Universiteit Leiden
The Netherlands

Computer Science

Real-time Melody Harmonization in an Improvisational Context

Jeroen Donkers

Supervisors:
Edwin van der Heide
Maarten Lamers

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

August 25, 2020

Abstract

This bachelor thesis explores the possibilities in creating a creative interaction between a computer system and a musician in the context of real-time musical improvisation. Existing models to analyse and categorize creativity and interactivity are presented and the models are used to reason what kind of system would be needed to create a satisfying interaction. After that it is explored how a genetic algorithm would be useful in making the interaction possible. It is reasoned that the crossover, mutation and fitness functions of a genetic algorithm in particular have the potential to give insight in to what extent the algorithm is creative and that various degrees of interaction can be achieved by altering these functions.

To see how a genetic algorithm would perform in this context, a system has been made that simulates the environment of a real-time jam session, where a human musician will be able to participate by playing an improvised melody in a predefined key, and where an agent using a genetic algorithm participates by harmonizing this melody using chords. This system is then analysed according to the models of creativity and interactivity.

The analysis of the system forms the base of the further discussion where some of the features that need further research are highlighted, like the need for a different process that would be able to continuously modify the fitness function in order to make transformational creativity (a form of creativity where the process of creating ideas is altered) possible, and the fact that there is a limit in to how quickly the agent can react interactively to user input.

However, its strengths are also discussed, like the potential to modify its behaviour based on real-time events in the jam session so it can interactively generate different creative musical ideas. The design of the system makes it easy for another process to alter its decision making, and to control the program output for specific features of an arrangement. Even with a complex fitness function, the genetic algorithm is able to perform well in real-time due to some configurable thresholds and the fact that the agent evaluates the state of the jam session only once every measure.

Contents

1 Introduction

Modern society is incorporating computer systems in our lives now more than ever to make things more convenient or interesting, which makes interacting with them an important aspect of designing these systems. The field of human-computer interaction knows increasingly more about how the dialogue between a human and a computer could be made more fluent. Still, there are some areas that are less explored compared to others. One of these areas describes the ways of interaction with computer systems that are able to generate music. In that area, it would be interesting to explore how a dialogue between a human and a computer system could be realised using mostly the language of music in such a way that the human and the system could create interactively music together. The starting point of this bachelor thesis is to find out whether some advancements can be made in creating an interactive musical dialogue between a human and a computer system. This can be used to satisfy an interest in exploring musical interaction explicitly, and it could potentially lead to interesting insights in human-computer interaction in general.

Music is a highly complex, abstract, diverse art form. Even though music depends heavily on cultural and subjective decision making, the field of Computer Science manages to play a role in creating music by offering the precision and speed of computers.

In its most primitive form, computers can be used as an instrument to create music by digitally generating predefined tones as sound waves. Computers also provide us audio processing tools, such as a Digital Audio Workstation, where you can record, edit, and play audio signals. However, computers have the potential to be more than just an instrument, and they may be able to understand music to a certain extent as well.
Just like any other form of art, it is a difficult if not an impossible task to formally and fully understand music. But that does not mean that no (formal) models exist that help us partly understand music. As music is so complex, there are multiple perspectives one can consider to build a model for. These models each help with solving different goals.

For example, some researchers are trying to use computers to analyse music by processing its audio signals, which can be studied for downbeat tracking, tempo estimation, and chord recognition. One other case is the separation of different audio sources (voices or instruments) from a single audio signal, which has been researched by people like J. Driedger [Dri16] and Y. Luo [LM17].

Others like E. Guaus [Gua09] are trying to categorize existing music into moods, styles and genres. This has applications in for instance the automatic labeling of music (for example, for radio stations), music recommendation and personal music organization.

Another example is (partially) generating music using AI. Research of N. Otani [OTKN12] explores the idea of generating chord progressions based on the listener's emotional impressions of music. More recent research from the company OpenAI tries to generate raw audio music with singing using their model called 'Jukebox' [DJP+20]. They have trained residual networks with a dataset of 1.2 million songs (raw audio with additional metadata like lyrics, artist, genre, etc) and are able to condition the generated raw audio music on specific artists, genres and lyrics.

One particular aspect of research in music is the temporal aspect, and whether it is in a real-time context or not. Often there are different approaches for doing something in real-time instead of working with something that is predefined. For example, regarding the processing of audio signals mentioned before, J. Driedger [Dri16] tried to reduce the complexity to process an audio signal by splitting it into a set of two or more mid-level components so these components can be analysed further. A generic iterative solution like this would not be possible if the analysis was required in real-time. In contrast, Luo [LM17] tried to separate a real-time audio signal with deep learning, but is a rather specific solution, as it focuses on speech separation only.

