Distributed Improvising Orchestra
Improv

Giulio Mecocci

Supervisor:	Dr. Robert Chatley
Second Marker:	Dr. Iain Phillips

June 16, 2015

Abstract

We present a distributed system of independent agents that cooperate to improvise music in real-time. Much literature exists on both distributed systems and algorithmic live music improvisation separately, but not on the two topics combined. In order to improvise music the system relies on Artificial Intelligence techniques such as `Case-based Reasoning`, `Genetic Algorithms` and `Artificial Neural Networks.` Agents communicate through the exchange of messages: the system relies entirely on a symbolic representation of music. While most existing systems rely on some sort of user input to start their improvisation process, Improv succeeds in performing completely autonomously. Our system is able to improvise music that involves multiple instruments, a feat that has not been attempted by many before. Improv also satisfies its real-time requirements: music is played back without any delay in systems with up to 10 musicians.

Acknowledgements

To my father, for his advice, guidance and inspiration throughout the entire course of my life so far.

To my mother, for her unconditional love and support: I could not have made it to the end of my degree without her.

To my brother, for just being a good brother.

To my supervisor, for all the useful insight he has provided and for being so available to answer questions, discuss ideas and review drafts.

To my friends for helping out with producing some of the data used in this project.

Contents

Chapter 1

Introduction

Feelings and emotions are traditionally believed to be something that only humans, not machines, can understand and convey. Music is one of the highest forms of conveying emotions, therefore, modeling its composition process in a way that allows a machine to perform it is a very difficult task.

Since Mozart's Musikalisches Würfelspiel (Musical Dice Game) [1], and maybe before, musicians and scientists have tried to find some recipe, an algorithm, able to generate music of comparable quality to that of a human composer.

Mozart's method involved a set of pre-composed measures of music which were selected at random by rolling dice and then pasted together to create a Minuet. Since the advent of computers, more sophisticated methods have been developed.

Much work has been done on counterpoint music generation and harmonisation: a melody is provided by a human and the machine will generate a suitable accompaniment. Most of the effort in algorithmic composition seems to have been devoted to offline composition, that is, a composition process that does not involve real-time improvisation.

A few systems able to improvise music have also been developed, however, most of them required a human to produce some musical input before they could generate their improvised solos, while others only improvised simple, single-instrument melodies.

The Princeton Laptop Orchestra [2] is a distributed systems for playing music; however, the system does not improvise music autonomously: human musicians play virtual instruments on their laptops, while the system takes care of synchronising the performance.

This project approaches the problem from a different angle: it draws inspiration from the way a group of human musicians getting together to improvise music interact with each other. We decided to model this as a distributed system of independent agents cooperating to improvise music. We named this system `Improv`.

A novel aspect of `Improv` is that it not only distributes the process of composing music, but also that of playing it: each musician in the system generates and plays music independently relying on communication to keep in sync with the other musicians. Another feature worth of mention is that each agent in the system can improvise music for different instruments. Most existing systems, on the other hand, improvise simple melodies or solos for a single instrument.

When analysing what happens when a few friends get together for a jamming session, we can identify three main aspects: each musician has a certain level of musical knowledge, including the knowledge of a few riffs[1] composed by some musical expert, musicians play together, following the same tempo in a synchronised fashion, and they listen to each other in order to decide what they are going to play next.

If we want to model this as a distributed system, we need to address four main issues: how to represent musical knowledge, how to generate music, how to simulate the act of listening and how to synchronise the different agents.

The representation of musical knowledge and the process of generating music are closely tied together, since we are trying to model a system in which agents take advantage of their prior knowledge to improvise new pieces of music. This can be addressed practically by building a symbolic musical knowledge-base from existing melodies composed by humans.

The act of listening to what the other agents are playing can be modeled by the exchange of messages containing information about what is being played. This is more suitable to the objectives of this project than relying on audio processing, since audio processing is very complex (it would probably deserve a project of its own), computationally intensive, and not so accurate.

One of the ideas at the core of this project is that if you consider a melody composed by a human musician and you divide it in sub-parts, useful information can be extracted on the relationship between those parts. More specifically, we assume that if you see a musical phrase `A` followed by a musical phrase `B` in the melody, `B` can be considered a good response to `A`. If you combine this with the assumption that good responses to phrase `A` are good candidate responses to a phrase similar to `A`, you lay the bases for the composition techniques adopted in this project and described in Section 5 and Section 6.

The question this project explores is whether it is possible for a completely autonomous system to improvise music that sounds good, or if this is still, for the time being, a task that only humans are able to accomplish.

[1]A brief musical phrase which can be used as a building block for improvisations

Chapter 2

Background

In this section we will review a few Machine Learning techniques and their application to algorithmic compositions; the strengths and weaknesses of each technique will be assessed.

2.1 Genetic Algorithms

2.1.1 Overview

Genetic algorithms are a family of algorithms that try to emulate the natural process of evolution and genetic mutation. These algorithms are usually adopted when trying to solve optimisation problems.

Information is encoded in a (usually fixed-size) `chromosome`, which can be represented, for example, as an array of bytes. Each candidate solution in the initial population is represented by a chromosome.

During each iteration of the algorithm, a subset of the population is chosen, using a `fitness function`, in order to "breed" a new generation. Pairs of chromosomes from this subset are selected as parents. Parents are mixed together using crossover and mutation to generate a new child.

The selection of parent pairs is then repeated until a certain population size is reached.

Crossover is the process of generating a new chromosome by reassembling sub-parts of the parents. While mutations are random changes in sub-parts of the newly generated chromosome. Mutations are usually used in order to preserve "genetic diversity" which should prevent the algorithm from getting stuck in local minima/maxima. See Figure 2.1 for an example.

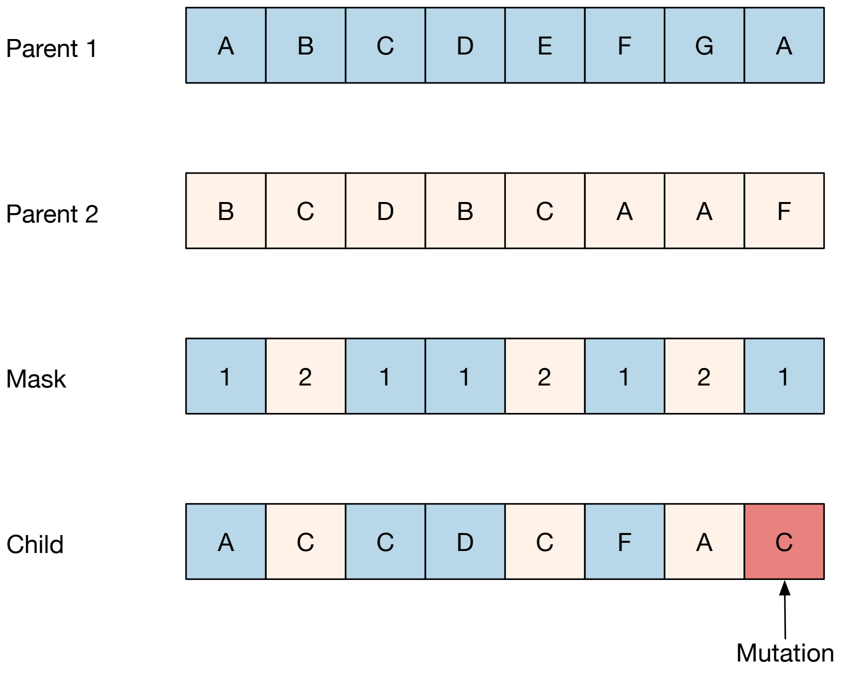


Figure 2.1: Crossover and Mutation

Common termination conditions include either the reaching of some `fitness` threshold or a fixed iteration count [3]

2.1.2 Application to Algorithmic Composition

Genetic algorithms, mainly due to their random component, are usually very suitable to the generation of original solutions to problems. However often it is very difficult to define an appropriate fitness function.

An analysis of two systems that have used genetic algorithms for the improvisation of jazz follows.

Yee-King proposes a system [4] that claims to be able to respond in real time to the sounds made by a human performer. In the paper he describes a system that analyses the rhythmic patterns and the note sequences played by a human performer in order to produce improvisations. Note sequences are stored in both a short and a long term memory. The short term memory is used by the system to analyse what the player is currently playing. The long term memory instead can be used by the system to select a permutation of what the player has played and play as part of its improvisation.

Another interesting system is GenJam [5], which uses genetic algorithms to improvise jazz solos, taking turns with a human performer. During a gig the human performer can use multiple pre-trained improvisers; the improvisers listen. One of the limitations of this system seems to be its fixed structure: it seems like the system needs a file that tells it what to do at each chorus (repetition), whether to rest, improvise or play the "head" (main tune). It also seems like the "head" of the tune is pre-written and pre-selected. It is not clear whether this fixed structure is a technical limitation or a necessity since also the human performer needs to know what the main tune is going to be in order to be able to improvise on it.

An interesting claim is that the representation scheme used by the system guarantees that “any sequence of 32 bits will decode to a measure of notes that are harmonically appropriate. In other words, GenJam can't play a theoretically wrong note.” In this system the fitness values used to assess populations of chords are derived from user input: a human listens to the system while it is improvising on a tune and press different keys on a keyboard according to whether they liked the piece or not.

Although, as the author pointed out, this kind of human-driven fitness function is the bottleneck of the system as a human can only process few audio outputs in a reasonable amount of time.

2.1.3 Strengths

Genetic algorithms are a good option for problems, like algorithmic composition, where there are multiple optimal solutions exist. On top of that, due to their random component, they usually perform well in tasks that involve a certain degree of creativity. If their fitness functions are well designed, they are also likely to produce novel, better solutions, starting from a population of good candidate solutions.

2.1.4 Weaknesses

The biggest problem of genetic algorithms is the difficulty in modelling a problem so that a genetic algorithm can be applied to it. Usually good chromosome designing is a hard task.

On top of that, the “if” in the last sentence of the previous paragraph should have been written in bold and underlined: designing a good fitness function is not straightforward it is not always clear how one should formalise what it means for a solution to be good. Take for example the problem of composing music: what does it mean, formally, for a melody to sound good? This question does not have an easy answer.

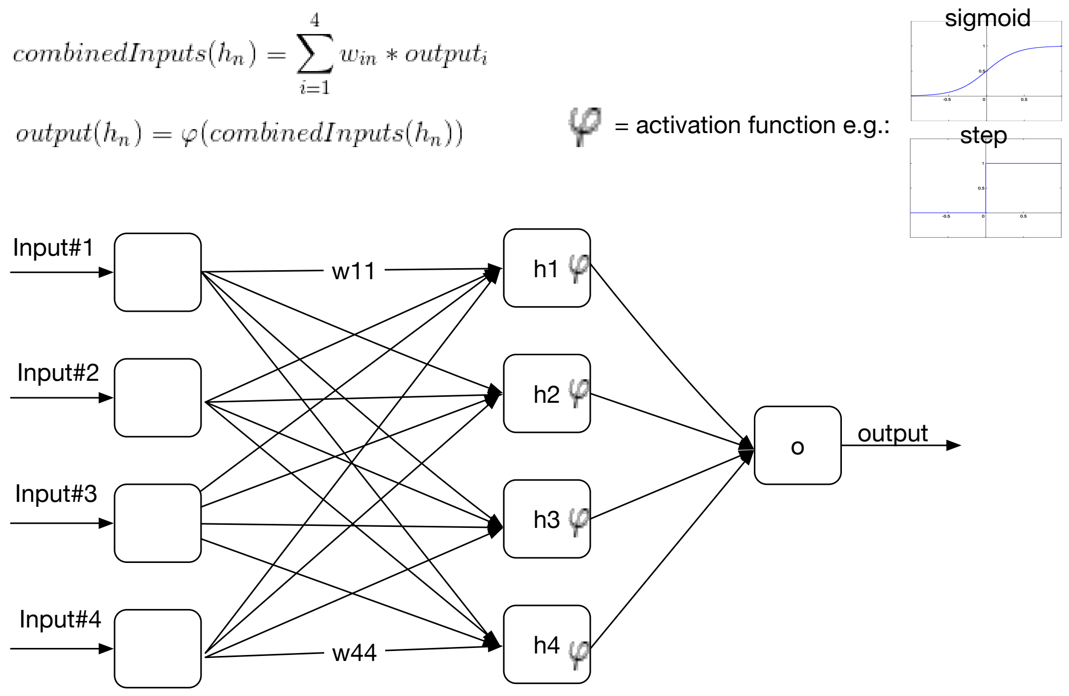
Finally, as mentioned before, for a genetic algorithm to work well it is necessary that its initial population contains mostly good candidate solutions: it is not always easy to produce such a population.

2.2 Artificial Neural Networks

2.2.1 Overview

Artificial Neural Networks (ANNs) are a family of statistical learning models that try to roughly imitate the way the human brain works.

ANNs can be thought of a network of interconnected nodes, known as “neurons”. Each neuron takes a set of inputs and produces an output. Weights can be applied to the inputs. Neurons are organised in layers where the outputs of the neurons of a layer are the inputs to the neurons of the next layer. There are three types of layers: the input layer, the hidden layers and the output layer. The input layer takes the inputs of the network and relay them to all the neurons in the first hidden layer. The hidden layers are where the “memory” of the network resides: by storing different weights for different inputs the determine how much the value of a given input affects the value of the output of the network. The output layer combines together the values from the last hidden layer and produces the output of the network.



$combinedInputs(h_n) = \sum_{i=1}^{4} w_{in} * output_i$
$output(h_n) = \varphi(combinedInputs(h_n))$
φ = activation function e.g.:
sigmoid
step
Input#1
Input#2
Input#3
Input#4
w11
w44
h1 φ
h2 φ
h3 φ
h4 φ
o
output

Figure 2.2: Artificial Neural Network - One Hidden Layer

The output of a neuron is the result of the application of an `activation function` to the weighted sum of its inputs. Common activation functions are linear functions and sigmoid functions.

Networks are generally trained by using annotated data-sets and algorithms like `backward propagation` that aim at minimising the error between the network's output and the ground truth by adjusting the network's weights [6].

2.2.2 Application to Algorithmic Composition

Artificial Neural Networks are typically used for classification or pattern recognition, but, with careful design, they can also be used to generate new patterns that share common characteristics with a training set.

For example [7] proposes a system where an Artificial Neural Network is used to learn the structure of music in a training set. Once the network is trained, new musical inputs can be provided to it in order to compose a new musical piece that will share the same structure of the melodies in the training set.

According to the author, the first aspect to be considered when trying to model a temporal process like music, is how to represent time. One of the common ways of representing time is that of sliding temporal windows of fixed size. This means that a piece of music is considered as a sequence of notes grouped in chunks of fixed time duration. However, the author not to divide time into windows of a fixed size, but to instead model a piece of music as a sequence of notes.

If time had been represented as a fixed-size sliding window, the system would have had to learn to produce sequences of blocks of notes. However, because of the chosen time representation, a sequential network, "which learns to produce a sequence of single notes rather than a set of notes simultaneously" was used.

The network leverages the hidden layers' memory of previous notes to produce the next note in the sequence by the means of a feedback loop where some of the neurons in the output layer are connected to some of the neurons in the input layer.

Notes are represented by their absolute pitch and duration, ignoring other factors as timber, loudness and tempo.

2.2.3 Strengths

Petri Toiviainen in [8] states that musical students learn the art of improvisation by example rather than by memorising rules. The students listen to the work of other musicians and infer from them implicit rules suitable for a given style of improvisation. Because of this, the author claims that rule-based systems are not very suitable for this task, while ANNs tend to be able to extract implicit structural characteristics from a corpus of examples.

2.2.4 Weaknesses

Toiviainen himself in [9] points out that neural networks have a few limitations. Among the limitations listed by the author, the following appear to be of greater importance: due to their complexity, ANNs have only been applied to "toy-sized", strongly-stylised problems. On top of that they work like black boxes: while an ANN might be able to capture the structure of a musical piece, understanding or explaining such structure at a higher level is no easy task[1]. Finally, while neural networks are capable of capturing the surface of the structure of a melody, they fail to infer higher-level features such as musical phrasing or tonality.

2.3 Markov Chains

2.3.1 Overview

Markov chains can be defined as a stochastic process that describes the transitions through a finite set of states [10] [11]. The next state is probabilistically selected according to some transition probability distribution over the current state and not the states that come before that.

By analysing a dataset, one can isolate a set of states and build a transition probability matrix between the states. Once such matrix is built, one can construct a Markov chain by picking one state and then iteratively selecting the next state based on the state transition probability distribution on the current state.

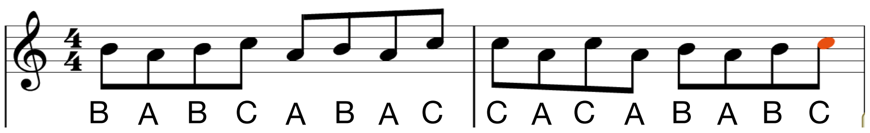
2.3.2 Application to Algorithmic Composition

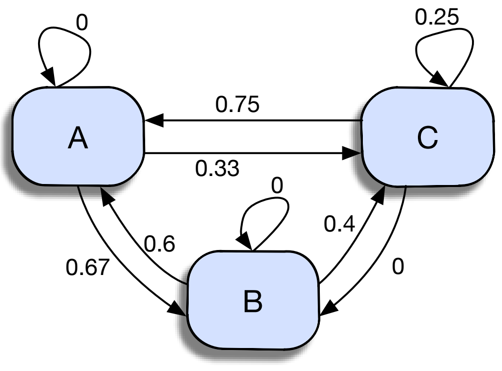
As we can see in Figure 2.3, Markov chains can be easily used in algorithmic composition: one can take a musical dataset and compute the transition probability between notes. Once that is done, generating a melody is just a matter of selecting note after note according to the transition probability matrix. In our example, if we were to start with the note `A`, we'd

[1] In the end neural networks are just a set of edges and weights, with little meaning associated to them

have a 67% chance of selecting B as our next note, but we would never select A (0% chance). If we selected C, instead, with a 33% chance, would then have a 25% of selecting C again.

Although this process is somewhat similar to that of Mozart's Dice Game [1], there is a fundamental difference between the two: Mozart's approach is completely random, whereas Markov chains are a stochastic process guided by a probability distribution over the states inferred from the training data. Put it in other words, Mozart was rolling a fair die, while an approach using Markov chains rolls a die that is biased towards state transitions that are more frequent in the training set: if number 3 on the die represents note B the probability of rolling a 3, unlike a fair die, change depending on the current note.





a->b	b->a	c->a
a->b	b->c	c->c
a->c	b->a	c->a
a->c	b->a	
a->b	b->c	
a->b		

From | To	A	B	C
A	0	0.67	0.33
B	0.6	0	0.4
C	0.75	0	0.25

Figure 2.3: State Transition Diagram and Matrix from Musical Score

Examples of applications of Markov chains to musical composition can be found in [12] [13].

More recent research focuses either on Hidden Markov Models [14], or on hybrid approaches [15] [16].

The authors of [15] propose a hybrid solution that uses Markov chains to generate inputs for a genetic algorithm. They use a Markov model to capture the transitional probability in the training corpus, using feature such as pitch, rhythm and timing information. Then they use Markov chains to generate an initial population which is then evolved using genetic algorithms. Since the initial population is generated through using a Markov model, which captures the characteristics of the training corpus, they claim that " [...] each individual in the population will resemble the user's implied musical style."

As mentioned above, [14] describes an approach based on Hidden Markov Models. Hidden Markov Models (HMMs) are generalizations of Markov chains. An HMM is basically a Markov chain with unobserved (hidden) states. In addition to computing a state transition probability distribution matrix, training an HMM also requires the computation of a probability distribution of observable outcomes for each hidden state. Once these matrices have been computed, it is possible to calculate the most likely sequence of states that produce a given sequence of outputs. One of the advantages of HMMs over simpler Markov chains is their ability to perform global optimisation over local optimisation [17]. Given their ability of inferring hidden state transition from observable state transitions, HHMs have been employed in the generation of musical accompaniment (counterpoint) [18].

2.3.3 Strengths

One of the main advantages of using Markov chains for algorithmic composition is their probabilistic nature: they manage to capture the structure of a training corpus, while at the same time preserving a level of "fuzziness". To say it in Jones' words [12]: "Finally, electronic and computer music is frequently criticized for sounding too fastidiously sterile, too regular, and hence too artificial. Stochastic techniques may be used to produced fuzzy edges and to "humanize" computer-generated sounds."

2.3.4 Weaknesses

According to [19], however, Markov chains have a big limitation: if probabilistic transitions between individual notes are considered, we get little meaningful information, yielding disappointing results. On the other hand, if transitions between bigger "chunks" of notes are considered (referred to in the paper as "higher order methods"), we get back fragments of the pieces in the training corpus, even entire exact repetitions. While exact repetitions of portions of a training corpus could sound good, they bear the risk of just rehashing, or even plainly play melodies present in the training corpus. If a generative process ends up playing back exactly the contents of the corpus on which it was trained, it could hardly be defined generative. Finally, if transitions between "chunks" of intermediate size are considered, we get back some new pieces that sound like the melodies in the training corpus, but also a much larger portion of ascending and descending scales.

