states, state-spaces, and musical state functionality which is specific to this project.

<#struct>state: Represents a state. A state contains a single chord voicing, a reward/penalty value for use in value iteration, and the user supplied harmonic progression. In retrospect, storing harmonic progression data in every state was unnecessary, and with more time, the code could have

been refactored to reduce the system's memory footprint.

<#struct>part-range: Represents the range of notes a voice is able to play. Stores notes representing the highest and lowest notes that a particular voice has the ability to play.

<#procedure>voicing-compare: Legacy function which returns 0 if voicings are equal. This is no longer used is part of the project.

<#procedure>pitch: Creates a note with a duration of *null-beat*

<#procedure>enumerate-part-range: Takes a part-range structure and returns a list of all notes between the two end points.

<#procedure>part-range-valid-pitches: Takes a part-range structure and a harmony structure, and returns a list of all notes in the enumerated part-range which are also part of the given harmony.

<#procedure>populate-state-space: Takes a harmonic progression, a note structure representing the key of the progression, and list of part-ranges, and creates a Markovian state table. The table is arranged such that the x-direction rows represent the harmonic progression of the excerpt, and the y-direction columns represent individual possibilities for each harmony.

<#procedure>shuffle-state-space: Takes a state space, and shuffles the y-direction columns to ensure there are no implicit relationships present in the state table due to the algorithmic generation of the state space.

<#procedure>print-state-space: Takes a two dimensional list of states and prints it

<#explicit-data-value>example-part-range-list: A list of part-range structures representing the soprano, alto, tenor, and bass voices of a classical four part chorale.

mdp-reward.rkt

This file implements functionality for a Markovian reward function. The reward function provides reward points and penalty points based on a Schenkerian Musical analysis of the given progression of voicings. There was not enough time to implement all aspects of Schenkerian analysis that were initially planned, however, enough functionality was implemented that the reward function provides a partially accurate measure of suitability of a chord voicing in its given context.

<#procedure>convolution-matrix: A constructor which takes a progression of chord voicings and the index of an individual voicing, and returns a three-column convolution matrix consisting whose leftmost column is a list of all notes in the chord preceding the indexed chord (or null if indexed chord is the first chord in the progression), whose center column is a list of notes in the indexed chord, and whose rightmost column is a list of all notes in the chord succeeding the indexed chord (or null if indexed chord is the final chord in the progression). The convolution matrices returned by this function are used by the reward function and its helpers.

<#procedure>map:* Applies a function to an arbitrarily long list of equal-length lists. This overcomes a limitation of Racket's built-in mapping procedure, which only has the ability to map over a fixed number of lists in any given expression.

<#procedure>mostly-stepwise?: A helper function of the main reward function, which checks to see if a segment of the voicing progression follows the rule of Schenkerian music theory that musical excerpts contain mostly step-wise motion. Takes a convolution matrix and returns false if total number of melodic skips in all melodic segments produced by individual voices in the matrix is greater than one. Returns true otherwise.

<#procedure>unresolved-skips?: A helper function of the main reward function, which checks to see if a segment of the voicing progression follows the rule of Schenkerian music theory that any skips in a musical excerpt are followed by stepwise motion in the opposite direction. Takes a convolution matrix and returns false if there exists a melodic segment in any voice in which there is a skip of an interval greater than a major 3rd (equivalent to the interval of C to E) which is not directly succeeded by step-wise motion in the direction opposite the skip.

<#procedure>soprano-proper-cadence?: A helper function of the main reward function, which checks to see if a segment of the voicing progression follows the rule of Schenkerian music theory that the melody of the highest voice in an excerpt must end on the tonic, and must progress step-wise to to the tonic (For example, given an excerpt in the key of C Major, the highest voice must either end with the notes D $\rightarrow$ C or B $\rightarrow$ C).

<#procedure>bass-begins-on-tonic?: A helper function of the main reward function, which checks to see if a segment of the voicing progression follows the rule of Schenkerian music theory that the first note played by the lowest voice is the tonic.

<#procedure>bass-proper-cadence?: A helper function of the main reward function, which checks to see if a segment of the voicing progression follows an approximate rule of Schenkerian music theory that the final note played by the lowest voice is the tonic. Schenkerian music theory is more sophisticated regarding the treatment of the final notes played by the lowest voice, but due to project time constraints, such an analysis was not possible to fully implement.