It is worth noting that these attempts to automate aspects of music act only as a tool to be used by musicians, and are not meant to eventually replace the musician himself. As musicians will work with computer systems to aid them, it is important to realize that they would benefit most from an interaction with these systems in a way that sparks creativity.

One suitable interaction would be a hands-on experience where a musician would be able to interact with the computer system to improvise music together in real-time.

1.1 Improvisation

One example of real-time interactions between musicians is a jam session – an activity where musicians improvise a musical piece with a minimal predefined set of rules about how the piece should come together. For example, the musicians can agree to start on a specific key, a specific rhythm, or even some existing musical theme [1]. These rules tend more to say something about the general structure of the piece and are there to give the musicians a collective understanding of the direction of the piece without restricting them to improvise and form new ideas inside this structure.

As M. Grachten mentions in his first pages on *Jazz improvisation as a computational process* [Gra01], the notion of improvisation is generally considered to be 'inventing music while playing it', or to put it more casually, 'on-the-fly composing'. At least in jazz improvisation, this is generally thought of as playing around with a certain musical theme (a melodic line) accompanied by some chord progression. Here, 'playing around' means coming up with new and interesting musical ideas, often in some way related to the original melodic line and chord progression. This is widely considered to be a creative process. Generating creative ideas to improvise will be discussed in 1.1.1.

In the context of an improvisational jam session, not every one will go off in crazy virtuosic tangents, developing interesting musical ideas on their own as much as they can. A musical piece performed like that would be widely considered as incoherent. Musicians listen to each other and try to support and emphasize each other's ideas. As a consequence, there are some musicians that take the responsibility of the direction of a certain musical idea, while the rest would take on a more supportive role.

Multiple examples of this can be seen in a jam session from Cory Henry (among others) [Hen20]. The musicians participating in the jam session have agreed to play around with the theme of *Creepin'* by *Stevie Wonder*. At the start of the musical piece, the musicians stay very close to the original song. Slowly, the musicians start to introduce elements – melodic lines, or chords –

[1] A musical theme is meant here in the broadest sense of the word. It can mean a short melody, or a whole chord structure. Jazz musicians often improvise over *jazz standards*, popular songs in jazz that contain very distinct melodious patterns or chord progressions.

that don't originate from *Creepin'* itself: improvisation is happening. A dialogue starts to form between the musicians, a musical conversation, where Cory Henry, the keyboardist, is playing the melody and harmony and seems in control of the direction of the conversation – the bass player and drummer follow the chords and melody closely and try to accompany it as best as they can. Around 3:32, you can hear the bass player say "Okay, I'll take it from here". The keyboardist takes on a more passive role by switching to playing only chords. The bassist receives almost all the attention by playing a melody (a solo). And the rhythmic and harmonious patterns the bassist suggests are being heard by the drummer and keyboardist which play around these ideas. A more subtle switch in control is seen during the ending of the solo. There are no verbal queues, but the musicians recognize that the solo is coming to an end nevertheless.

The handing/taking over of control does not always require that all musicians recognize it immediately. An example can be seen at the video of Cory Henry at 14:15, where the drummer is in the middle of playing a solo. The keyboardist was under the impression that the solo ended, and started to play something more dominant – but as soon as he heard the drummer continuing his solo, he stopped to make sure the control was again totally in the hands of the drummer.

With the above examples it is clear that a jam session could require different forms of interactivity [2], so it will ultimately be beneficial if an improvising computer system would incorporate this in some way. This notion of interactivity will be elaborated on in 1.1.2.

There are a number of studies that look into musical improvisation, but which are limited in the ways they interact with other participants. Again the example of GenJam [Bil94] can be used. It generates jazz solos over a particular arrangement. The chord progression and song structure needs to be predefined, and a human listener can only react positively or negatively to a certain melody, and influences GenJam to play melodies that would most likely initiate a positive reaction.

In the context of an improvisational jam session, GenJam would act as a solution for recreating the behaviour of a soloist. However, it would be interesting to explore the idea of recreating the behaviour of a musician that is not necessarily so dominant in determining the structure of the dialogue of the song. Instead, it might be interesting to see how a computer system might behave differently depending on the state of the song, so that multiple levels of interaction could be embedded in the dialogue.