2.4 Case-based Reasoning

`Case-based Reasoning` (`CBR`) is a machine learning technique that relies on the knowledge of solutions to existing problems to find solutions to new problems. The technique is inspired by the way humans usually approach problem solving: they base most of their reasoning in practical situations on their past experience.This technique, therefore, relies on a knowledge base where pairs of `(problem, solution)` are stored and indexed by `problem` for easy retrieval.

`Case-based Reasoning` can be summarised in four core steps:

- **Retrieve** ⇒
  In this step we a new problem is faced, the knowledge base is searched to find similar problems; usually the most similar problem is selected
- **Reuse** ⇒
  The solution to the most similar problem is adapted to the current situation
- **Revise** ⇒
  If the known solution does not work "out-of-the-box", it is slightly tweaked to work in the new situation

- ◇ **Retain ⇒**
  If the tweaked solution is deemed a good solution for the new situation, the current problem and this new solutions are added to the knowledge base

The terminology of `Case-based Reasoning` can be quite vague at times, it is therefore useful to consider an example to clarify things.

Imagine you have to change the wheels of a car, but you have never done it before. However, you have changed wheels to a motorbike and you remember how. This would be a pair of problem and solution in your knowledge base: the problem description could be something like `change(wheels).of(motorbike)”` and the solution could be represented as a set of steps.

Now, when facing the new problem of changing the wheels of a car, the first step of the process requires you to find a similar problem in your knowledge base. Once you've identified changing the wheels of a motorbike as a similar problem, the next step tells you to adapt the solution to the other problem to the current situation. For example, you might need to use a different set of tools to unscrew the wheels of a car than those used for a motorbike. Once you have obtained the correct tools you and try to apply same solution to the current problem you realise that a car is heavier than a motorbike, and, therefore, you cannot just lay it on one side to remove one of the wheels. This is where the third step come into play: you can tweak the known solution by introducing a new step and a new tool, the jack to lift the car. At this point, you can go ahead and solve your problem. If you managed to successfully change the wheels of your car this way[2], you can add the newly acquired knowledge to the knowledge base.

2.4.1 Application to Algorithmic Composition

There are two ways to music composition using `CBR`: one that purely relies on the technique itself, the other that consists in combining it with some other generative algorithm. Most of the literature deals with the first approach.

`MuzaCazUza` [20], is an example of a pure `CBR`-based melody generation system. In order to generate a new melody, the system takes as input a human-generated string that represents the structure of the melody and the properties each of the musical phrases should have. For each section in the provided melody structure, the system uses a case database, built from melodies composed by human musicians, to find musical phrases that have features as similar as possible to those provided in the user input.

Every time a musical phrase needs to be found, the whole case base is traversed and a similarity score (based on features like pitch and duration) is computed between each entry in the database and the corresponding phrase description provided by the user. The entry with the highest score is then selected.

A set of transformations is then applied to the selected entry, so that it will be even more similar to the phrase description provided by the user. Two such transformations are: transposition, where the pitches of the phrase are brought to a different musical key, and repetition, where portion of the phrases are repeated.

Finally, the authors evaluated their work by doing a qualitative analysis of the impact that changing weights of features in the similarity measure has on the generated melody.

An example of a hybrid system is given by Parikh [21], who based on research by Ramalho, et al [22] developed `IRIS`, a `CBR`-based jazz improvising system. Parikh's system aims at automatically generating an improvised solo for a melody. It will therefore play the beginning

[2]If you didn't, you might want to ask yourself a couple of questions..

of a pre-composed melody, then improvise a solo and finally play the last part of the pre-composed melody.

His core intuition was that approaches that try to improvise music a note at a time are counter-intuitive and go against the way musicians actually create music. He, therefore, argues in favour of a fragment-based model: "A fragment-based approach alleviates the need to decide what is musically valid and preserves many of the qualities that made the piece musically interesting." This approach takes as a starting point a database of musical fragments extracted from the works of well known musicians; those fragments are then composed and modified by the system during its live performance.

Fragments are extracted from `MIDI` files and stored in a database. Whether this extraction is performed manually or automatically is not clear, however, judging by the type of meta-data associated with each fragment, and song, such as the number of choruses it contains, suggests that this process involved some human analyst.

Some features, called "properties" by Parikh, such as the average note interval, are then extracted for each fragment. These properties are then used to generate inference maps (recall Markov chains discussed in Section 2.3) between fragments in each analysed song. Inference maps can be thought of as probability matrices that describe the probability of observing a certain feature value given the current feature value; thus, choosing the next fragment based on them, has a random, stochastic component. This random component of the selection process is presented by Parikh as what makes his system more creative than the one by Ramalho, et al [22]: " their program does not use any randomness and the final result is completely determined by the CBR process. Creativity by its very nature is tied to randomness."

Once the system has started improvising the solo, it uses the inference maps it computed to determine what features the next fragment should have: fragments are selected one at a time and, like in a Markov chain, what fragment will be chosen next only depends on the current one and not those before it. Since inference maps only provide information about the features the next fragment should have, a case database is used to find the fragment that has features most similar to the required ones.

The third step of `CBR` is the revision of a known solution. In `IRIS`, once a fragment is selected, the system revises by applying a set of transformations that make it more suitable to the current musical context. One such transformation is transposition: if a fragment in `A Major` is required, but the match in the database was a fragment in `C Minor`, the fragment can be transposed accordingly. However, as the author himself admits, such transformations are dangerous as they might result in something that does not sound good at all.

Finally, the system does not implement the last step of `CBR`: transformed fragments, generated during the improvisation, are not retained in the database, since the system does not support interactive human feedback on the improvisation and no other good way of determining whether a fragment sounds good or not has been found.

2.4.2 Strengths

There are two main advantages to `CBR`: it is easy, computationally speaking, to grow the knowledge base by adding new elements to it, and the technique preserves the original knowledge in some sort of symbolic representation that is easier to understand than a set of weights in a `ANN`. On top of that, the fact that existing knowledge is preserved as-is is a big advantage in fields where it is not exactly clear what parts of a solution make it a good solution. This is the case in algorithmic composition: as of today, nobody has come up with a working formal model that defines what makes music sound good.

2.4.3 Weaknesses

There are three main challenges in applying `CBR` to a problem: it can be difficult to come up with a good similarity measure[3], it cannot be straightforward to revise an existing solution to the new problem, and it is not easy to decide whether the revised solution is a good solution that should be added to the knowledge base.

Take algorithmic composition, since it is not clear yet how to measure the "goodness" of a musical phrase, most authors limit themselves to simple transformations like transposition hoping not alter the quality of a phrase composed by a human musician. This means that the revision of phrase is extremely limited lest producing a new phrase that does not sound good.

The same applies to the retention phase: if you cannot easily evaluate the "goodness" of a musical phrase, you cannot decide whether to add a new one to the knowledge base or not. This is why some systems are interactive, allowing a user to provide a input on the quality of a phrase before deciding to discard it or retain it.

[3]This depends, partly, on the difficulty of extracting features that are representative enough to describe a complex case

Chapter 3

Architecture

3.1 Overview

While the systems described in Section 2 focus on some specific musical genre, or musical instrument or rely on human input during a live performance, `Improv` does not share these limitations. In fact, it is an autonomous distributed improvising system of independent agents, each playing a different instruments. The system is designed to be genre-agnostic[1] and it contains no optimisations specific to any musical genres.

An overview of the system's structure follows.

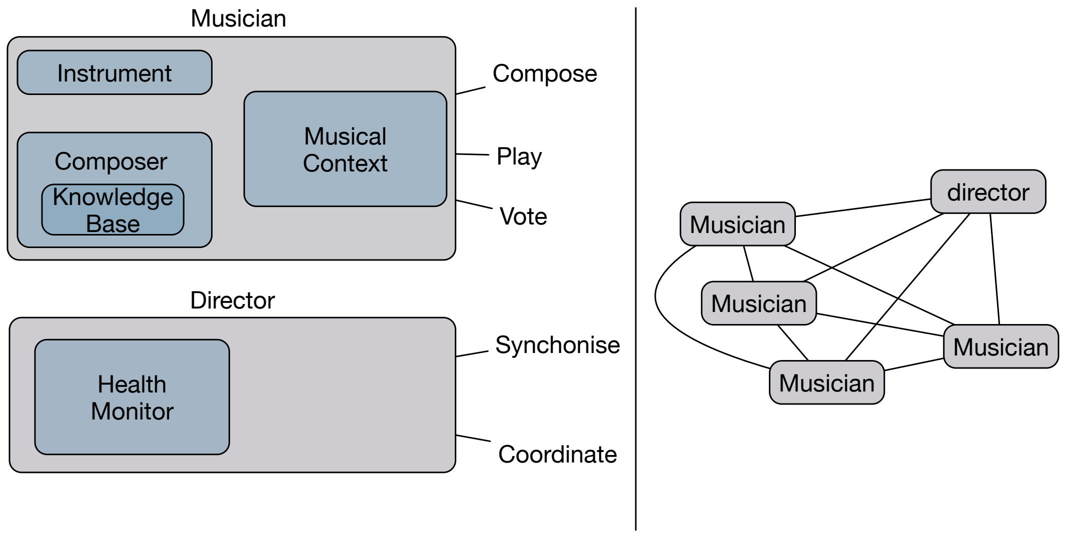


Figure 3.1: Overview of Components of the System

Time is divided in windows. In each time window, agents play music and exchange messages that contain information about what they are playing. This information is used by each agent to decide what it is going to play in the next time window.

[1]The system is designed to support the improvisation in multiple genres, however we did not have enough time to tag and classify enough training data for different genres. More details in Section 4.2 and Section 10.1

The core assumption on which the project rests is that if one starts from a set of musical phrases that are assumed to sound good and somehow combines them together the result should sound reasonably good too. While it is hard to assess whether music sounds good or not, a few experiments have been run to try to determine whether it sound repetitive or not (more details are given in Section 8.3

Because of this assumption, we can then take a set of melodies composed by humans, segment them and let the distributed system recombine them together in real-time to try to obtain an improvisation that sounds reasonably good.

As we can see in Figure 3.2 we can divide the project into two parts: training and live performance.

In the first part, a set of `MIDI` files is processed in order to create a knowledge base that the system can then use to generate its improvisations. For more details on this process please refer to Section 5.

During the live performance, agents in the system collect information about what others are playing and use the musical knowledge acquired during their training to generate a musical phrase that they are going to play in the next time window.

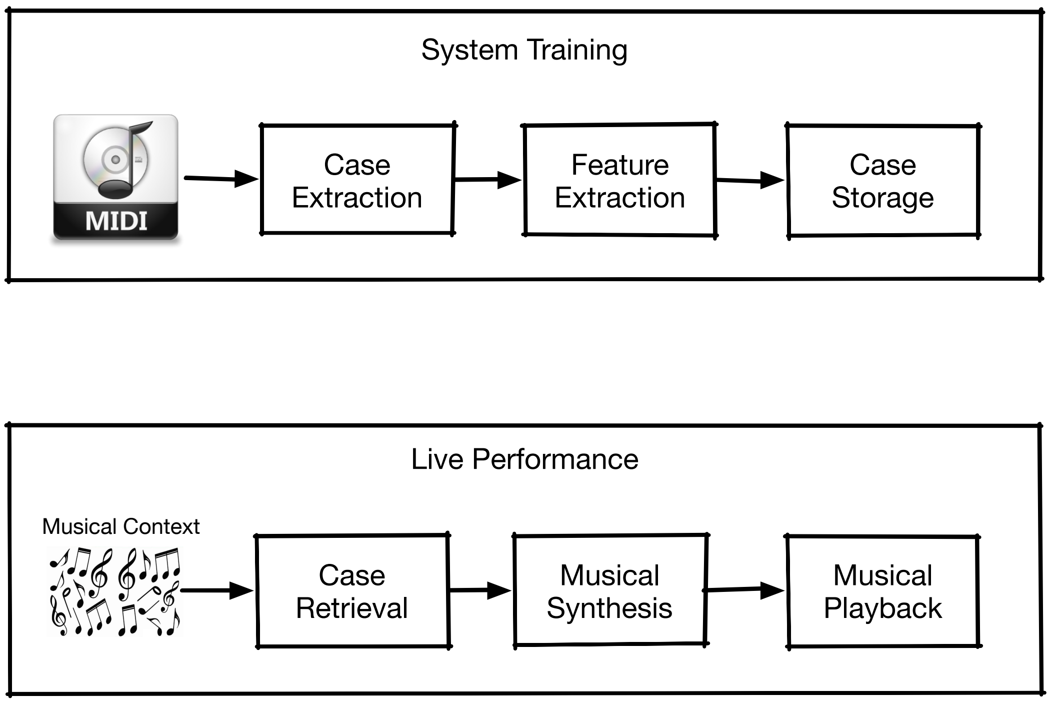


Figure 3.2: System Workflow

We have chosen `Scala` as the programming language for the development of this project due to the fact that it has nice built-in support for concurrency, it runs on the `JVM`, it can use `Java` libraries and it is very expressive (no boilerplate code needs to be written).

3.2 Actors

The agents in the system have been implemented using AKKA [23] actors. AKKA is a framework that makes it easy to build distributed systems in `Scala`. Each actor can send and receive messages from other actors. The delivery of messages is not guaranteed, but the message ordering is.

In Improv there are two main types of actors: musicians and directors. The following sections will describe them in more depth.

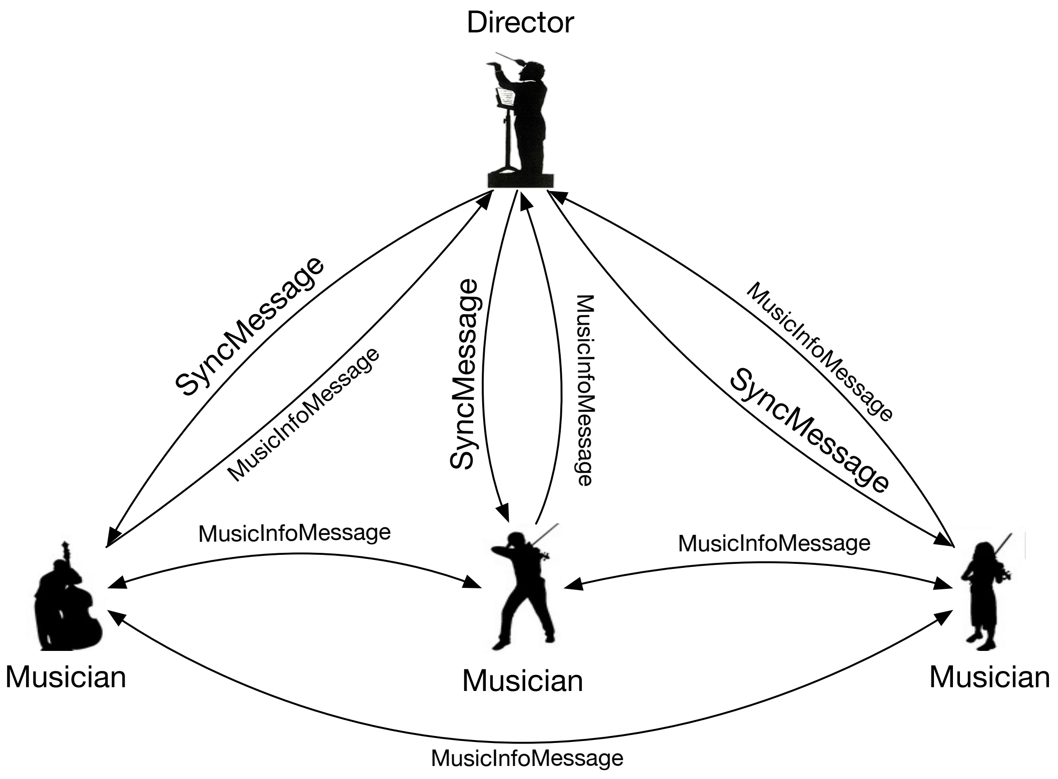


Figure 3.3: Message Flow

3.2.1 Musicians

Musicians are actors that are tasked with generating and playing music. They have two main components: a composer and an instrument.

As mentioned in the introduction, we want agents in our system to mimic the process of musicians listening and reacting to each other's musical performances during a jamming session. The process has been modeled as follows: every time a musician starts to play some music, it broadcast a `MusicInfoMessage` containing symbolic information what it is playing to all other actors in the system. Musicians collect those messages in a cache, effectively listening to one another and building a musical context that will allow them to react to what they have "heard" by generating a suitable musical response.

Each musician has a component, the composer, that is responsible of generating a musical phrase given a certain musical context. More details are given in Section 3.3.

Once a response has been generated by the composer, a musician uses its instrument to actually play it. More details on how musical phrases are played are given in Section 7

Musicians' Behaviours

Each musician in the system has a set of behaviours associated with it. These behaviours determine how it will react to incoming messages from other actors in the system.

An example of such behaviours is the `BoredomBehaviour`: an actor grows bored as the improvisation progresses, making it more inclined to want to stop the improvisation. For more details refer to Section 4.

3.2.2 Directors

Directors are actors whose role is to synchronise musicians. If musicians go out of sync, the improvisation will most likely not sound good.

Alternative methods of synchronisation have been considered, such as the use of distributed consensus algorithms to periodically synchronise the clock of the musicians. However, using a director seemed a simpler and effective approach.

In order to keep the musicians in sync, the director sends a `SyncMessage(t: timeTick)` to all musicians in the system. Two implementations for a director have been provided in this project: a `SimpleDirector`, and a `WaitingDirector`.

The `SimpleDirector` sends a `SyncMessage` at regular intervals, while the `WaitingDirector` waits for all players to be done playing before broadcasting the next `SyncMessage`.

Whenever the `WaitingDirector` receives a `MusicInfoMessage` it adds its sender to a set of musicians who are considered to be still playing. The `MusicInfoMessage` will also contain information about whether the musician is aware of the director's identity; if that is not the case, the director will send a `DirectorIdentityInfoMessage` to the sender.

When a musician is finished playing, it will send a `FinishedPlaying` message to the director. Once the director receives such a message it will remove the sender from the set of musicians still considered to be playing. Once that set is empty, the director will broadcast the next `SyncMessage`.

It might be the case, though, that messages from some of the musicians are lost, or that some musicians have crashed. To avoid blocking the whole system in such circumstances, the `WaitingDirector` is equipped with a system health monitor which notifies it whenever a musician is suspected to have crashed. When the director receives such a notification, it removes the suspect from the set of musicians considered to be playing. More details about the health monitor are provided in Section 3.4.

3.3 Composers

Composers are a crucial part of the system as they are responsible for generating new musical phrases in response to what other musicians have played in the previous time window. A composer in itself is not an actor, but is a component on which an actor relies for the generation of musical phrases. Composers take instruments and a set of musical phrases as their input and produce a musical phrase as their output. Refer to Section 5.2 for more detail about musical notation.

There are three types of composers: the `RandomComposer`, the `MIDIReaderComposer` and the `CBRComposer`.

3.3.1 RandomComposer

This composer simply generates a random musical phrase of a fixed duration regardless of its input. A random phrase is made up of a random combination of rests and notes of random pitches and durations, which are generated until the desired phrase length is reached.

3.3.2 MIDIReaderComposer

This composer generates the next musical phrase by reading it from a `MIDI` file. It makes use of a music parser to obtain an iterator over a list of musical phrases contained in the file and just returns the next each time its `compose` method is called, until the end of the file.

3.3.3 CBRComposer - a.k.a.: CaseBasedReasoningComposer

This composer, instead, makes use of a knowledge base and a population selector to generate its musical response. The knowledge base stores pairs of musical phrases[2]: each pair represents a phrase and the phrase that was played after it in some song, composed by a human musician, in the training set. More details about the knowledge base database can be found in the Section 5.4.

Once a population of candidate musical responses has been retrieved, a population selector is used to generate a single musical response. For more details on the music generation process please refer to Section 6.

3.4 Monitors

Monitors are components of the system that can either be created as stand-alone actors or used by actors as add-on modules. In the first case, monitors can subscribe to receive any messages broadcast on the network; in the second case, the actor which makes use of the monitor is responsible for notifying monitor of any relevant messages or events on the network.

There are two main types of monitors: the `StatsMonitor`, and the `HealthMonitor`.

The role of the `StatsMonitor` is to collect statistics about messages and actors in the system. For example it may collect information about the number of active actors in the system in a given time-frame.

The role of the `HealthMonitor` is to provide information about whether an actor is considered to have crashed or not. The `HealthMonitor` trait[3] also extends the `Observable` trait, and notifies all its observers as soon as some actor is suspected to have crashed.

The `SimpleHealthMonitor` is an implementation of such trait: every time it receives a message from an actor in the system, it schedules, after a specified timeout[4], a task to notify its observers that the actor is suspected of having crashed. The monitor also keeps a

[2]And the instruments with which they have been played

[3]A trait is the `Scala` equivalent of a `Java` interface. However, it's more similar to a `Java 8` interface, than to the earlier versions, since a trait can have default implementations and abstract fields

[4]The timeout can be specified as a constructor argument

map between actor IDs and the `Cancellable` notification tasks, so that upon the receipt of a new message from an actor, the old task associated to it can be canceled and a new one can be created, thus resetting the timeout.