<#procedure>reward-function: This function takes a generated voicing progression, and updates the values of each state in the progression based on a partial Schenkerian analysis provided by its helper functions. To determine the value of each state, the reward function analyses the state's musi-

cal fitness in relation to the two states immediately adjacent to it in the progression. It does this by creating a convolution kernel based on the voicings of the three given states, calling helper functions on the convolution kernel to analyze the local states fitness in relation to each other, then moving the kernel to the next state and repeating the process. The result is that each state is given an updated value based on the context it is currently present in.

mdp-main.rkt

<#procedure>value-iteration: This function recursively carries out value iteration over a factored state space. Each recursive call corresponds to a complete iteration over the state space.

The function takes a state-table and a desired recursion depth. First, it checks to see that the length of all table columns is equal. It will trim any column which is longer than the shortest column.

Since the state table is represented as list of linked lists, each column in the table corresponds to a linked list whose members all have the same harmony. However, the system's reward function requires the voicings to be factored as chord progressions. This can be accomplished by reconstructing the table such that the linked lists correspond to the state-tables rows. Therefore, this function creates a temporary table, the iteration table, which is identical to the state table but the lists represent rows instead of columns.

For each row in the state table, this function passes the entire row of states to the reward function to be analyzed. Each state's value is updated based on its fitness in the context of the row it is currently in.

Once all rows have been passed to the reward function, a new state table is created from the iteration table, which is identical to the old state table but with updated Markov values. To decide which states will be compared to which others during the next call of value iteration, the new state table is re-factored. Each column in the state space is sorted such that the states with the highest reward values (or smallest penalty), rise to the top of the column. Groups of suitable states tend to rise together, because the reward function measures each states suitability based on its context in surrounding states.

Eventually, the top row of the state table converges to a single progression of voicings, where each state has the highest state value in the column. Suitability of voicings in the particular context of the top row is ensured, because voicings which are not suitable in the top row are penalized, and more suitable voicings take their place.

Therefore, the top row of the state table after many value iterations ultimately results in an approximate best progression of chord voicings.

<#procedure>best-path: This function takes a state table and returns a list represinting the top row, or approximate best voicing progression.

<#procedure>run-markov-system-trial: This function takes a harmonic progression and a key. It creates a factored state space, and displays the result of performing value iteration 0 times, 10 times, 100 times, and 1000 times.

<#procedure>play-markov-path: This takes a list of voicings and stream it as a musical audio excerpt.

For example:

```
(require "mdp-state.rkt")
(play-markov-path
  (best-path
    (value-iteration
      (map
        shuffle
        (populate-state-space
          progression1
          (note 'C 5 null-beat)
          example-part-range-list))
        100)))
```

Would create a shuffled state space, apply value iteration 100 times, and play the audio of the best voicing progression of progression1 in the key of C.

ANALYSIS OF RESULTS

The Markov Decision in Tonal Music system was tested several times using different data sets as input each time. The resulting output of the system is available to view at:

https://github.com/benoid/tonal-music-generator/tree/master/trial-results

The system was minimally successful at learning proper music composition techniques. Analysis of system output confirms that greater magnitudes of value iteration tend to increase the smoothness of the excerpt's melodic lines when compared to excerpts produced with lesser or no value iteration.

However, the system is unable to produce excerpts which follow all rules of classical music theory. Part of the reason for this is that some planned aspects of the reward function were unimplemented due to time constraints. A more critical issue is that even some aspects of the reward function that were implemented fully are not always learned by the system. This causes occasional abruptness in the cohesiveness of produced excerpts, despite an overall tendency toward becoming smoother.

DISCUSSION

There were many obstacles to creating this project, the largest of which was managing the size of the state space. Since specific context is important when determining the suitability of a chord voicing, voicings can not be treated as individual states the way most MDP systems treat them. As such the original design for the project called for each state to contain a current chord voicing, and a list of the sequence of

chord voicings leading up to the current voicing. The issue was that such a state space would have a memory footprint orders of magnitude greater than an average computer could store, and would take an incredibly long time to compute.

To remedy this problem, the system can not attempt a brute force approach like originally planned. Instead, chord voicings are treated as individual states, and the state space is factored into harmonic progressions. When value iteration is run, states are compared to adjacent states in their factored harmonic progressions using a technique based on kernel convolution. After each iteration of value iteration, the state space is re-factored, grouping states with the highest values together into their own progressions.

There are some ethical considerations to consider when discussing computer generated music. Is music still art if it is generated by an algorithm? Will the quality of computer composed music surpass the quality of human composed music? Will render human composers obsolete, crushing an industry?

CONCLUSION

ACKNOWLEDGMENTS

The work described in this paper was conducted as part of a Fall 2016 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin.

REFERENCES