If the observable monitors are instantiated as actors, notifications will be in the form of messages to its observers, otherwise they will use some provided callback function.

3.5 Messages

The following sections will describe the different types of messages used for communication between the agents in the system. All messages include a sender: this is useful for example when musicians need to reply to the director.

3.5.1 SyncMessage

This message is sent by the director to all the musicians in the system and serves the purpose of making them play all the same time[5]

3.5.2 MusicInfoMessage

This message is broadcast to the network by each musician as soon as it starts playing a musical phrase. The message contains the following fields:

◇ **PHRASE** ⇒

> The musical phrase the musician has just played. This information will be used by the other musicians to generate a new musical phrase to be played in the next time window.

◇ **TIME** ⇒

> The time window in which the musical phrase has been played

◇ **DIRECTOR** ⇒

> The identity of the director, if the musician is aware of it, $None$ otherwise

◇ **INSTRUMENTTYPE** ⇒

> The type of the instrument of the instrument used by the musician to play the musical phrase (`PIANO` for example).

3.5.3 DirectorIdentityMessage

When a director receives a `MusicInfoMessage` from a musician, it checks whether the `director` field is *None* or not. In case it is, the director knows that the musician is not aware of the director's identity, so it sends it this message which contains the director's unique identifier.

[5]Neglecting the small differences in the time the messages are received cause by the latency on the network

Chapter 4

Distributed Consensus

Sometimes the musicians in the system will need to take a collective decision about some aspect of the improvisation, be it the genre of music to improvise or when to stop playing.

There are three formal requirements for distributed consensus:

1. **Agreement** ⇒
   No two processes decide differently
2. **Validity** ⇒
   If any process decides on a value `V`, then `V` must have been proposed by some process in the network
3. **Termination** ⇒
   Every non-faulty process eventually decides

A very common approach to achieve distributed consensus is that of having a coordinator which handles a voting process: when a decision needs to be made, agents in the network communicate their vote to the coordinator, which, in turn will broadcast the decision, once the voting has terminated.

Our system already has an actor which is responsible for the synchronisation of musicians: the director. It's easy to see how this actor would be a good candidate to act as a coordinator. For the sake of keeping things simple, we will assume that the director cannot crash.

In the following sections we analyse how the two main decisions mentioned above, termination and genre selection, are taken by the system.

4.1 Termination

This problem resembles that of `Atomic Commitment` in distributed databases: a unanimous decision on whether a distributed transaction should be committed or aborted must be reached. The formal requirements of such problem can be formalised as follows:

1. All participants that reach a decision reach the same decision (`commit` or `abort`)
2. If a participant decides `commit`, then all participants must have voted `yes`
3. If all participants voted `yes`, and failure occurred, then all participants decide `commit`
4. Each participant decides at most once

The 2-phase commit (2PC) algorithm is often used to solve such problem. The algorithm can be described as follows:

1. The coordinator proposes a vote on a binary decision (e.g.: "commit") [Initialisation]
2. Each participant votes either yes or no [Phase 1]
3. The coordinator broadcasts commit if every participant has voted yes, otherwise abort.

The disadvantage of 2PC is that it has a single point of failure: if the coordinator crashes, the algorithm might not terminate: while it is safe for a participant to decide abort without waiting for a decision from the coordinator if it has voted no[1], if a participant vote yes it has to wait for a decision from the coordinator before deciding as some other participant might have voted no[2]. However, this limitation does not apply in our case because we assumed that our coordinator never fails.

It looks like 2PC could be a suitable algorithm to reach consensus on termination. However, as we mentioned above, 2PC needs to be initiated by the coordinator. So we are left with one final question to answer: how does the director, acting as the coordinator, know when to initiate the voting process?

As mentioned in Section 3.2.1, each musician has a BoredomBehaviour which determines how a musician will act and react to incoming messages based on its level of boredom. Boredom has been implemented as an exponential function of the time passed since the musician has started improvising. There are two thresholds defined by the BoredomBehaviour: the first determines whether the musician is willing to terminate the improvisation if some other musician proposes to do so, the second (higher than the first) determines whether the musician itself is willing to propose termination. Once the boredom level has gone above the second threshold, the musician will send a message to the director asking it to begin a voting on termination. Once the voting process has started, musicians whose boredom level is above the first threshold will vote yes, while the others will vote no. This process is illustrated in Figure 4.1.

[1]If a participant votes no, by the requirements of Atomic Commitment, it must be the case that all other participants will abort as soon as they learn the decision from the coordinator, thus it is safe to abort without waiting for a decision from the coordinator

[2]Safe termination can be achieved with 2PC if a stronger broadcast protocol, like Uniform Timed Reliable Broadcast, is used.

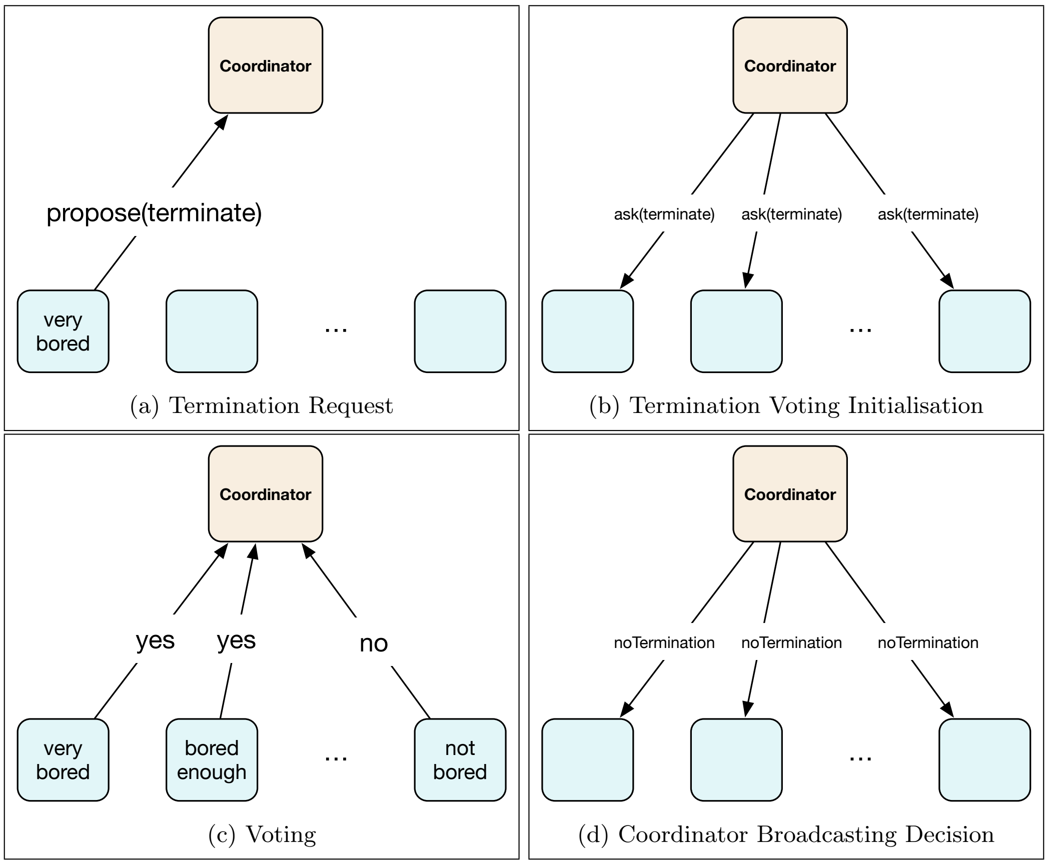


Figure 4.1: Reaching Consensus on Termination

4.2 Genre Selection

Unlike the termination problem, choosing a musical genre for the improvisation session does not involve a binary decision: each musician has to choose one of many possible musical genres.

However, the algorithm described in the previous section can be easily generalised to reach consensus for problems that do not involved binary decisions: instead of just voting `yes` or `no`, each musician, during the voting phase, can propose a their favourite value. In the case of musical genre, each musician can pick one and vote for it.

Clearly, if the decision is not a binary decision, a unanimous voting process does not work very well as the chance of all musicians voting for the same genre decreases as the number of musicians in the system gets larger. Because of this, we can instead use a majority voting system where the genre chosen for the improvisation session will be the one that has received the majority of votes. In theory this has one problem: what happens if one of the musicians in the system does not know how to improvise a specific genre. Since in this project all musicians share the same knowledge base, and each musician will only vote for a genre it can improvise on, we will not face this problem. However, if one were to experiment with musicians having different knowledge bases, this problem could be solved by adding to their vote information about what genres they are able to improvise. Then the genre could be chosen as follows:

- Each musician votes `(preferredGenre, supportedGenres)`

◇ The director collects all votes and ranks genres by the number of votes received

◇ The director takes the intersection of all the `supportedGenres` sets[3]

◇ The director chooses and broadcasts the genre that has highest rank within the computed set intersection

One final difference with the termination problem is that a musician needs to know about what musical genre to improvise on before it can start playing, therefore a musician joining the improvisation session after it has already started needs to learn what genre the others musicians are improvising. This could be done by including the genre in the $MusicInfoMessage$ described in Section 3.3. However, since a musician needs only to learn about the genre once, doing so would unnecessarily increase the message size (although, arguably, by a negligible amount).

The alternative adopted in this project is that, as soon as a musician receives a `SyncMessage` from the director, if it does not know what genre to improvise, it will ask for a voting process to choose the genre. As we can see in Figure 4.2, if the director is aware of a genre already chosen by other musicians, it will simply reply to the musician with information about the genre, otherwise it will start a system-wide voting process to determine it.

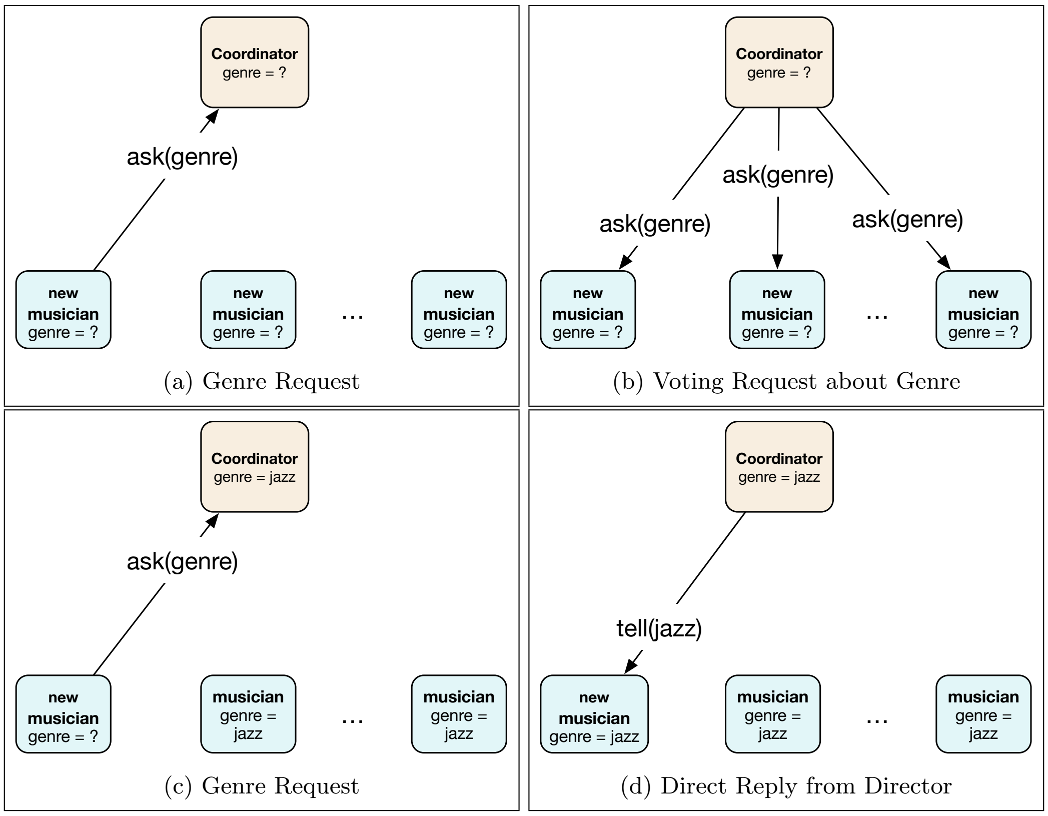


Figure 4.2: Improvisation Genre Inquiry

[3]If the intersection of these sets is empty, clearly it is not possible to start an improvisation session with all the musician participating to it. An option at this point would be to terminate. Another option would be to find a non-empty set intersection of `supportedGenres` by trying to exclude a musician at a time from the improvisation

Chapter 5

Training

In this section we will describe the different steps of the training process. Each actor relies on a knowledge base in order to generate music. Training is the process of creating such knowledge base from existing data.

More specifically, since the machine learning technique adopted in this system is Case-based Reasoning (CBR) in combination with Genetic Algorithms (GA), the training process will produce a case database which can be used to generate an initial population for the GA.

The existing data will come from a set of `MIDI` files collected from the internet.

As we can see in Figure 5.1, each `MIDI` file is parsed, and converted into our musical representation (for details on our musical representation, please refer to Section 5.2. Then a set of musical cases is extracted. A musical case is made up of a musical phrase and instrument type. Features will then be extracted for each case, thus creating a description for it. The case description will then be used to store the case in the database.

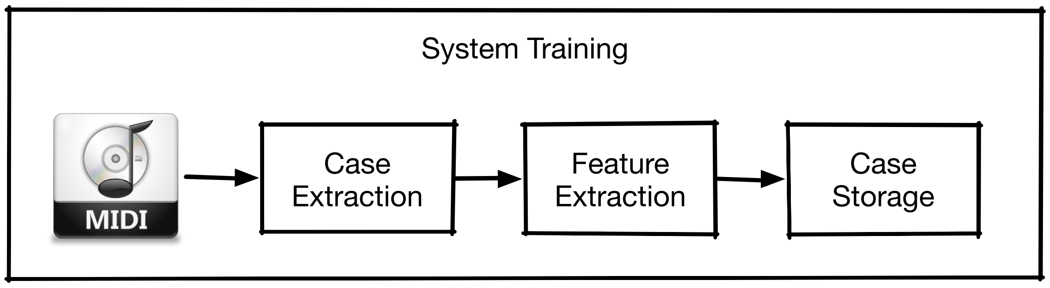


Figure 5.1: Training Workflow

In Figure 5.2, instead, we can observe the architecture of the portion of the system involved with training. The `SystemTrainer` is composed of a `CaseExtractor` and a `CaseIndex`. The `CaseExtractor` is used to extract a list of tuples of musical phrases from a `MIDI` file, the cases are then stored in the `CaseIndex` which represents the core of the knowledge base.

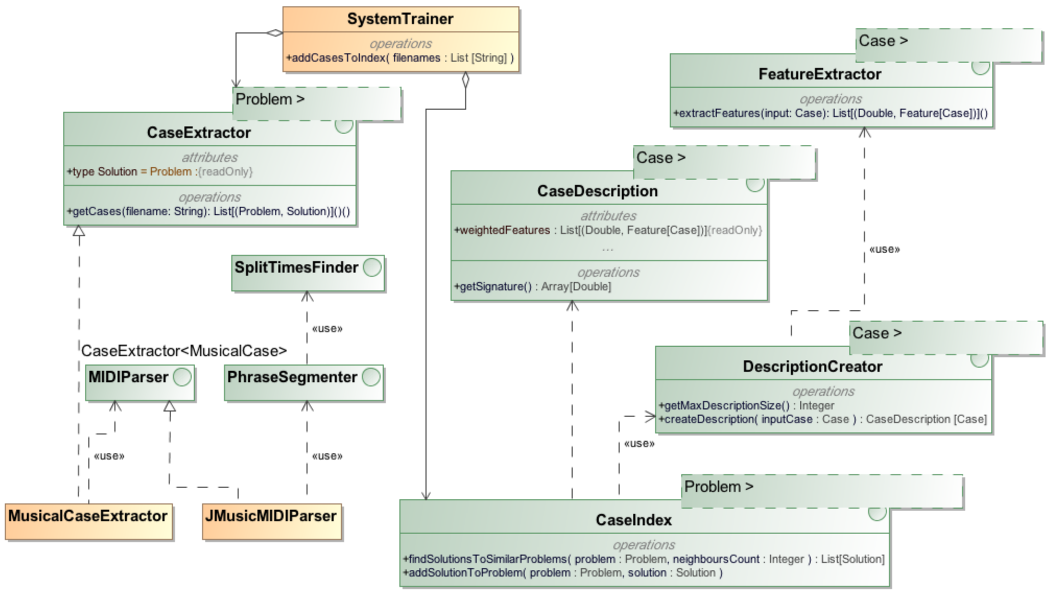

SystemTrainer
operations
+addCasesToIndex(filenames : List [String])
Problem >
CaseExtractor
attributes
+type Solution = Problem :{readOnly}
operations
+getCases(filename: String): List[(Problem, Solution)]()()
SplitTimesFinder
«use»
CaseExtractor<MusicalCase>
MIDIParser
PhraseSegmenter
«use»
«use»
MusicalCaseExtractor
JMusicMIDIParser
Case >
FeatureExtractor
operations
+extractFeatures(input: Case): List[(Double, Feature[Case])]()
Case >
CaseDescription
attributes
+weightedFeatures : List[(Double, Feature[Case])]{readOnly}
...
operations
+getSignature() : Array[Double]
«use»
Case >
DescriptionCreator
operations
+getMaxDescriptionSize() : Integer
+createDescription(inputCase : Case) : CaseDescription [Case]
«use»
Problem >
CaseIndex
operations
+findSolutionsToSimilarProblems(problem : Problem, neighboursCount : Integer) : List[Solution]
+addSolutionToProblem(problem : Problem, solution : Solution)

Figure 5.2: Training Sub-system Architecture

5.1 MIDI Parsing

Since this project focuses on the generation of music rather than audio signal processing, we decided to adopt a symbolic music format for our training data.

Both `MIDI` and `MusicXML` were considered as candidate formats, the latter being a more modern format that has better support for musical notation. In the end, however, `MIDI` was chosen because of the quantity of melodies freely available in that format. On top of that, since we have chosen `Scala` as the programming language for this project, we need libraries that can run on the `JVM`; it turned out that there are not any good libraries that supported `MusicXML` parsing and can run on the `JVM`.

5.1.1 Introduction to the MIDI Format

`MIDI`, short for `Musical Instrument Digital Interface`, is a standard that defines a communication protocol, a digital interface and a set of connectors that allow digital musical instruments and computers to communicate with each other.

As a file format, `MIDI` can be thought of as an event stream. As we can see in Figure 5.3, each file is made up of chunks: a header chunk that contains data about the whole file and a set of track chunks. Each track chunk contains a sequence of track events, which can be of two types: track events convey information such as when and on what channel a note should be activated and when it should be deactivated (`NoteOn`, `NoteOff`). The other type is meta events, which carry information such as the time signature and tempo of a track.

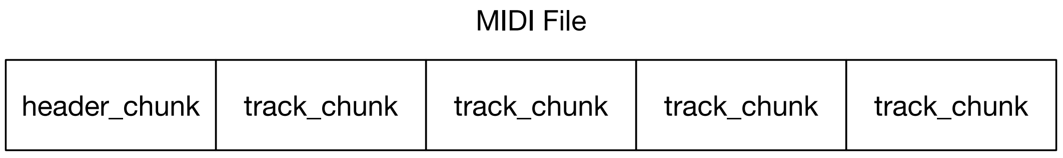


Figure 5.3: MIDI File Format

Events in different tracks are evaluated in parallel: this means that if two `NoteOn` events on different tracks have the same starting time, the two notes will be played at the same time.

Finally, `MIDI` supports multiple channels (up to 16), each of which can be associated with a `MIDI` instrument. Figure 5.4 shows an intuitive way of logically interpreting the `MIDI` file format.

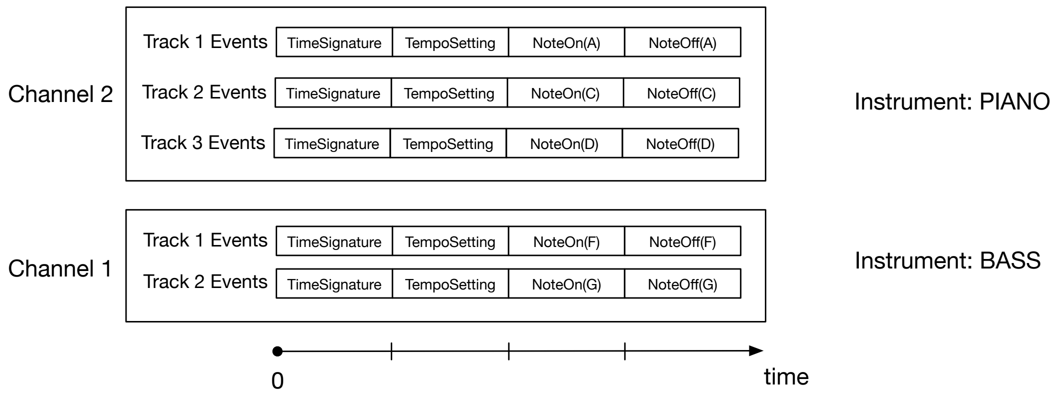


Figure 5.4: MIDI Channels, Tracks and Events

5.1.2 JMusic

`JMusic` is a `Java` library that offers a wide range of musical functionality that includes `MIDI` parsing. As we can see in Figure 5.5, the library turns a `MIDI` sequence into a `Score`, which contains a list of `Parts`. Each `Part` pretty much corresponds to a `MIDI` channel, is associated with a musical instrument and contains a list of `Phrases`. A `Phrase` corresponds to a `MIDI` track and contains a list `Notes`.

This library was selected as our `MIDI` parsing library. While the structure of the library was generally good, some of its methods required some changes to improve their performance. Section 5.5.1 gives an example of such optimisations.

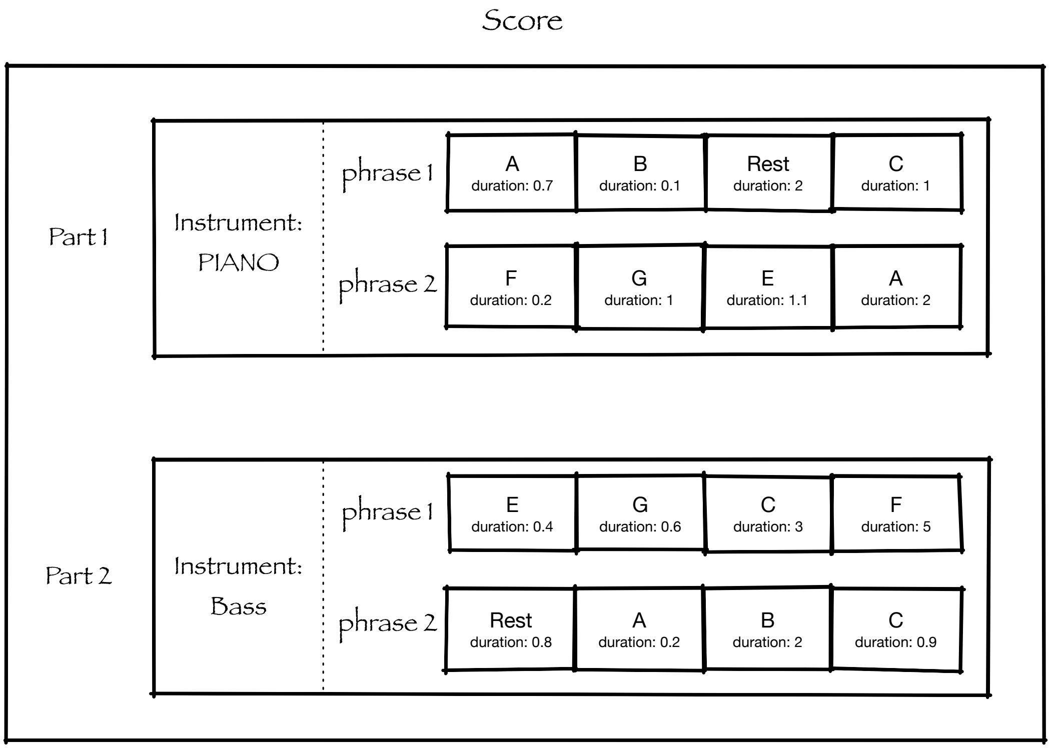
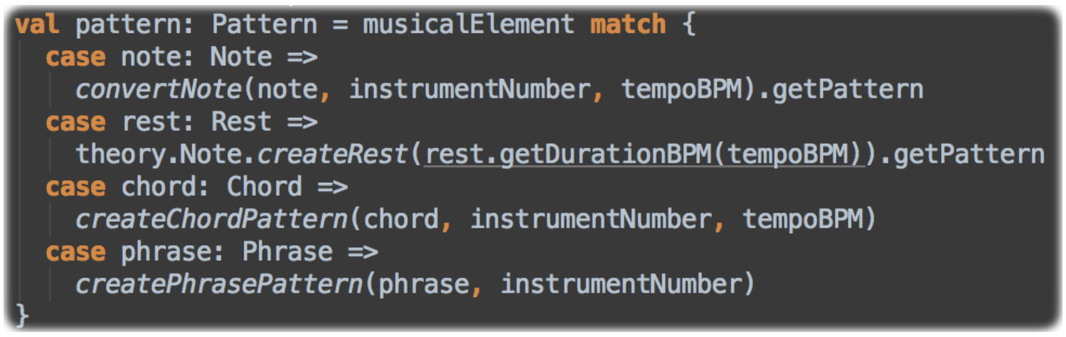


Figure 5.5: JMusic's Representation of a parsed MIDI File

5.2 Musical Representation

There are three main factors that drove the choice of adopting our own musical representation:

1. Given the complexity and low-quality design of `JMusic`'s codebase, even introducing a small change proved a difficult and long task. By adopting our own format we didn't need to change the library's source every time a new bit of functionality was required.
2. By building our own musical representation directly in `Scala` we could benefit from `Scala`'s `case classes`, which work very well with pattern matching (see Figure 5.6 for an example)
3. If we design the core of our system so that it makes use of our own representation, we are then granted a higher degree of freedom in the choice of libraries for `MIDI` parsing or sound playback. Once a better library comes out, we can simply adopt it and just change the code that converts their representation to ours, without having to touch the core of our system.



```
val pattern: Pattern = musicalElement match {
  case note: Note =>
    convertNote(note, instrumentNumber, tempoBPM).getPattern
  case rest: Rest =>
    theory.Note.createRest(rest.getDurationBPM(tempoBPM)).getPattern
  case chord: Chord =>
    createChordPattern(chord, instrumentNumber, tempoBPM)
  case phrase: Phrase =>
    createPhrasePattern(phrase, instrumentNumber)
}
```

Figure 5.6: Pattern Matching on MusicalElement

5.2.1 MusicalElement

This is the basic type in our musical representation. This trait defines a set method among which we can highlight the following:

- `getDuration(timeUnit: TimeUnit = NANOSECONDS): BigInt` ⇒
  Returns the duration of the musical element in the specified unit of time
- `getStartTime(timeUnit: TimeUnit = NANOSECONDS): BigInt` ⇒
  Returns the start time of the musical element in the specified unit of time
- `getEndTime(timeUnit: TimeUnit = NANOSECONDS): BigInt` ⇒
  Returns the starTime of the musical element in the specified unit of time
- `withDuration(timeUnit: TimeUnit = NANOSECONDS): MusicalElement` ⇒
  Returns a copy of the musical element but with its duration changed.
- `withStartTime(timeUnit: TimeUnit = NANOSECONDS): MusicalElement` ⇒
  Returns a copy of the musical element but with its start time changed.
- `withEndTime(timeUnit: TimeUnit = NANOSECONDS): MusicalElement` ⇒
  Returns a copy of the musical element but with its end time changed.

`Scala` introduces the concept of a `companion object`. A companion is an object with the same name as some class or trait and has access to all their methods and fields (including the private ones). An `object` in `Scala` can be compared to a `Singleton` class with only static methods in `Java`. Companion objects are frequently used to define functions that transform or somehow operate on the class/trait with which they are associated.

The `MusicalElement`'s companion object defines a few methods that help manipulate musical elements. Notably among them we can mention the method:

```
def split(elem: MusicalElement, splitTimeNS: BigInt):
    (Option[MusicalElement], Option[MusicalElement])
```

This will take a musical element and a time at which it should be split as input and produce a tuple of optional musical elements. As we can see in Figure 5.7 the method works as follows:

- If the split time is before the start time of the musical element, return `(None, Some(elem))`
- If the split time is after the start time of the musical element, return `(Some(elem), None)`

◇ If the split time lies between the start and end times of the musical element, return a tuple of musical elements so that the left element has the same start time as the original element, but duration equal to $splitTime - elem.getStartTime$. The right element instead will have start time equal to $splitTime$ and duration equal to $elem.getEndTime - splitTime$

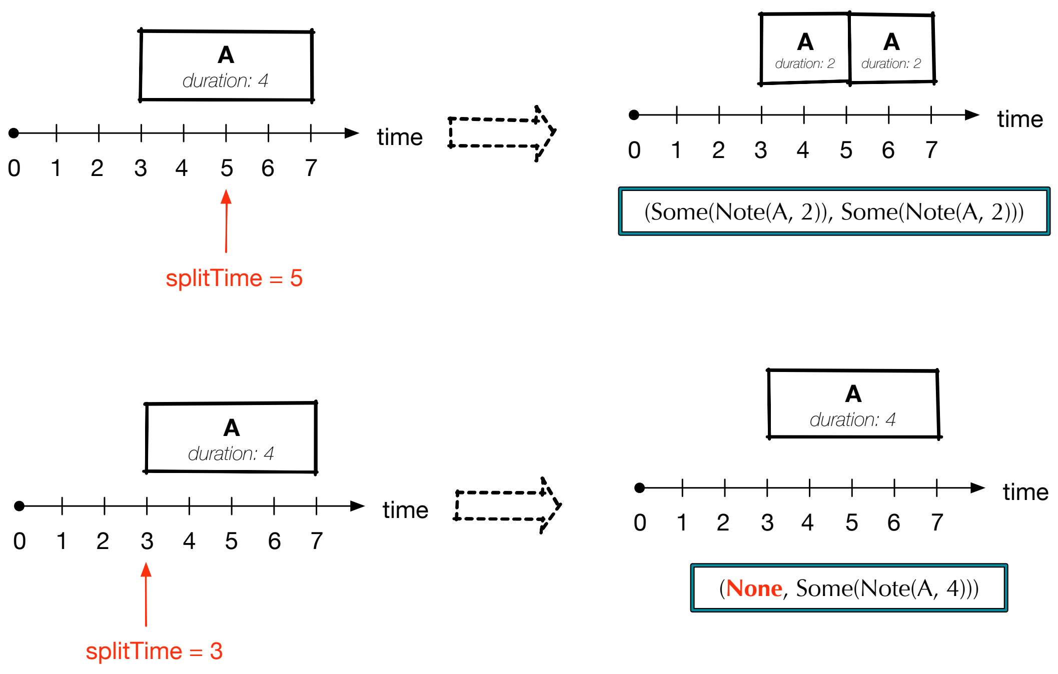


Figure 5.7: Splitting of a Note

5.2.2 Note

In addition to the methods inherited from `MusicalElement` a `Note` has the following fields:

◇ `octave` ⇒
The octave in which the note is situated, obtained by taking the pitch modulo 12 (if you consider alterations, sharp and flat, there are 12 notes in an octave).

◇ `midiPitch` ⇒
The `MIDI` pitch of the note, which ranges from 0 to 127

◇ `accidental` ⇒
The accidental of a note can be either `Flat` (♭), `Sharp` (♯) or `Natural` (♮). The `Flat` accidental lowers the note's pitch by a semitone, while `Sharp` raises it by a semitone. `Natural` instead, leaves it unchanged.

◇ `loudness` ⇒
The loudness of a note can range from `SILENT` to `FFF`, which stands for "fortissississimo" (Italian for "very very loud").

5.2.3 Chord

A `Chord` is a set of notes with the same start time and duration that are played at the same time.

Calling the `withDuration` or `withStartTime` methods on a `Chord` results in a new `Chord` with the appropriate transformations applied to all the notes that belong to it.

`JMusic`'s `MIDI` parsing capabilities do not include the recognition of chords, therefore some techniques to extract them from a list of musical phrases in `JMusic Part` had to be devised. Please refer to Section 6.3.1 for more detail.

5.2.4 Phrase

In addition to the methods inherited from `MusicalElement` a `Phrase` has the following fields:

- ◇ `musicalElements` ⇒
  A list of musical elements ordered by their start time.
- ◇ `polyphony` ⇒
  A flag that determines whether the phrase is a multi-voice phrase or not. A multi-voice phrase is the direct equivalent of a `JMusic`'s part: it is made up of phrases that are to be played concurrently. The flag cannot be set unless the phrase only contains single-voice phrases in its `musicalElements`.
- ◇ `tempoBPM` ⇒
  The tempo at which the phrase should be played.

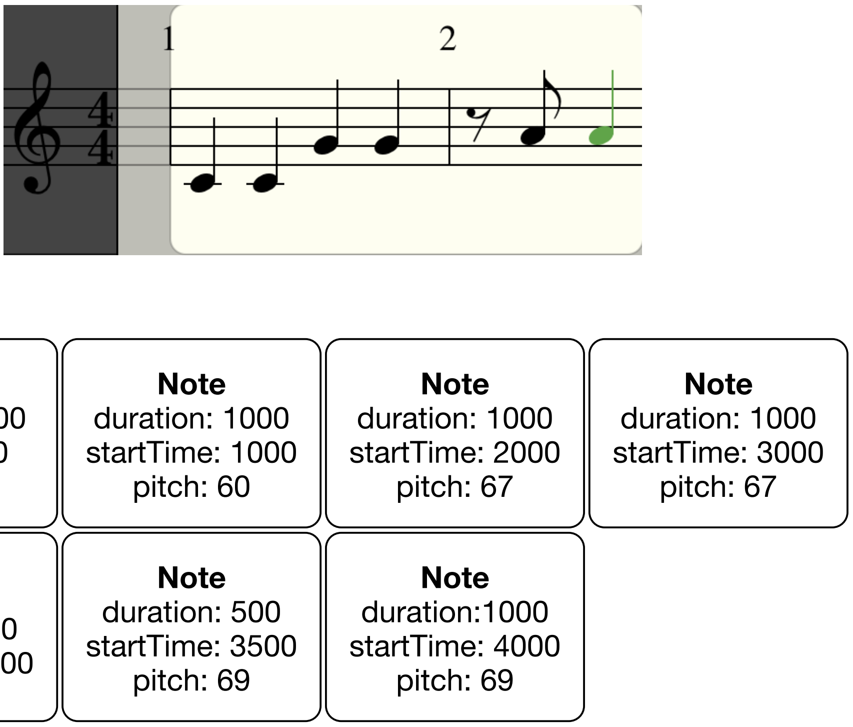


Figure 5.8: Musical Score Converted to our Musical Representation

5.3 Case Extraction

In order to populate our database and create our knowledge base, it is necessary to extract some information from the melodies in the training set.

As mentioned in Section 2.4, `Case-based Reasoning` relies on an index of cases, which represent problems and their relative solution. The problem each musician in the system needs to solve is what `Phrase` to play next based on its knowledge about what the other musicians have played.

Once phrased the problem in those terms, it is easy to see where `CBR` comes into play: given a set of existing melodies, we can process them, instrument by instrument, phrase by phrase and create a database of pairs of phrases. Each pair represents a phrase and its successor.

One of the assumption that have been made in this project is that the melodies included in the training set have been composed by competent human composers. It is thus safe to assume that if we split a phrase that belongs to one of such melodies into a list of sub-phrases, each adjacent pair can be considered a valid `(problem, solution)` pair to be added to the database.

5.3.1 Phrase Segmentation

In order to create a set of valid `(problem, solution)` pairs, we need to be able to divide a phrase into a set of sub-phrases.

As we can see in Figure 5.9, a `PhraseSegmenter` is a component that performs such task: it takes a `Phrase` as input and it produces a list of `Phrase`s as its output. To do so it relies on a component, the `SplitTimesFinder`, which provides it with a list of times at which the `Phrase` should be segmented.

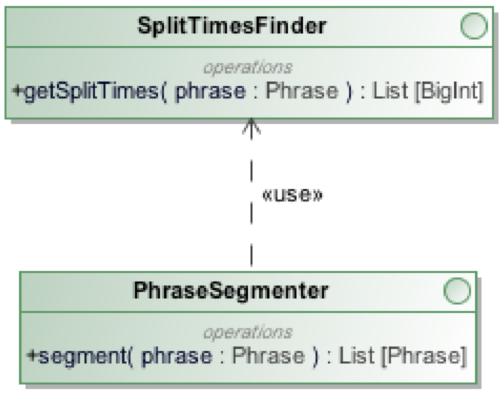

SplitTimesFinder
operations
+getSplitTimes(phrase : Phrase) : List [BigInt]
«use»
PhraseSegmenter
operations
+segment(phrase : Phrase) : List [Phrase]

Figure 5.9: Phrase Segmenter

In order to split a phrase, the `PhraseSplitter` will perform the following operations:

1. Set current phrase to `Option(phrase)`
2. Create an empty list of phrases
3. Get next split time if the current phrase is defined, else go to 8
4. Call `Phrase.split(phrase)` to get a pair `(newPhrase, rest)`
5. If `newPhrase` is defined, add it to the list of phrases
6. Set current phrase to `rest`
7. Repeat from 3
8. Return the list of phrases

Since the method `withDuration` on the `Phrase` class produces a scaled[1] version of the `Phrase` rather than cutting it at the specified time, the `split` method defined for a `MusicalElement` could not be used.

In order to split a phrase at a given time we perform the following steps:

1. Set the current time to the start time of the phrase
2. Create two empty lists
3. Get the next musical element in the phrase
4. Define $newTime = curTime + elem.getDuration$
5. If the current time is greater than the split time, add the element to the second list, otherwise
6. If the new time is less than or equal to the split time, we simply add the current element to the first list. If that is not the case, it means that our split time lies between the start and end of the current element. Because of this we will split the current musical element at the split time and add its first half to the first list and the second half to the second list.
7. We update the current time to be equal to the new time
8. Once we've iterated on all the elements we return a pair of `Phrase`s built from the two lists. All the elements in the second `Phrase` will be shifted so that the start time of the phrase is 0. This is necessary because otherwise we'd create a `Phrase` with a potentially long sound at the beginning: imagine we split a phrase at the 30-second mark, if we didn't shift the right phrase, we would have a phrase that starts at the 30-second mark. Such phrase, when played, would have a 30-second silence at the beginning.

When dealing with a multi-voice phrase, that is, a phrase with the `polyphony` flag set, we split all the phrases it contains, thus obtaining two set of phrases. We then combine those sets together two yield a pair of multi-voice phrases.

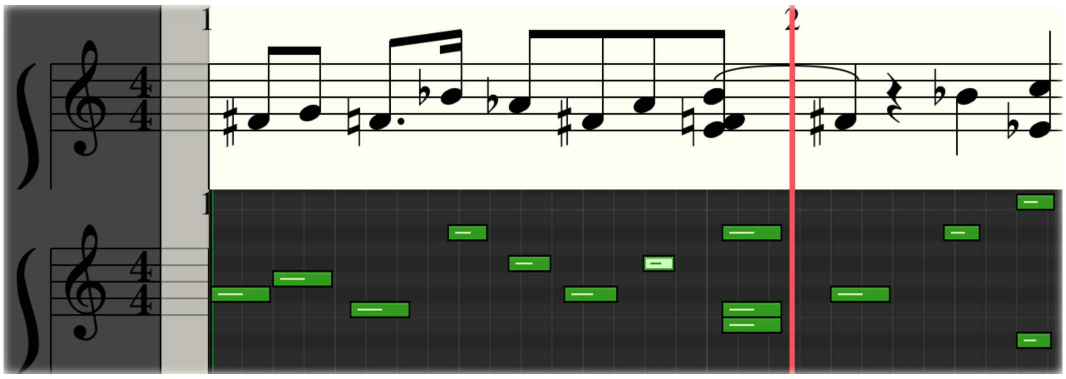


Figure 5.10: Splitting a Phrase at Every Bar

As we can see in Figure 5.10, a good way of segmenting a phrase, while at the same time avoiding splitting it into parts that would feel incomplete is to split it at every musical bar. Unfortunately, the `MIDI` format does not preserve the concept of musical bars: while it is a decent format for musical playback, all information about music notation are lost.

[1]A scaled phrase is a phrase that contains all the elements as the original phrase, but has a different duration. The ratios between the duration of the elements in the phrase will be preserved

To go around this limitation we have two options: we can split the phrase at intervals of fixed duration or we can make use of a technique called Local Boundary Detection Model. The latter technique suggests candidate local boundary points in a melody, which can then be converted into split times for our phrase segmentation algorithm.

Local Boundary Detection Model

The Local Boundary Detection Model [24] calculates the boundary strength between intervals in a melody by examining the strength of local discontinuities. Peaks in boundary strength are to be considered good candidates for a local boundary in the melody. This model relies on two rules: the `Change` rule and the `Proximity` rule.

The first rule states that the boundary strength is proportional to the change between two consecutive intervals.

The second rule states that if two consecutive intervals are different, the boundary introduced and larger interval is proportionally stronger.

The first rule can be implemented by a function that computes the degree of change between intervals, while the second can be implemented by multiplying the output of the function by the absolute value of the interval.

A description of the algorithm illustrated by Cambouropoulos [24] follows:

◇ Take a melody and convert it into a set of parametric interval profiles. An interval profile is a sequence of numbers that represent the intervals between consecutive musical elements with respect to a given parameter. The parameters considered by this algorithm are `pitch`, `ioi` (inter-onset intervals) and `rests` (the time intervals between the beginning of a note and the end of the previous). Here Cambouropoulos adopts a different definition of `ioi` than [25]: the latter defines it as the time interval between the beginnings of successive events or notes, not including the duration of the events. Clearly, if this was the definition adopted by Cambouropoulos, `ioi` would be effectively the same as `rests`. It is thus safe to assume that Cambouropoulos includes the duration of the previous note in the `ioi`.

◇ Compute the degree of change r between two consecutive interval values x_i and x_{i+1} as follows:

$$r_{i,i+1} = \frac{|x_i - x_{i+1}|}{x_i + x_{i+1}} \qquad \text{iff } x + x_{i+1} \neq 0 \wedge x_i, x_{i+1} \geq 0 \tag{5.1}$$

$$r_{i,i+1} = 0 \qquad \text{iff } x_i = x_{i+1} = 0 \tag{5.2}$$

Cambouropoulos suggests to add a small value to all the intervals in order to avoid irregularities introduced by intervals of size 0. It is also safe to assume, given the conditions on the provided equations, that all intervals should have positive values, thus requiring us to take the absolute value of pitch differences. For example `G` followed by `A` would yield a semitone interval value of -1, which would result in an unknown degree of change r.

◇ Compute the strength of boundary s_i for interval x_i as follows:

$$s_i = x_i * (r_{i-1,i} + r_{i,i+1}) \tag{5.3}$$

◇ Compute the a sequence of boundary strengths $S = [s_1, s_2, \cdots, s_n]$ for each parameter (`pitch`, `ioi` and `rest`) and normalise it in the range $[0, 1]$.

◇ Compute an overall sequence of boundary strengths by linearly combine the sequences for each parameter: $S_{overall} = w_p * S_{pitch} + w_i * S_{ioi} + w_r * S_{rests}$ (addition here is to be considered as vector element-wise addition). Cambouropoulos proposes a set of weights: $w_p = 0.25, w_i = 0.5, w_r = 0.25$

◇ Find local peaks in the sequence: they will indicate local boundaries in the melody.

Once local peaks have been found in the approach described above, we can use their index to work out what interval they correspond to, and, subsequently, what time in the sentence. For example, if we detect that the third interval constitutes a local boundary, we know that a good split point is between the third and fourth note in the phrase, which means that the combined duration of the first three notes gives us a split time.

Most often we have to deal with multi-voice phrases (phrases with multiple notes played at the same time). Cambouropoulos mentions that it is possible to compute the degree of change also for chords and not only for individual notes; however, he does not mention how one might do so. In order to overcome this problem two approaches have been adopted: perform the `LBDM` analysis on the longer of the sub-phrases in a multi-voice phrase, relying on the assumption that the phrase which carries the most information about the melodic line will have more elements, or the sub-phrases can be merged together yielding a single-voice phrase which can chords as well as notes. For more details on the merging of multi-voice phrases into one single-voice phrase refer to Section 6.3.1.

When dealing with chords instead of notes, intervals are calculated in a slightly different way: the pitch of a chord is calculated as the sum of the pitches of the notes that belong to it, while, since notes in a chord start and end at the same time, the start and end times are taken from any note belonging to the chord.

For a better comparison between the two approaches, please refer to Section 8.2.

5.3.2 Case Generation

The objective of our training process is the generation of a knowledge base that provides good musical response candidates to musical phrases a musician has "heard" other musicians playing.

For the sake of simplicity, let's assume that our training set only contains a song in which three instruments are playing: a piano, a cello and a bass. In Figure 5.11 we can see how the melody looks like after phrases for each instrument have been segmented. The arrows represent the relationships that we would like to create in our knowledge base between the phrases. For example, we can see how a pianist who hears a cello musician play `Phrase 21` might consider `Phrase 12` an appropriate response, as it a phrase played by the piano player in the figure immediately after a cello player played `Phrase 21`. Similarly, a piano player who hears a bass musician playing `Phrase 32` can consider `Phrase 13` an appropriate response.

The technique can be described more formally with the following pseudo-code:

```
val cases: List[(Problem, Solution)]
for each instrument:
    for each phrase played by it:
        for each instrument other than this:
            successor = findPhraseChronologicallyAfter(phrase)
            cases += (phrase, successor)
```

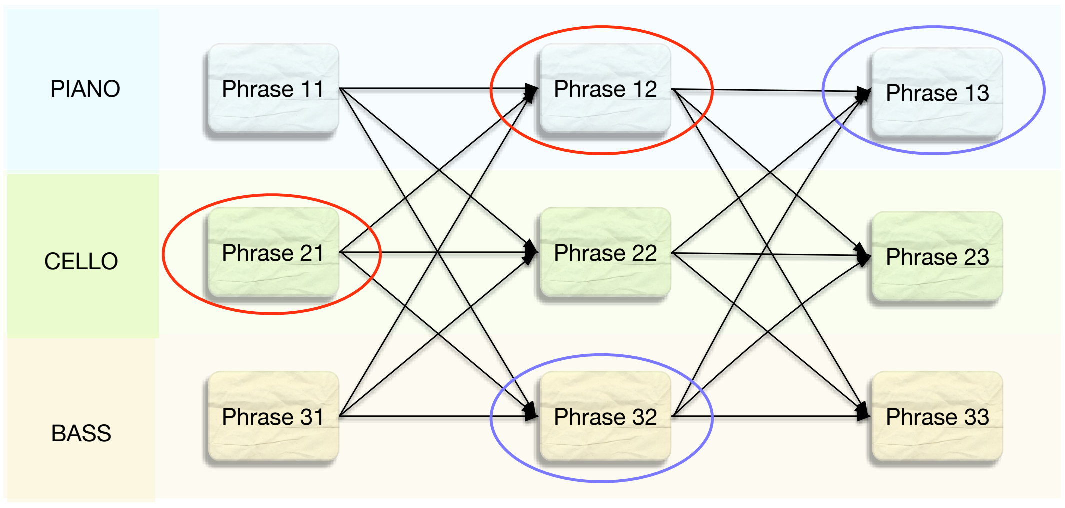


Figure 5.11: Extracting Relationships Between Sub-phrases

Figure 5.12 shows a few `(problem, solution)` pairs extracted following the reasoning illustrated above: `Phrase 21`, played by a cello player, is identified as a potential problem to which `Phrase 12` is a solution.

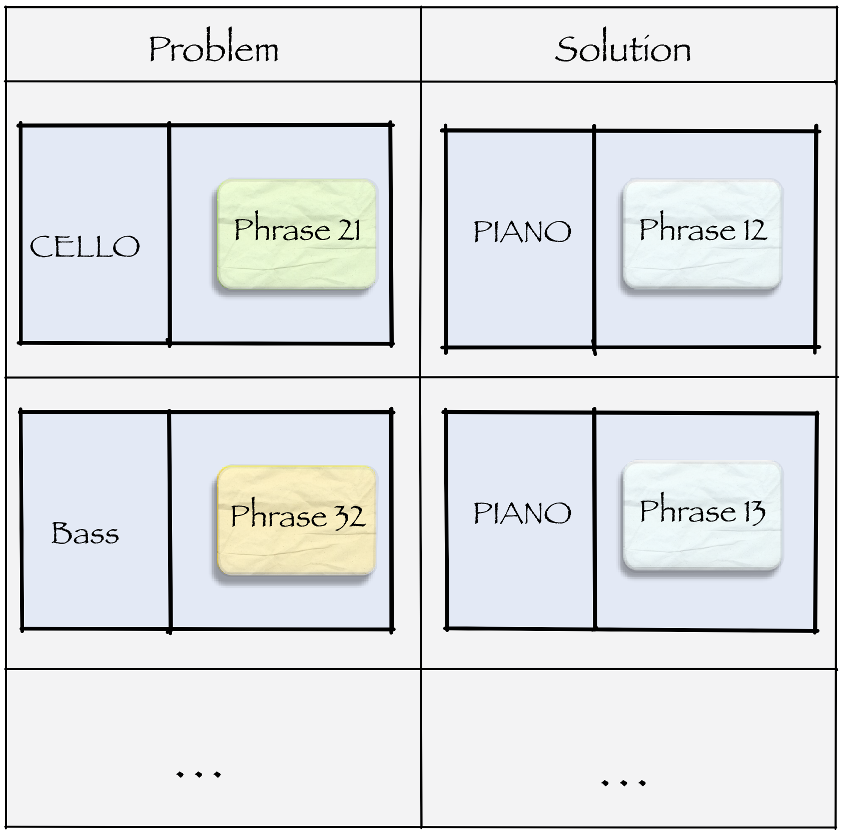


Figure 5.12: Example Cases Extracted From File

5.4 Storage

In this section we describe the way we store information in our knowledge base. It is important to keep in mind the fact that, since we are building a real-time system that improvises music, we need fast response times when accessing its knowledge base. More details on the performance of our storage system are given in Section 8.1.

As already stated, the idea underlying `Improv`'s learning is that if two phrases are similar, then their immediate successors will be similar too.

Because of this, as described in Section 5.3.2, we have extracted a set set of tuples each of which represents a phrase and its successor.

In order to take advantage of this information during our music generation process we need to store it in a way that makes it easy for a musician to find a set of similar phrases to the ones he has heard.

To achieve this we need mainly two things: a similarity measure and an implementation of the `KNearestNeighbours` (`KNN`) algorithm. `KNN` is an algorithm which, given a set of elements, an element and a function that computes the similarity between two elements, returns up to k elements that are most similar to the provided element.

Naïve implementations of the `KNN` algorithm, simply compute the similarity measure between all the points in the database and the target point, sort them by descending order of similarity and return the top k elements. This approach has a lower bound of $O(n \log n)$, which is the cost of sorting n elements with fast algorithms like `Heapsort`.

If k is a small number (less than $\log n$ for reasonable values of n), one could potentially avoid sorting the whole set of elements by distance but simply compute the similarity between all the elements and the target and then perform k linear scans to find the k most similar elements. The complexity of this approach is $O(kn)$. Even though one comes up with an even better selection algorithm to select the k biggest/smallest elements in a list, the complexity of this approach is still bound by the $O(n)$ traversal of all the elements in the database to compute their similarity to the target.

It is possible to reach sub-linear performance by adopting a different approach that does not involve having to compute the similarity measure for all the entries in the database.

Since musical phrases can be represented as point in a high-dimensional space (see Section 5.4.3 for more details), one can take advantage of a special data structure, the kd-tree [26], to index data and reduce the search space. Thanks to their space partitioning properties, it is possible to perform the `KNN` algorithm in roughly $O(k \log n)$ [27] [26]. For more details about this data structure, please refer to Section 5.4.1.

5.4.1 Introduction to KD-Trees

A kd-tree is a specialisation of a Binary Space Partitioning tree which can be used to efficiently organise points in a k-dimensional space. Each node in the tree represent a k-dimensional and each non-leaf node in the tree creates a binary partition of the space along a given axis. Each node introduces a hyper-plane perpendicular to one of the axes, so that all the points on one side of the hyper-plane will be in one sub-tree, while all the points on the other side will be in the other sub-tree.

For example, lets consider the 2d-Tree depicted in Figure 5.13; we can notice that each node represents a point in a 2D space as a pair of `(X, Y)` coordinates. We can also notice that each node has one of the coordinates underlined and in red colour: that coordinate represents the position of the line[2], perpendicular to that axis, which partitions the space into two regions.

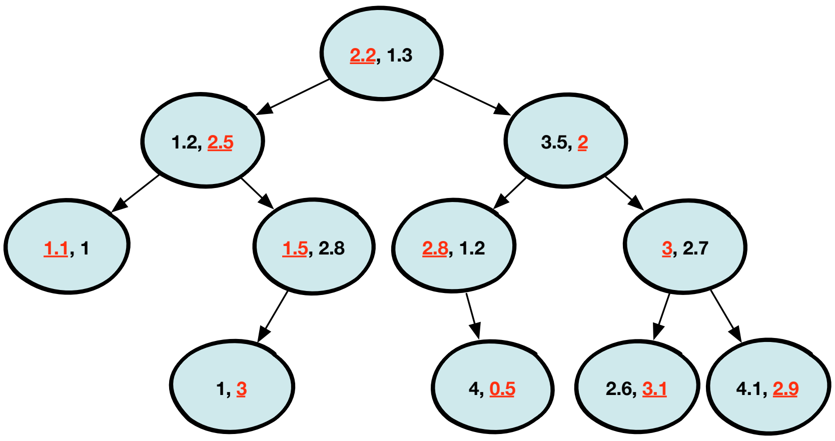


Figure 5.13: Complete 2D-Tree

In Figure 5.14 we can see the effect that choosing `(2.2, 1.3)` as our root node has in terms of partitioning the space. The pivoting dimension chosen for this split is the `x axis`, therefore a line perpendicular to it and passing through the point chosen as our root will divide the space into two regions, one where all elements have x coordinates < 2.2, the x coordinate of our root, and the other where elements will have x coordinates $>= 2.2$.

If we continue our inspection of the tree, we can notice how all the nodes to the left of the root have x coordinates less than that of the root, while all the nodes to its right have x coordinates greater than that of the root. The same applies all the other nodes: the nodes in the left sub-tree of a node which partitions the space according to a given dimension (highlighted in red) will have a value for that coordinate less than that of the node, while those in the right sub-tree will have a value greater than that.

[2]It would have been a hyper-plane in a higher-dimensional space

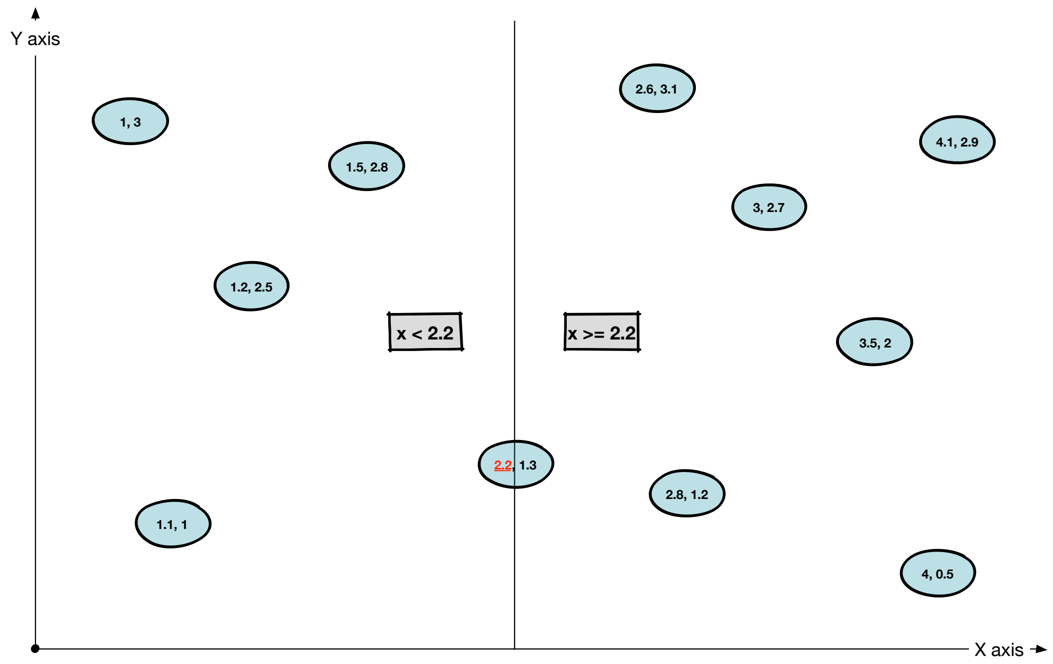


Figure 5.14: Partitioning Space Around Tree Root

The following pseudo-code gives an overview of how to build a kd-tree from a provided set of points with k dimensions.

```
Given a set of k-dimensional points

def buildTree(points):
    if set of points is empty:
        return

    pivotDimension = choosePivotDimension(points)
    pivotPoint = select point from points taking into account pivotDimension

    (leftPoints, righPoints) = partition(points, pivotPoint, pivotDimension)

    node = Node(pivotPoint)

    node.left = buildTree(leftPoints)
    node.right = buildTree(rightPoints)

    return node
```

The `partition` method used in the pseudo-code above divides the set of points into two sets: one which has all values for the `pivotDimension` less than the value of the `pivotPoint` for that dimension (the `leftPoints` set), one which has values greater than that (the `rightPoints` set).

The `choosePivotDimension` method, instead, selects a dimension which will be used to partition the space at the current node by analysing the current set of nodes.

More detail on the selection of the pivoting dimension and the point which will be inserted next in the tree are provided in Section **Balancing**

Finally, Figure 5.15 shows how each node in the tree partitions the space in regions.

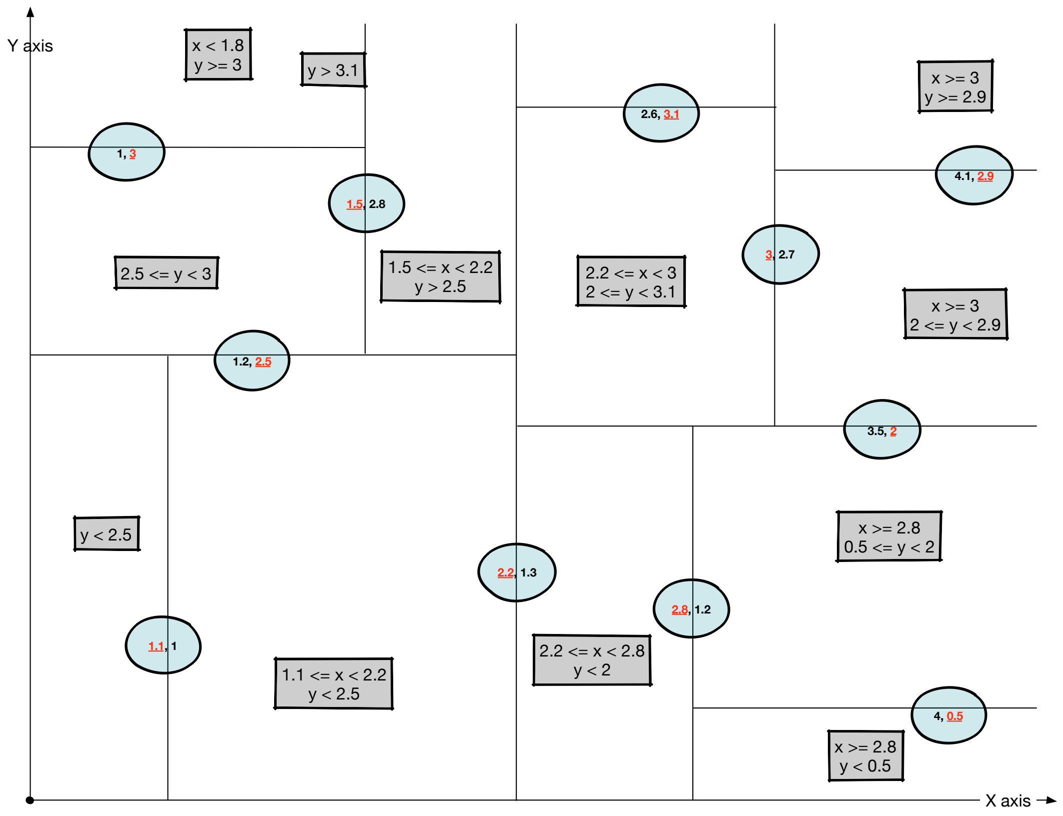


Figure 5.15: Space Partitioning of the 2d-Tree of Figure 5.13

Balancing

In order to achieve good performance during search operations, the tree needs to be reasonably balanced[3]. However, due to the fact that the space is partitioned across different dimensions at each non-leaf node, re-balancing techniques used for AVL trees, such as rotations, cannot be used, as they could break the ordering along one of the dimensions.

Because of this limitation, it is easier to build a balanced tree when all data is available at once, rather than keeping it balanced as it is built.

There are two decisions to be made when building a balanced kd-tree from a dataset: to which axis the splitting hyper-plane associated with each node should be perpendicular, and what point to choose as the next node while the tree is being built.

A common approach to the selection of the pivoting dimension is to cycle over them: nodes at level n in a k-dimensional tree will partition the space around the $(\ n \ \% \ k\ )^{th}$ dimension.

More advanced ways of selecting the pivoting dimension exist, however they are more complex to implement or lead to benefits only in some circumstances. For example, as mentioned by [26], the technique described by [28] relies on splitting the space along the dimension with maximum variance. This should result in square regions, as it is unlikely that the maximum variance will be along the same dimension twice in a row. Apparently, though, this works

[3]That way each node will be at approximately the same distance from the root

well with uniform distributions of points while not so well with skewed distributions, in which case the partitioned hyper-regions will tend to be long and thin.

Once a pivoting dimension has been selected, it is necessary to choose the point whose coordinate will determine the position of the perpendicular splitting hyper-plane. A common approach is to select the point whose coordinate is the median of the dataset with respect to the selected dimension. The downside of this approach is that computing the median each time we partition the space can be expensive, if the dataset is very large: computing the median of a dataset involves sorting it, which, using algorithms like `Quicksort` or `Mergesort` can be done in $O(n\ \log n)$ time. Since we have $O(\log n)$ splits (height of a balanced tree), we would end up with $O(n\ \log^2 n)$ complexity. To speed things up, Brown [29] suggests that points be sorted along each dimension (e.g.: first sort along first dimension, then sort along the second dimension, ...) prior to building the tree. During the building of the tree, the initial sorting is preserved, thus eliminating the need for further sorting. With this pre-processing step, the complexity of building a balanced tree is reduced to $O(kn\ \log n)$, as we have to sort the dataset k times, one per each dimension.

Improvements on java-ml's Implementation

The implementation of a kd-tree in the `java-ml` library had two main limitations: it did not expose an iterator and it did not provide a way of re-balancing of the tree. Whenever a node was removed from the tree, it was simply marked as `deleted`, potentially leaving the tree unbalanced.

We improved upon the original implementation by providing an in-order iterator which also skips nodes marked as `deleted`. On top of that, we introduced a self-balancing mechanism which kicks in as soon as the number of nodes marked as `deleted` in the tree goes above a certain threshold. The re-balancing of the tree can also be manually triggered by calling the `rebalance()` method.

As mentioned above, usual re-balancing techniques such a AVL-tree rotations are not suitable for kd-trees, however, if we are given an existing tree, we can extract all its nodes and build a new, balanced, tree without the nodes marked as `deleted`. We, therefore, made use of the technique described above and by [29] to re-balance the tree. The technique improves

5.4.2 KDIndex

Section 5.4.1 describes a data structure, the kd-tree, that is useful to store points in a high-dimensional space and perform `KNN` efficiently.

In our project, we construct the description of a phrase by the means of feature extraction (described in Section 5.4.3). This process causes a loss of information about the phrase: while the description of a phrase is good enough to identify it, you cannot actually play it.

This process of feature extraction is useful to describe the `problem`, a phrase being played, but not its solution, a phrase that is a good candidate successor. This is because, while we only need to identify a `problem` to find similar `problems`, we actually need to be able to play their solutions.

Since our system improvises music, we should try to provide as fast access as possible to our knowledge base; this can be achieved by keeping it in memory.

A first approach would be that of letting each node in the kd-tree have a problem's description as its key and store the solution as its value. This, however, could lead very high memory usage if the knowledge base grows to be really large. The reason why this might not be ideal is that eventually we would have to keep only parts of the index in memory, while writing the rest to disk, thus slowing down the search process.

In order to mitigate this, we created our own data structure, the `KDIndex` which stores problem descriptions and solutions separately. This data structure has two components: our enhanced version of `java-ml`'s implementation of a kd-tree, and an interface for key-value store.

Whenever a pair `(problem, solution)` is added to the `KDIndex`, the solution is stored in the key-value store, which returns a unique identifier for it. A node is then inserted in the kd-tree with the problem description as its key, and a `List[UUID]` as its value. If two problems with different solutions share the same description, the node will hold the unique identifiers of both solutions as its value.

This process is completely transparent to the user: the interface exposed both accepts and returns pairs of `(problem, solution)`.

This allows us to keep the index with the problem descriptions always in memory[4], while the solutions can be loaded on demand. Two implementations have been provided a for the key-value store: a `Hazelcast` wrapper, and a `MapDB` wrapper. The former is a distributed in-memory key-value store, while the latter is a local in-memory key-value store that supports spilling to disk: if the map grows to big to be kept in memory, parts of it will be written to disk.

5.4.3 Feature Extraction

As mentioned above, a musical phrase can be represented as a point in a high-dimensional space. In order to do so, however, we need to define the dimensions of such space.

We can, for example, define a set of features that we believe describe well any phrase. Once that is done, we can create a space by associating one dimension to the domain of all possible values of a feature. Assuming we've selected a set of n features, we end up with an n-dimensional space where any possible musical phrase is represented by a point.

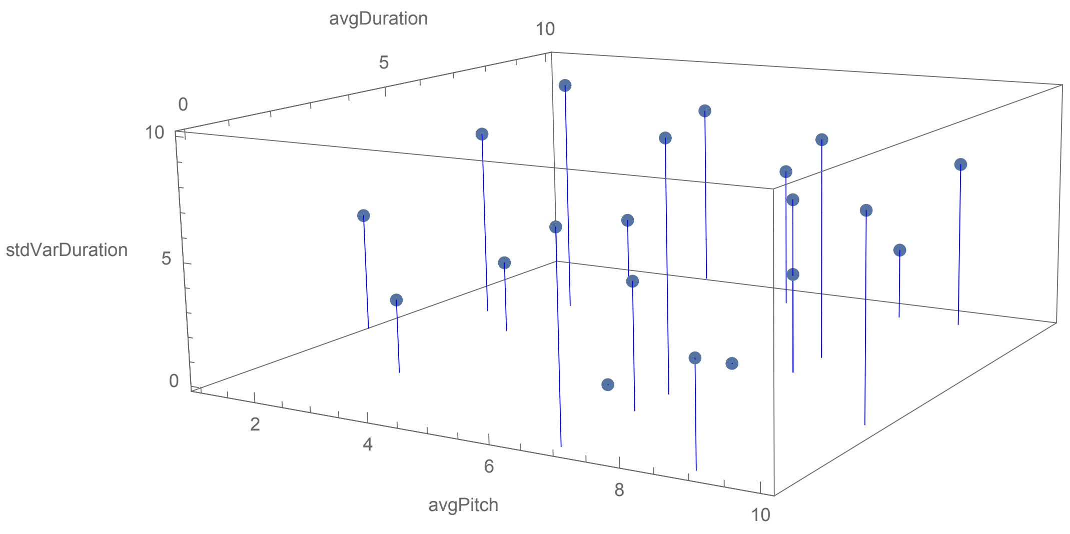


Figure 5.16: Phrases in a 3D Space

[4]Currently we extract an array of 163 doubles as our feature array. This means that the description of a problem will take up 1 304 bytes (∼1KB). Considered this, 1GB of RAM will hold 1 000 000 musical cases, which seems like a fair amount

Feature extractors have been implemented with inspiration from [30] and [31]. A few of them are described below:

- ◇ `AverageNoteDurationExtractor` ⇒
  This extractor computes the average duration of notes in a phrase (rests are ignored)
- ◇ `AverageMelodicIntervalExtractor` ⇒
  This extractor computes the average difference in pitch between consecutive notes
- ◇ `AverageStartTimeIntervalExtractor` ⇒
  This extractor computes the average time difference between the start times of notes. Given two notes, this metric potentially combines together the duration of the first note and the duration of the rest between two notes (if there is one).
- ◇ `DirectionOfMotionExtractor` ⇒
  This extractor computes the ratio between the number of times a note had a higher pitch than its predecessor (upward motion) and the number of times it had a lower pitch (downward motion).
- ◇ `AverageDurationExtractor` ⇒
  This extractor computes the average duration of all elements in a phrase, including rests.
- ◇ `IntervalBetweenMostCommonPitchesExtractor` ⇒
  This extractor computes a histogram of the pitches, selects the two most frequent ones and computes the interval between them.
- ◇ `MostCommonMelodicIntervalExtractor` ⇒
  As described above a melodic interval is the difference in pitch between two adjavent notes. This extractor returns the interval that appears most frequently in the phrase.
- ◇ `NoteDensityExtractor` ⇒
  This extractor computes the ratio between the number of notes in a phrase and the phrase duration (computed as the sum of all the durations of the elements in the phrase, including rests).
- ◇ `RepeatedNotesExtractor` ⇒
  This extractor computes the ratio between the number of repeated notes and the number of notes in a phrase. A repeated note is a note that is followed by a note with the same pitch.
- ◇ `VarianceOfNoteDurationExtractor` ⇒
  This extractor computes the statistical variance in the duration of the notes in a phrase.
- ◇ `TempoExtractor` ⇒
  This extractor simply extracts the tempo of a phrase.
- ◇ `KeyExtractor` ⇒
  This extractor simply extracts the musical key of a phrase. Musical keys are represented by the number of flats and sharps they have and their quality, minor of major[5].
- ◇ `BasicPitchHistogramExtractor` ⇒
  This extractor computes the histogram of the notes' pitches in a phrase. Since `MIDI` pitches range from 0 to 127, the feature produced by this extractor will have 128 elements.

Each feature extractor produces a single feature which can have one or more dimensions: for example, the `AverageNoteDurationExtractor` produces a one-dimensional feature, while

[5]A key with 0 flats and 0 sharps, for example can correspond to a C major scale or to an A minor scale.

the `BasicPitchHistogramExtractor` extracts a feature with 128 dimension (one per each possible `MIDI` pitch).

Feature extractors are then grouped together, their features are weighted[6] and combined together in an array of doubles which represents the phrase as a point in a high-dimensional space.

5.5 Optimisations

5.5.1 JMusic

The library provided a good starting point that helped with `MIDI` parsing. However, on multiple occasions it turned out to be necessary to patch the source code in order to obtain either better performance or introduce more functionality. An example of one such modification is the improvement of the performance of the retrieval a `Note`'s start time. The start time of a `Note` depends on its position in a `Phrase`, the start time of the `Phrase` and the duration of the `Note`s that come before it. Originally, in order to get a `Note`'s start time, one had to perform the following steps:

1. Find the `Phrase` to which the `Note` belongs[7]
2. Get the list of `Notes` for that `Phrase`
3. Perform a linear scan to find the index of the `Note` in the list
4. Call the method `getStartTimeForNote(int noteIndex)` on the `Phrase`, which iterated from 0 to the provided index, accumulating the duration of the notes in the phrase.

Clearly this method is inefficient as it requires $O(n)$ iterations to get the duration of a single `Note` in a `Phrase` with n `Note`s.

To improve this, the method `getStartTime()` was added to the `Note` class, and an `Optional<Map<Note, Double»` together with `getNoteStartTime(Note note)`, which takes a `Note` instead of its index as argument, were added to `Phrase`. When `getStartTime()` is called on a `Note`, the following happens:

1. The `Note` calls `this.myPhrase.getNoteStartStartTime(this)`
2. If the map between `Note`s and start times is populated, the method simply returns the start time corresponding to the provided `Note`. Otherwise, the `Phrase` computes the start times for all the `Note`s and stores them in the map.

Whenever a `Note` is added to or removed from the `Phrase` the map is invalidated (i.e.: The `Optional` is set to `Empty`).

This reduces the complexity to amortised $O(1)$, while preserving the flexibility of being able to move the `Note` to another `Phrase` and getting its start time recomputed automatically[8].

[6]For simplicity, in the current version of the project each feature has the same weight, but in future it is possible to explore changing the weights to give more importance to more significant features

[7]fortunately, each `Note` has a reference to the `Phrase` it belongs to

[8]If the start time was set as a `Note`'s field, one would have to manually update it if the `Note` was moved to a different `Phrase`.

Chapter 6

Musical Generation

As mentioned previously, each musician in the system generates and plays a new musical phrase every time it receives a `SyncMessage` from the director. Every musician is independent from the others in the sense that it is only responsible for the generation and playback of their own musical phrase, without any sort of centralised computation taking place. However, if musicians did not take into account the fact that there are other musicians playing together with them, the music generated by each of them individually would likely not sound well together, especially with a large number of musicians. Imagine what would happen if a group of friends got together and started improvising while wearing earplugs: unless they knew each other very well, the result would probably sound uncoordinated and not pleasant, even if each of them, separately, was playing a good tune.

In order to overcome this problem, as mentioned in Section 3.2.1, each musician accumulates a musical context made up of information about what the other musicians are playing. Each musician can then use such information to produce a new musical phrase. The following sections will explain how this is accomplished.

6.1 Composing a Musical Phrase

As mentioned in Section 3.3, each musician has a `Composer`, a module responsible for the generation of musical phrases. When a musician receives a `SyncMessage` from the director, it provides its composer with a set of musical phrases and the instruments with which they were played by other musicians since the previous `SyncMessage`.

In this sense, the generation works a bit like a Markov chain: what a musician is going to play next only depends on the previous musical context, while musical contexts before that are discarded.

The generation process, however, relies on a `Genetic Algorithm` whose initial population is generated by using a `Case-based Reasoning` approach: elements of the initial population are retrieved from our knowledge base by finding phrases in it that are similar to the phrases in the musical context. This approach is depicted in Figure 6.1

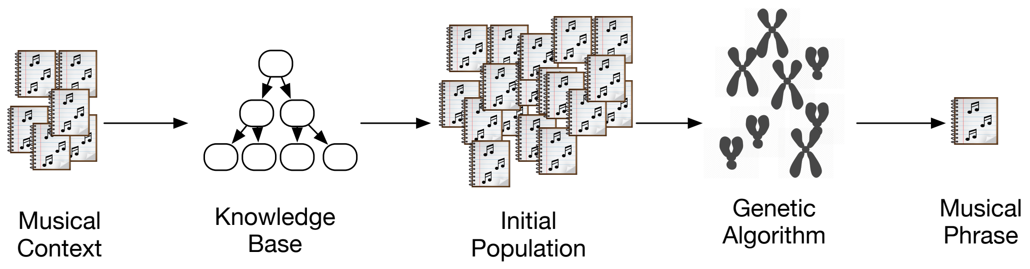


Figure 6.1: Generating a Musical Phrase from a Musical Context

6.2 Generating the Initial Population

As mentioned in Section 2.1.4, one of the problems with genetic algorithms is that they only work well if their initial population contains good candidate solutions. This is why an initial population needs to be built with care.

One of the fundamental ideas in this project is that if we take the successor of a musical phrase, it will be a suitable candidate successor for a similar musical phrase. Therefore, in order to build the initial population, each musician can do the following:

- ◇ For each `(phrase, instrument)` pair in the musical context:
    - ◇ Find the k most similiar phrases in the knowledge base
    - ◇ For each of those phrases, add its successor to the initial population
- ◇ Filter out any phrase in the initial population that was played with an instrument different than that of the musician

Figure 6.2 illustrates one of the steps of this process: a musician that plays the piano has heard three other musicians playing some phrases, which have been added to its musical context. The musician starts looking for phrases in the knowledge base that are similar to the first phrase in its musical context, in the example `Phrase 314`. To do so, it generates a description of `Phrase 314` by the means of feature extraction (described in Section 5.4.3) and queries the knowledge base for the three[1] most similar phrases in the knowledge base. Once it has collected all the successors of the similar phrases, it filters out all the solutions that are played with a different instrument (in the example `Phrase 33` is played with a `Cello`) and adds the remaining phrases to the initial population.

The musician would then repeat those same steps for the rest of the phrases in the musical context, in the example `Phrase 159` and `Phrase 265`.

[1]The system actually looks for ten nearest neighbours, but, since they would not fit in the diagram, and the number is arbitrary, three will do just fine as an example.

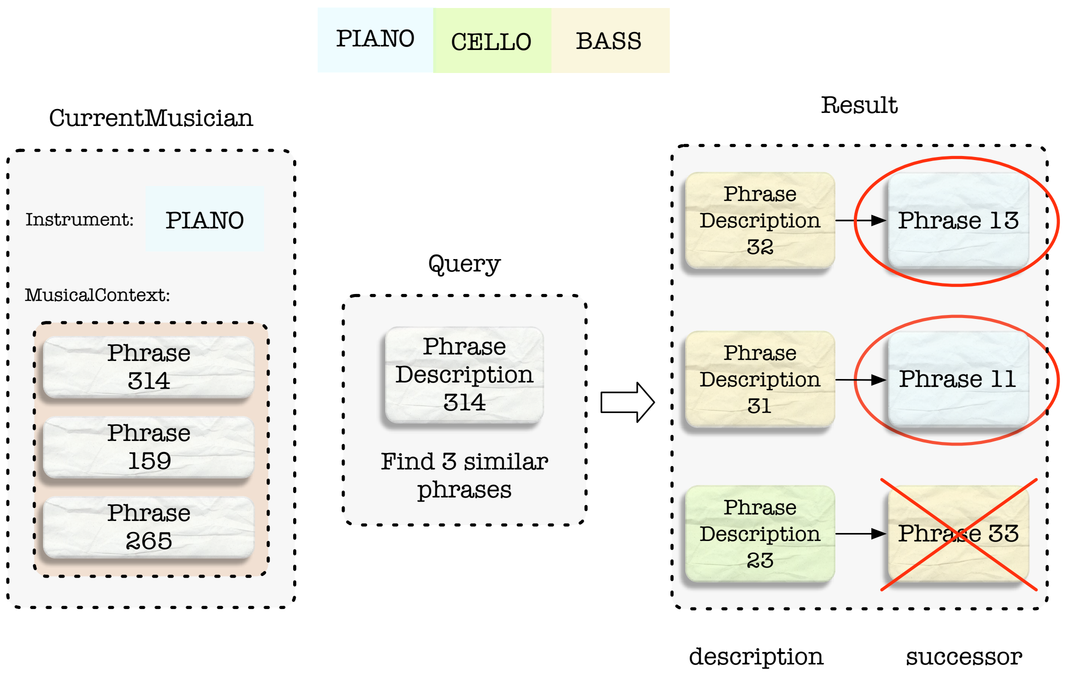


Figure 6.2: Generating the Initial Population based on the Musical Context

6.3 Evolving the Population

Once an initial population has been generated, as described in Section 6.2, we need to evolve it and then select the fittest individual as the phrase we are going to play next.

At the beginning each iteration of the genetic algorithm, every phrase in the population is assigned a chance of survival proportional to their fitness according to our fitness function. A portion[2]of the population is then selected by sampling at random the population according to the chances of survival of each individual. Phrases that have survived are paired together and a new phrase is generated as we will describe in Section 6.3.1. After a number[3] of iterations, the fittest individual is chosen as our new phrase.

The following sections describe the design of chromosomes and our fitness function.

6.3.1 Chromosome Design

Since we had already developed a set of functions to deal with phrase segmentation, merging and splitting, we have decided to design the chromosome representation of a phrase a sequence of chords and notes. This is exactly equivalent to a single-voice phrase in our representation.

Usually chromosomes, not only have the same length, but also have corresponding cross-over points, as it has been illustrated in Figure 2.1 in Section 2.1. However, since, as we have seen in Section 5.3.1, different phrases come with different optimal segmentation points, it would not be ideal to simply cut them at pre-defined points. Because of this, we will not use a cross-over mask, but will instead use a different approach:

[2]This proportion is arbitrary and should be tweaked to see if it influences the quality of the improvisation
[3]This number is arbitrary and should be tweaked by through experiments.

- ◇ If the phrases are not single-voice phrases, convert them by using the phrase merging algorithm described in Section 6.3.1
- ◇ Segment each phrase with one of the techniques described in Section 5.3.1
- ◇ Select a segment from each phrase alternatively until no more segments are available in the shortest phrase

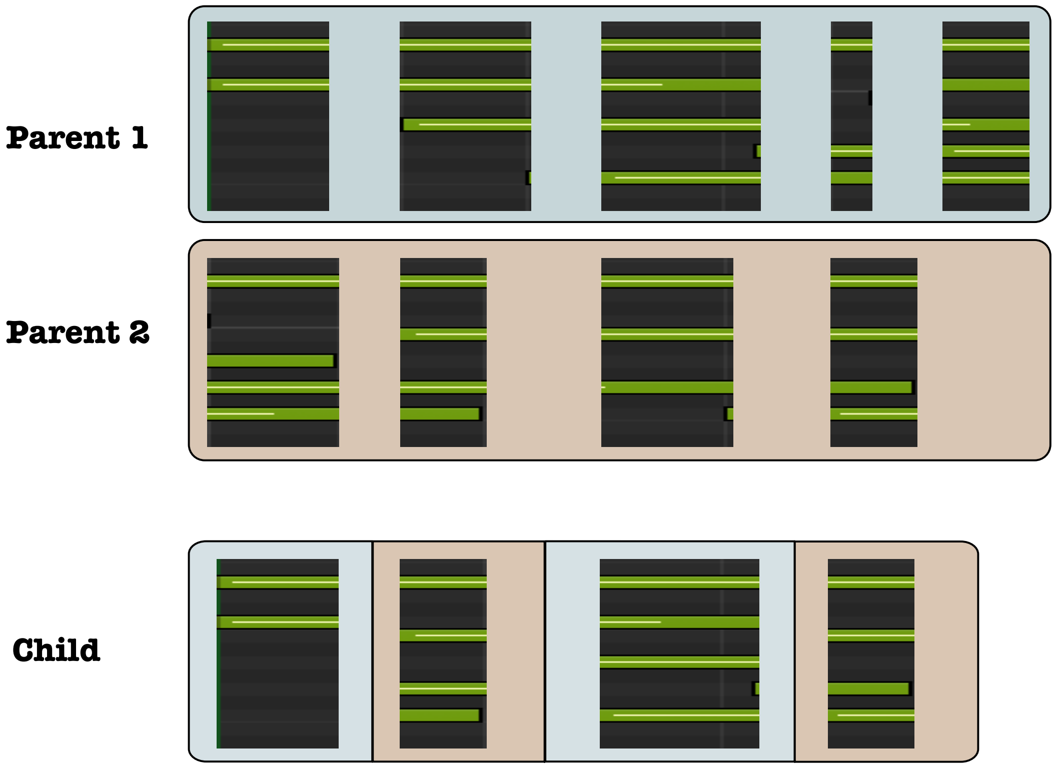


Figure 6.3: Chromosome Cross-over

Since generated phrases can have different durations, this process is aware of the desired phrase length[4], and therefore can either truncate or scale phrases at the end of the breeding process. Both truncating a phrase or scaling it can potentially reduce their quality, so which one to choose will have to depend on a trial-and-error process. Early examples suggest that better results[5] are obtained with truncation rather than scaling.

Note that, in theory, a genetic algorithm should introduce random mutations on top of crossover, however we decided not to implement them lest spoiling the new phrase by introducing a random note or chord. However, since we are sampling at random

Finally, an alternative approach could be that of creating a new phrase by taking different features from the parents: for example one could generate a new phrase by taking the rhythmic properties (like start times and durations) from one parent, and the melodic properties (like pitch and loudness) from the other parent. However, due to the lack of time, this approach has not been experimented with and is left as a future experiment for the reader.

[4]Each musician selects the average phrase duration in its musical context as the target phrase duration
[5]According to the author's personal musical preferences

Phrase Merging

A multi-voice phrase is a phrase that is made up only of sub-phrases that are played concurrently. In order to transform a multi-voice phrase into a single-voice phrase we need to merge all its sub-phrases together while preserving the relative and absolute ordering. This can be achieved with the following algorithm:

```
phraseElements = List[MusicalElement]()
activeNotes = List[MusicalElement]()

times = (startTimes ++ endTimes).distinct.sorted

for (time <- times) {
    phraseElements += createChord(activeNotes)

  if (all active notes end at the same time) {
      phraseElements += endMarker
  }

  activeNotes = removeOrTruncate(activeNotes, time)
  activeNotes ++= getNotesStartingAt(time)
}

Phrase().withMusicalElements(phraseElements)
```

The algorithm retrieves a list of start and end times of each note in the phrase a sorts it in ascending order. It then traverses the list of times an merges into a chord[6] all the notes active in the phrase at that time. A note is considered active if the current time is at or after its start time and before its end time.

After that, the list of active notes is updated: the start time of all notes that are still active is updated to the current time[7], while all notes in the active that are no longer active are removed from the set. Finally, all the notes in the phrase starting at the current time are added to the list of active notes.

[6]If there only one active note, then it is simply added to the `phraseElements` without creating a new chord.

[7]Since all notes are immutable, what actually happens is that a new list of with modified copies of the notes is created

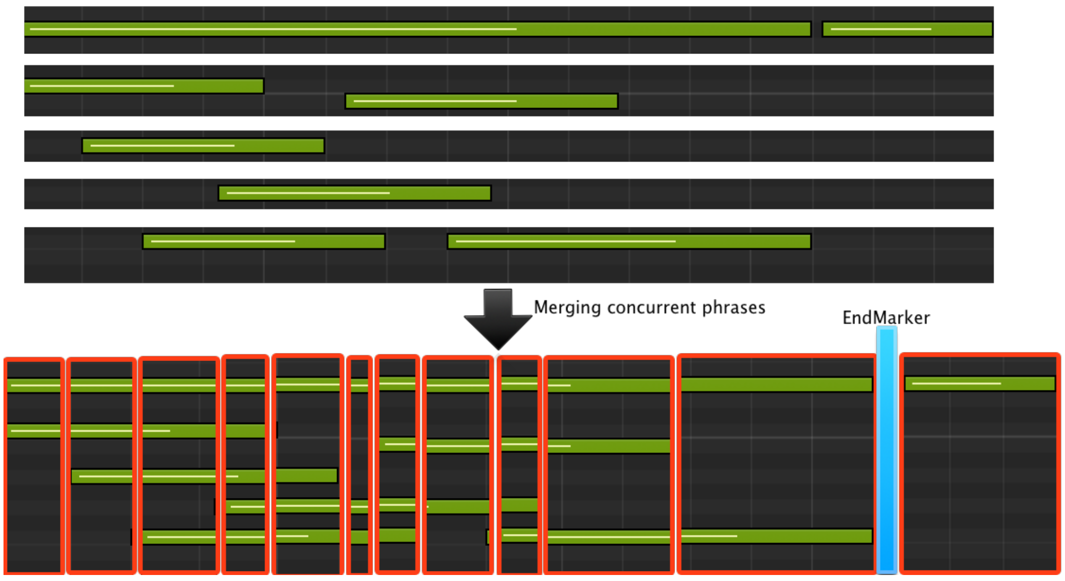


Figure 6.4: Transforming a Multi-voice Phrase into a Single-voice Phrase

An example of the algorithm's result is shown in Figure 6.4. Each green line in the picture represents a note while each black band represents a sub-phrase. The five concurrent sub-phrases in the the original multi-voice phrase are turned into a single-voice phrase that contains a sequence of chords (red boxes).

As we can see the two phrases are pretty much equivalent, but for one aspect: notes that were played and held in the multi-voice phrase are played multiple times in different chords in the single-voice phrase.

It is also possible to appreciate the purpose of the blue "EndMarker": if that piece of information was not present, one would not be able to tell if the note in the top phrase was originally repeated or not.

Phrase Splitting

Phrase merging is done for the sole purpose of simplifying a set of operations over phrases, like the cross-over. However, as mentioned above, it has a side effect: notes that were not repeated in the original phrase can be repeated in the merged phrase. This can, at times, be unpleasant to the ear. Therefore, once all operations have been performed on the merged phrase, we can use a splitting algorithm to recover the the original multi-voice phrase. A simplified[8] version of the algorithm's code is provided below:

```
def splitPhrase(phrase: Phrase) = {
  activeElements = List[MusicalElement]()
  // # lists = # notes in largest chord
  phrasesElements = List[List[MusicalElement]]()
  musicalElements = phrase.musicalElements.sortedBy(startTime)

  for (musicalElement <- musicalElements) {
    musicalElement match {
```

[8]The code is similar to the actual implementation, however some lines have been replaced with lines that are easier to read, but will not compile. For the actual code, please refer to the source codes archive submitted on CATE

```
        case chord: Chord =>
          activeElements = chord.notes
        case elem =>
          activeElements = List(elem)
      }

      rest = new rest with same start time and duration as ‘musicalElement‘
      silentPhrasesCount = phraseElements.size - activeElements.count
      activeElements ++= list of #silentPhrasesCount copies of ‘rest‘

      for (elem <- activeElements) {
          addToPhrase(phrasesElements(idx), rest)
      }
    }

    phrases = create a sub-phrase for list in ‘phrasesElements‘
    Phrase()
      .withMusicalElements(phrases)
      .withPolyphony()
  }
```

In a multi-voice phrase a chord is represented by the notes that belong to it being on different concurrent sub-phrases at the same time. Because of this, when splitting a phrase, we need to find what is the size of the largest chord in it (i.e.: how many notes that chord contains). We will then know the number of sub-phrases necessary to represent the single-voice phrase as a multi-voice phrase.

Once a list for each sub-phrase has been initialised, the algorithm sorts the musical elements by their start time and starts traversing them. Chords are expanded into the notes they contain. Each note is then added to a different list of phrase elements. If there are not enough notes to be added to all the existing lists of phrase elements, a rest with the same start time and duration as the current notes will be added to the remaining lists.

The `addToPhrase` will inspect the last element of a phrase before adding the new element to it, if the current element matches the previous, that is to say, they are either both rests, or notes with the same attributes, it will merge them into one and update[9] the last element in the phrase. This is a crucial step as it allows for a correct reconstruction of the multi-voice phrase without repeated notes. Also, to prevent repeated notes from being merged when they were not, as mentioned in the previous section, an `EndMarker` can be added during the merging stop. If an `EndMaker` is the last element of a phrase, it is simply removed and the new element is appended to the phrase without merging it with the previous.

At the end of the algorithm each list of phrase elements is converted to a sub-phrase and a multi-voice phrase that contains all the sub-phrases is returned.

6.3.2 Fitness Function

If coming up with a good chromosome design is a difficult task, even more so is designing a good fitness function. One of the main problems in our case is that we are trying to quantify something that, as humans, we can feel but not really explain. That is, when you hear a good piece of music, you can tell that it sounds good, but most often you would not be able to say why.

[9]A custom implementation of a `ListBuffer` had to be provided since `Scala`'s default implementation, despite using linked lists, did not allow for efficient update of the last element in the buffer: it was necessary to copy the whole list to z.

Because of the difficulty of capturing and formalising what it means for a piece of music to sound good, we approached the problem from a different angle: we tried to teach the system, by example, what we consider good music. More technically, we trained an Artificial Neural Network as our fitness function.

The training process for the `ANN` can be summarised as follows:

- ◇ The user selects a `MIDI` file as input
- ◇ Using the technique described in Section 5.3.1, a set of musical phrases is extracted from the file
- ◇ Each phrase, by the means of feature extraction (see Section 5.4.3, is turned into an array of doubles, which represents the description of the phrase
- ◇ Each phrase is then played back to the user, who is asked to enter a decimal rating from 0 to 10
- ◇ A pair `(description, rating)` is then added to a data structure that holds the training data for the `ANN` and the user is asked if they want to continue

The data structure also keeps track of the last phrase that was rated in the file, so that the rating process can be stopped and resumed at any time, without loss of data.

Different `MIDI` files can be rated, even partially, independently and their data can be merged before training. This way we can speed the data generation process up by splitting it among different users[10]. Once enough data has been collected, a `MultiPerceptron` Neural Network with learning rule `Back Propagation with Momentum` has been trained. The reason why this specific type of `Neural Network` has been chosen is that it was the one that was most familiar to us.

Unfortunately, the `ANN` implementation we are using, `neuroph`, does not support cross-validation. Cross-validation is the process of splitting the training data into chunks and using some of them to train the network, while using the rest to make sure the network is not over-fitting. The idea behind a Neural Network is that it is able to capture the structure hidden in the data by modifying its weights to minimise the average output error during training. However, if a Neural Network over-fits the data, it means that it has learnt the weights necessary to replicate the expected output of the training set so well that it will not perform well on different data.

To mitigate this, we try not to let the average network error drop below a certain threshold[11]. On top of that, we train the network on a different set of musical files than those stored in the knowledge base, so that newly bred phrases have a better chance at surviving[12].

[10]In our case, friends and family have helped

[11]This is achieved by supervising the training process and observing the network error curve, manually stopping the training process before it reaches a very low value

[12]If the network was over-fit on the contents of the knowledge base, well rated elements in the initial population would most likely supersede the newly bred ones, thus making the improvisation more like a rehash of the knowledge base

Chapter 7

Playing Sounds

Once a musical phrase has been generated, a musician needs to actually play it. In order to do so, each musician is equipped with an `Instrument`. Each instrument knows how to turn our musical representation into sound. Two libraries have been tested for this purpose, with the latter yielding better results, despite having more limited functionality and instrument quality.

7.1 Overtone

`Overtone` is a library for `Clojure` that uses a famous audio engine, `SuperCollider`, to provide a higher-level musical library. `Overtone` was initially considered a good candidate because of the features it offered:

1. A library of musical functions (scale/chord generators, rhythms, arpeggiators)
2. Good support for concurrency
3. An API for fetching sounds from `http://freesound.org/`, so that high quality sampled instruments could easily be used.

However, `Scala` was chosen as a language for the project due to its better readability and no `Overtone` client existed for either `Java` or `Scala`. Because of this, a client had to be implemented from scratch.

Different alternatives have been explored while trying to build such client, including loading overtone on a `Clojure` `REPL`[1] local server. This option has eventually been ruled out because of the extra latency it would introduce.

Fortunately, both `Scala` and `Clojure` run on the `JVM`, and a very rudimentary just-in-time `Clojure` compiler library exists for `Scala` allowing for `Clojure` code to be dynamically loaded and executed from within a `Scala` application.

There were a few complications along the road, such as the fact that `Overtone` required `SuperCollider` to be installed on the system, which would greatly complicate the deployment of `Improv`. To overcome this, native libraries from `SuperCollider` were included in the build path of the client (and exported in the `jar` artifact). This way the libraries required by `Overtone` could be extracted to a temporary directory at run-time and added to the `JAVA_PATH` through a `JVM` parameter.

[1]Read Eval Print Loop: a basic interactive language shell, where one can input a command, or a series of commands, have them evaluated and read their output on the standard output

The client allows a very basic interaction with Overtone: Clojure functions and commands can be sent to it as a string, which is then dynamically interpreted in a Clojure environment that has Overtone loaded as a library. To provide easier interaction with Overtone a layer of helper classes has been developed to make tasks as loading instruments and playing notes easier.

Eventually a big flaw emerged: the only way to shut down a musician that used an OvertoneInstrument turned out to be issuing System.exit(). We couldn't find any other way to close the Clojure compiler or exit from Overtone. This wouldn't be a big problem if each musician, the director and the message queue, were all running on different machines. However, if all the agents were running of the same machine, killing one actor would result in the shutdown of the whole system, which is undesired.

7.2 JFugue

JFugue is a library with multiple features: it can parse MIDI files, synthesise music and has built-in notions of music theory. Its MIDI-parsing capabilities, however, are limited since it simply provides listeners for a parser without actually building a symbolic representation as JMusic does. On the other hand, its music playback capabilities are superior to those of JMusic: the latter does not offer a simple way of selecting an instrument to play a given sequence, while JFugue does.

While some POJOs[2] have been provided by the author of the library, if one wants to achieve good control over and music playback, one has to use JFugue's Domain Specific Language Staccato.

Staccato's notation is quite simple: each note is represented by its name followed by the octave in which it should be played and its duration. For example "@1.4 A5/0.2" means that at beat[3]1.4 the note A will be played on the 5th octave for 0.2 beats. On top of that, specifying what instrument should be used to play a specific melody is a simple as prepending "I[InstrumentName]" in front of it.

It is pretty easy to our musical notation into Staccato: elements in phrases can be traversed and converted to their Staccato representation prepended with "@element.startTime". The only part where we could have had some issues was the conversion of multi-voice phrases, as they are made up of concurrent sub-phrases of elements with different durations and start times. However, Staccato deals with grouping together notes that start at the same time on its own. For example in "@0 A5/1 @3 B2/1 @0 D6/2", it's clear that A and D should be played together, despite not being adjacent. This way, whenever we need to convert a multi-voice phrase, we can simply process its sub-phrases sequentially, without worrying about preserving the chronological order of musical elements.

JFugue also provides a music player that takes a Staccato string and plays it. However it had a limitation: despite using a MIDI sequencer that was running on a separate thread, it was busy waiting for it, unnecessarily making its play method a blocking call. This caused a problem in the system: since every musician makes use of an instrument, if the play method is a blocking call, the musician will not receive any messages while it is playing. This is not ideal, since musicians still need to react to messages from the director and other musicians in the system while they are playing. Because of this we patched the source code of the library so that the play method no longer was a blocking call. The player class was also enhanced with the Observer pattern: once the player is done playing, it notifies all its observers with the event FinishedPlaying.

[2]Plain Old Java Objects: objects whose sole purpose is to hold data
[3]JFugue measures time in beats, which then depend on the tempo in Beats Per Minute (BPM) of the phrase to be converted in actual time

However, JFugue's player turned out not to perform very well in a distributed environment: music playback would start with unpredictable delays or not happen altogether. This might have been caused by the way it managed Java's MIDI sequencers. The problem was solved in the end by making use of JFugue to turn a Staccato string into a playable MIDI sequence, and then by implementing our own wrapper around Java's MIDI sequencer to actually play it.

Chapter 8

Evaluation

Evaluating a system that improvises music not an easy task: the fact that a piece of music sounds good is strongly influenced by individual tastes and is not easy to formalise. Because of this, most of the experiments reported in this section aim at observing the properties of the generated music.

In the first sub-section we will analyse the performance obtained when retrieving information from our knowledge base comparing it to an approach that does not use a kd-tree to index data.

We will then analyse the performance of the two variants of the LBDM phrase splitting algorithm proposed in Section Local Boundary Detection Model.

Finally we will try to determine whether the music generated by the system is repetitive or not.

8.1 Performance of a KDTree

As mentioned in Section 5.4.1 using a kd-tree as the data structure to index our knowledge base has turned out to be a key point to making our implementation able to meet the time constraints of a system that improvises music in real-time.

Figure 8.1 illustrates the time it takes to find the 10 nearest neighbours of a random target point. One of the curves represents the performance of performing this operation on a kd-tree, while the other represents the performance of performing it over a flat database, that is a database where data is not spatially indexed.

To perform the search over the flat data the following approach has been used:

- ◇ Calculate the distance between each point in the database and the target point
- ◇ Sort the elements in ascending order by distance
- ◇ Select the top 10 results

By looking at the graph it is clear that the performance of the kd-tree supersede that of the flat db: with 70 000 total elements in the knowledge base, it takes the kd-tree only 69 milliseconds to find the 10 nearest neighbours, it take the other approach 5000 milliseconds (5 seconds). Clearly 5 seconds is not an acceptable amount of time in a real-time system.

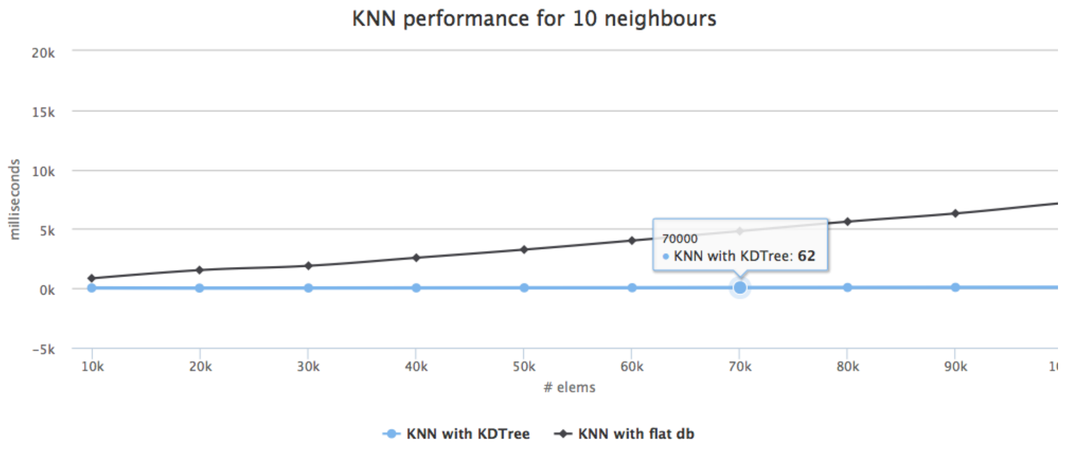


Figure 8.1: Performance Comparison between KD-Tree and Non-Indexed Data

The performances over the flat database can be improved by avoiding to sort the whole case base:

1. Calculate the distance between each point in the database and the target point: create a set of pairs `(element, distance)`
2. Perform a linear scan to find the element with the smallest distance and add it to the list of results
3. Remove the closest element from the set
4. Repeat from Step 2 until the 10 closest elements have been found

This approach is more efficient than the previous if the number of neighbours is generally smaller than $\log n$ where n is the number of elements in the set.

In Figure 8.2 we can see how the difference in performance between the flat db and the kd-tree has been reduced. However, the kd-tree still performs 4 times faster than the flat db: with around 400 000 elements in the knowledge base, the it takes the kd-tree around 500 milliseconds to find the nearest neighbours, while the flat db finds them in around 2000 milliseconds.

In conclusion, without the speed improvements introduced by indexing the data in a kd-tree, the real-time constraints of `Improv` could not have been met.

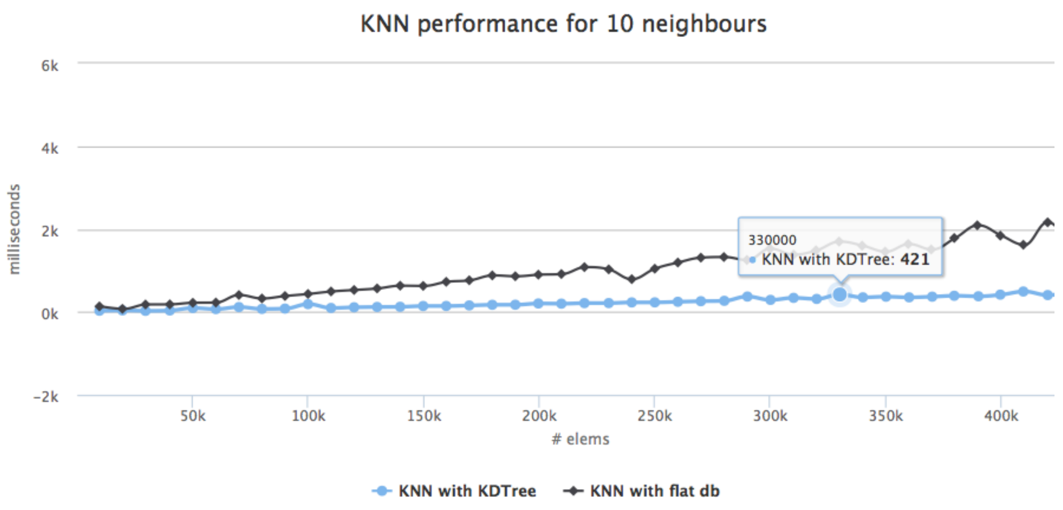


Figure 8.2: Performance Comparison between KD-Tree and Non-Indexed Data

8.2 Phrase Segmentation - LBDM

This section briefly compares the performances of the two variants of the LBDM phrase splitting algorithm proposed in Section Local Boundary Detection Model.

Each figure contains a plot of the different profiles involved in the computation of the LBDM boundary strength together with a horizontal line that represents the average peak value and is used as a threshold: whenever there is a peak in the combined profile that is higher than that line, that point is considered a good candidate phrase boundary for segmentation.

Figure 8.3 illustrates the phrase boundaries detected by the first variant of the algorithm, the one that, if presented with a multi-voice phrase, tries to run the LBDM analysis on the largest sub-phrase, hoping that it contains the most information about the melodic line of the phrase.

As we can see, the profile graph is very noisy: while a few peaks actually indicate splits that seem reasonable, there are too many of them above the threshold, which means that many small phrases would be produced if this variant were used for phrase segmentation. Phrases that are too small are not a desirable outcome as they do not carry much information with them.

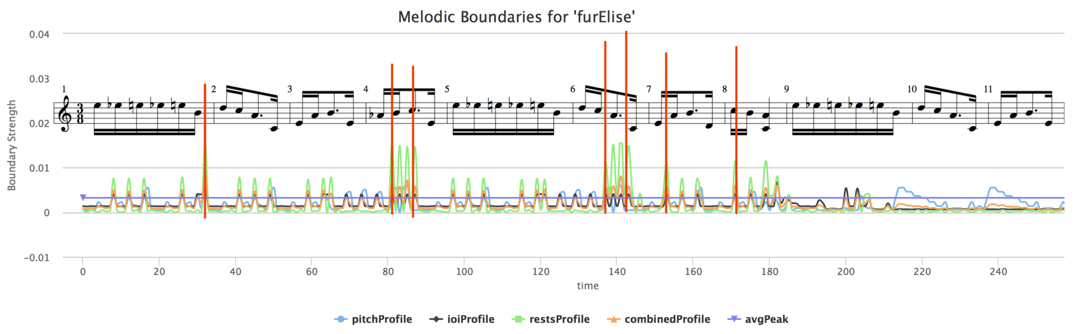


Figure 8.3: Local Boundaries Computed with Melodic Line

Figure 8.4 shows the results of the application of the second variant of the algorithm to the same melody. This variant makes use of the merging algorithm described in Section 6.3.1 to convert multi-voice phrases to single-voice phrases before running the LBDM analysis on on them.

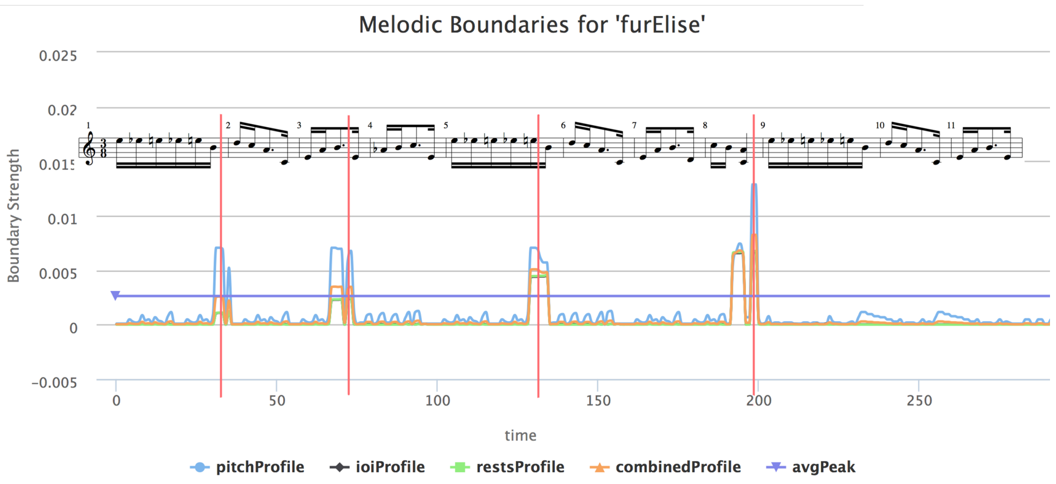


Figure 8.4: Local Boundaries Computed with Merged Phrase

As we can see, much better results have been achieved, with only a few of the peaks are above the threshold and they indicate splitting points that look reasonable to the human eye.

After taking a closer look at the data, we realised that this difference in performance between the two variants could have been caused by the presence of a fair amount of chords[1]. If there are many chords in a melody, especially chords with many notes, the corresponding symbolic representation will be a multi-voice phrase with many sub-phrases, which makes it more likely for the longest sub-phrase not to contain enough information for an accurate LBDM analysis.

To further explore this theory, we have manually composed, by using a `MIDI` composition software, a melody that contains repeated chords of three notes. While the second variant of the algorithm keeps producing interesting results, as shown in Figure 8.5, the first variant completely failed at identifying any phrase boundaries. This is because the information contained in the longest sub-phrase was insufficient for an accurate LBDM analysis. Figure 8.6 shows the profiles obtained by running the first variant of the algorithm.

[1]Only one chord is visible in the figure, however there are more in the rest of the melody. The LBDM analysis has been run on the whole of `Für Elise`'s right hand part, but only part of it would fit in the margins of this page

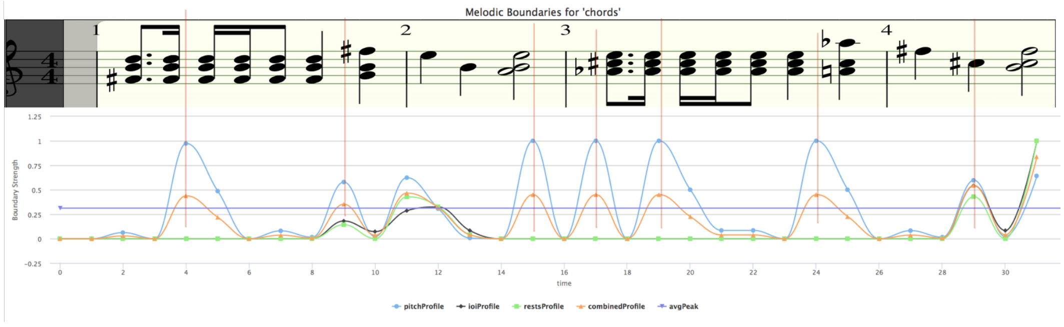


Figure 8.5: Local Boundaries Computed with Melodic Line

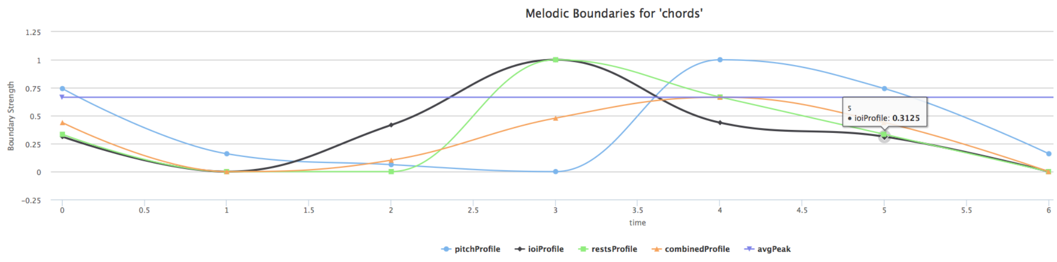


Figure 8.6: Local Boundaries Computed with Melodic Line

8.3 Phrase Variance

In this section we try to analyse how the phrase distance profiles of the improvised melodies compare to the average profile in the knowledge base.

The distance between two phrases is defined as the `Euclidean Distance` between their representations (as described in Section 5.4.3, a phrase can be represented as a point in a high-dimensional space). A distance profile is a sequence of values for the distance metric between pairs of adjacent phrases in a melody.

This metric was inspired by Jones' [12] remarks about algorithmically generated music being criticised for being too repetitive. The phrase distance profile of a melody should be a good indicator of whether it is repetitive or not.

Figure 8.7 shows a comparison between the average phrase distance profile of all the contents of the knowledge base, and those of three melodies generated by a single musician. Figure 8.8 instead compares the average profile with those of three melodies composed by three different musicians listening to each other.

While nothing can be concluded about the quality of the generated melodies, we can can definitely observe that the distance between adjacent phrases tends to oscillate around the average of the knowledge base. This might suggest that the generation process produces melodies that have similar properties to the phrases that are stored in the knowledge base, as one might expect.

We can also notice that while the distance between two phrases in the average profile has a very low variance, the opposite is true for the improvised melodies. This might be explained by the fact that there is a random component in the musical generation process which could sometimes select more similar phrases while other times more different phrases.

Finally, we can notice how in Figure 8.8, as pointed out by the red arrows, there seems to be higher correlation between the variations of the profiles of the three melodies, suggesting that the agents are actually listening to each other.

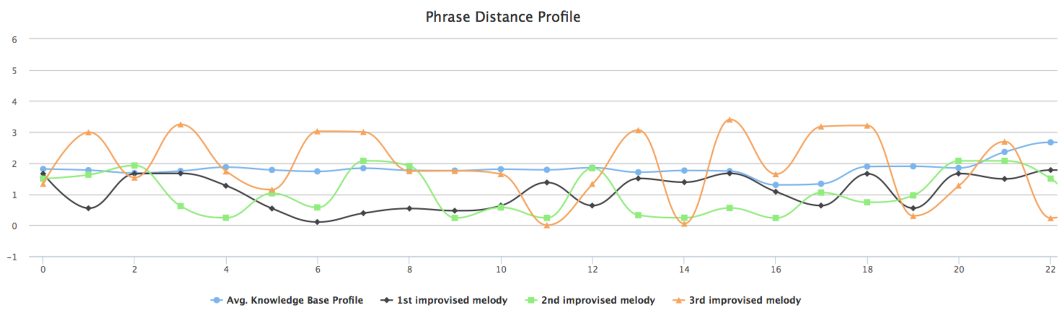


Figure 8.7: Phrase Distance Profiles for Melodies Generated by One Musician

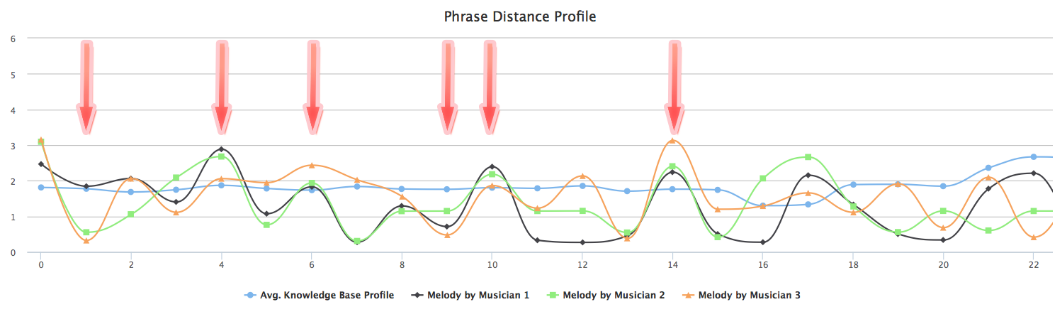


Figure 8.8: Phrase Distance Profiles for Melodies Generated by Three Musicians

Chapter 9

Conclusion

In conclusion we have developed a distributed system in which a set of independent musicians "listen" to each other through the exchange of messages and improvise music.

The system relies on a knowledge base of musical phrases extracted from a corpus of melodies composed by human musicians and stored as `MIDI` files. This knowledge base is used by each musician, independently, to generate, depending on the current musical context[1] for each musician, an initial population of phrases; this solves of the core problems with Genetic Algorithms: making sure that the initial population contains good candidate solutions. Once the initial population has been generated, it is evolved by genetic cross-over and the fittest individual is eventually selected. The fitness function used in our implementation is an Artificial Neural Network trained to approximate the musical tastes of the author and his friends.

`Improv` is a system that at performance time, unlike many other system, is completely independent from any human input. However, it strongly relies on human insight and creativity to work well: without a knowledge base of good musical pieces composed by human musicians, the system would not work. In fact, one could say that while some other systems rely on input from humans during or immediately before a performance, `Improv` strongly relied on it during the training process.

In conclusion we can say that, composing music is still today a task that is best performed by humans, but that there is room for tighter cooperation between them and AI systems.

[1]The musical context for a musician is the knowledge it has about what other musicians have playing in the current time window

Chapter 10

Future Work

One of the few certainties in software development[1] is that time is never enough. Because of this, there are a few features that we would have liked to add to the system but we could not.

10.1 Genre Negotiation

As mentioned in Section 4.2, musicians in `Improv` support voting to select a musical genre to improvise. However there currently are two limitations: there is no data about the musical genre in the `MIDI` files used for training, therefore the genre selected in the voting process is actually disregarded. Therefore, in the future, one might want to tag the files in the training set with information about their genre so that the improvisation can actually reflect the selected genre.

On top of that, currently all musicians share the same knowledge base. It would be interesting to experiment with creating different knowledge bases for each musician to better model the different musical backgrounds human musicians have.

This experiment, as mentioned Section 4.2, would require a change in the way genre negotiation is performed: instead of simply selecting the genre voted by the majority of musicians in the system we would have to account for the capabilities of each musician. This means that there are two parameters to be accounted for in the selection of the genre: the number of votes a genre has received and the number of musicians that are able to improvise that genre.

10.2 Retaining Good Pieces of Music

Currently there is no support for collecting human feedback on the improvised pieces of music. Therefore, it is not safe to add them to the knowledge base without the risk of polluting it with musical phrases that do not sound good.

Some systems require interactive user feedback during a musical performance, but we believe that, unless the feedback is passively collected through some sort of automatic sentiment analysis on the audience[2], it would spoil the performance.

[1] And maybe in life too

[2] Although the accuracy of such techniques is arguable

An alternative, perhaps better, approach is that of saving to disk all the pairs `(phrase, successor)` improvised by the system and allowing a human, through some GUI to play them back and decide whether to add them to the knowledge base or not. This would be a very slow process of growing the knowledge base, but it is important that no low-quality musical phrases are added to the knowledge base, as that would violate the assumption that all the pairs in the knowledge base are examples of good musical responses.

If this latter approach is deemed too slow, one could simply keep ignoring this last step of `CBR`, the retention of new pairs `(problem, solution)`, and simply grow the knowledge base by adding musical phrases from `MIDI` files containing new melodies composed by humans.

10.3 Performing over a Network

While the system is designed so that each musician can run on different machines on the same network[3], this has not been tried yet. If all the machines are in the same room, as in the PLOrk [2], making this work would only require a way of advertising the address of the message queue to all nodes in the system. This could be done by manually providing the IP address and port of the message queue to each musician. Alternatively, if on a Local Area Network, a protocol similar to `ARP` could be implemented to discover the location of the message queue. If musicians are communicating over the Internet, they could contact a server running some centralised service like `Apache Zookeeper` which would provide configuration information and IP addresses for message queues of different distributed orchestras.

Finally, if the laptops are not located in the same room, the system would need to be modified in so that while each machine would still run just musician for the purpose of composing new musical phrases, it would also play locally phrases generated by other musicians, so that each user would be able to hear the whole improvisation and not just what its musician is playing. Since there is a certain amount of latency of a network, if phrases composed by other musicians were played as they received by each node in the network, we could run into synchronisation issues due to messages delayed by the network's latency. To prevent this we could simply keep messages with musical information until the next `SyncMessage` is received by the director. When that happens we know that the director has received the broadcast messages from all the active musicians. Even though we still have no guarantee that the current musician has actually received those messages, we can play them locally at this point: we could, in theory, extend the `SyncMessage` so that it contains information about how many musicians the director believes are active and wait until we receive messages from them, however, that could potentially delay the play back indefinitely. This could be mitigated by equipping each musician with a health monitor (as described in Section 3.4) and only wait for messages from musicians it believes are still active.

[3] In fact the `AKKA` framework even when run locally, creates a message queue on a local port which is used by actors to exchange messages

Bibliography

[1] “Mozart Dice Game - `http://www.amaranthpublishing.com/MozartDiceGame.htm`.”

[2] D. Trueman, P. Cook, S. Smallwood, and G. Wang, “PLOrk: The Princeton Laptop Orchestra, Year 1,” tech. rep., July 2006.

[3] J. E. Smith and A. E. Eiben, *Introduction to Evolutionary Computing* . Springer, Oct. 2008.

[4] M. J. Yee-King, “An autonomous timbre matching improviser,” 2011.

[5] J. A. Biles, “GenJam: A Genetic Algorithm for Generating Jazz Solos,” *ICMC Proceedings*, pp. 131–137, 1994.

[6] M. Hassoun, *Fundamentals of Artificial Neural Networks.* A Bradford Book, Jan. 2003.

[7] P. M. TODD, “A Connectionist Approach to Algorithmic Composition,” *Computer Music Journal*, vol. 13, no. 4, pp. 27–43, 1989.

[8] P. Toiviainen, “Modeling the Target-Note Technique of Bebop-Style Jazz Improvisation: An Artificial Neural Network Approach,” *Music Perception: An Interdisciplinary Journal*, vol. 12, pp. 399–413, July 1995.

[9] P. Toiviainen, “Symbolic AI versus Connectionism in Music Research.,” *Readings in Music and Artificial Intelligence*, 2000.

[10] J. R. Norris, “Markov chains.,” *Cambridge University Press 1998*, 1998.

[11] N. Privault, *Understanding Markov Chains: Examples and Applications* . Springer, July 2013.

[12] K. JONES, “Compositional Applications of Stochastic-Processes,” *Computer Music Journal*, vol. 5, no. 2, pp. 45–61, 1981.

[13] P. Langston, “Six techniques for algorithmic music composition,” *15th International Computer Music Conference*, 1989.

[14] Y. VISELL, “Spontaneous organisation, pattern models, and music,” *Organised Sound*, vol. 9, pp. 151–165, Aug. 2004.

[15] B. Manaris, D. Hughes, and Y. Vassilandonakis, “Monterey mirror: Combining Markov models, genetic algorithms, and power laws,” in *Proceedings of 1st . . .* , 2011.

[16] J. Gillick, K. Tang, and R. M. Keller, “Learning jazz grammars,” in *Proceedings of the Sound . . .* , 2009.

[17] R. J. Elliott, L. Aggoun, and J. B. Moore, *Hidden Markov Models.* Estimation and Control, Springer Science & Business Media, Sept. 2008.

[18] K. M. Biyikoglu, “A Markov model for chorale harmonization,” *Preceedings of the 5 th Triennial ESCOM . . .* , 2003.

[19] J. A. Moorer, “Music and computer composition,” *Communications of the ACM*, vol. 15, Feb. 1972.

[20] P. Ribeiro, F. C. Pereira, M. Ferrand, and A. Cardoso, “Case-based Melody generation with MuzaCazUza,” in *Proceedings of the AISB' . . .* , 2001.

[21] T. Parikh, “IRIS: artificially Intelligent Real-time Improvisation System,” pp. 1–94, Oct. 2004.

[22] G. L. Ramalho, P.-Y. Rolland, and J.-G. Ganascia, “An Artificially Intelligent Jazz Performer,” *Journal of New Music Research*, vol. 28, pp. 105–129, Aug. 2010.

[23] “AKKA Documentation - `http://akka.io/docs/`.”

[24] E. Cambouropoulos, “The local boundary detection model (LBDM) and its application in the study of expressive timing,” in *Proceedings of the International Computer . . .* , 2001.

[25] R. Parncutt and G. McPherson, *The Science and Psychology of Music Performance: Creative Strategies for Teaching and Learning.* Creative Strategies for Teaching and Learning, Oxford University Press, Mar. 2002.

[26] A. W. Moore, *Moore, Andrew W.* An Introductory Tutorial on kd-trees,(Extract from Andrew Moore‘s PhD Thesis: Efficient Memory-based Learning for Robot Control). PhD Thesis.

[27] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, “An optimal algorithm for approximate nearest neighbor searching fixed dimensions,” *Journal of the ACM*, vol. 45, pp. 891–923, Nov. 1998.

[28] S. M. Omohundro, “Omohundro: Efficient algorithms with neural network behavior - Google Scholar,” 1987.

[29] R. A. Brown, “Building a Balanced k-d Tree in O(kn log n) Time.,” *CoRR abs/1006.4948*, vol. 1410, p. 5420, Oct. 2014.

[30] C. McKay and I. Fujinaga, “jSymbolic: A feature extractor for MIDI files,” in *Proceedings of the International Computer . . .* , 2006.

[31] C. McKay, “Automatic Genre Classification of MIDI Recordings,” 2004.