1.1.1 Creativity

When analysing (musical) improvisation, one must also look into *creativity*. A jam session, or performing music in general, is considered a creative process. If there was some notion of what creativity actually entails, it could be used by a system that mimics human improvisation to generate 'creative' ideas. In the context of creativity in its general sense, M. Boden [Bod09] has taken an influential approach to describing it:

> "Creativity is the ability to come up with ideas or artefacts that are new, surprising, and valuable. "Ideas," here, includes concepts, poems, musical compositions, scientific

[2]In the video, it could also be seen that the musicians occasionally give each other verbal queues to let the other musicians know that they like what is being played. This aspect will be briefly discussed in 5.1.

> theories, cooking recipes, choreography, jokes ... and so on, and on. ”Artefacts” include paintings, sculpture, steam-engines, vacuum cleaners, pottery, origami, penny-whistles ... and you can name many more.” [Bod09] – p.1

It is not totally clear whether Boden is making observations about creativity, or actually defining it herself. Nevertheless, Boden’s notion of creativity is providing a structure in which a potential ’creative’ system could be evaluated.

Describing what is *new, surprising, valuable* sounds vague, but there are definitely some things that can be said about it.

Boden divides the meaning of ’new’ into two groups: P-creativity (Psychological creativity) and H-creativity (Historical creativity).

> ”P-creativity involves coming up with a surprising, valuable idea that’s new to the person who comes up with it. It doesn’t matter how many people have had that idea before. But if a new idea is H-creative, that means that (so far as we know) no-one else has had it before: it has arisen for the first time in human history.” [Bod09] – p.2

To further define creativity, Boden distinguishes between three different ways on how a *surprising* idea or artifact that is labeled as creative can be contrived:

1. *Combinational creativity*
   Using existing ideas, it is possible to create new artifacts by making unfamiliar combinations of these ideas. The process of finding these combinations is considered to be creative, and an artifact that illustrates these unfamiliar combinations is often called a creative idea. For example, creative analogies, poetic imagery and collage in painting exhibit this form of creativity.

2. *Exploratory creativity*
   Other than only combining existing ideas, some structured style of thought is used to generate new ideas or artifacts. Within a certain style of thought, many ideas are possible. Ideas that are generated using the same style of thought live in the conceptual space of that style.

   > ”They’re normally picked up from one’s own culture or peer-group, but are occasionally borrowed from other cultures. In either case, they’re already there: they aren’t originated by one individual mind. They include ways of writing prose or poetry; styles of sculpture, painting, or music; theories in chemistry or biology; fashions of couture or choreography, nouvel cuisine and good old meat-and-two-veg ... in short, any disciplined way of thinking that’s familiar to (and valued by) a certain social group.” [Bod09] – p.3

   Exploratory creativity is the process of exploring the conceptual space. Even if the idea itself is not necessarily surprising, it is still valuable as it can enable previously unexpected areas of the conceptual space to become more apparent. This gives more insight at what potential the thinking style has.

3. *Transformational creativity*
   Now, using this structured style of thinking can render certain thoughts unthinkable. It is possible to realise the limitations of a thinking style. This brings us to the notion of

> transformational creativity. It involves a thought that, with respect to the conceptual space couldn't have been thought of before. It requires that the creator of the idea alters the pre-existing style in some way. The process of altering the style of thought (and in turn the conceptual space) is called transformational creativity.

The above three types of creativity try to describe the *novelty* of ideas – and in what way an idea can be surprising. Boden realised that in this model, there was a problem in describing what are *valuable* ideas:

> "Our aesthetic values are difficult to recognize, more difficult to put into words, and even more difficult to state really clearly. [...] Moreover, they change" [Bod09] – p.8

So, Boden has merely sketched the outline of some model of creativity. One can get a feel for how the model would work, but there are questions that arise when getting into specifics. And these lie not only in the value of ideas: What are these conceptual spaces, *exactly*? What does a transformation of such a search space look like?

Others have tried to extend upon Boden's model. G. Ritchie made an effort to better describe the conceptual space in order to get a better understanding of how transforming it would look like and what it would entail [Rit06]. What Ritchie noticed is that if transformation (other than reduction) where to be possible, the conceptual space must be embedded in some larger space of possibilities. He gives an example of creativity in chess:

> "Suppose the set of chess games consists of all valid move-sequences, then anything which transcended that definition would not be a game of chess - that all-encompassing set of games is the "logically possible" set. There are then two possible positions:
>
> 1. There is a set of "typical games", a proper subset of the logically possible games, which constitutes the conceptual space, and this space is available for transformation within the larger logically possible set.
>
> 2. The logically possible set of games is exactly the conceptual space, so no valid game of chess can display transformational creativity - the only way that a chess player can be transformationally creative is to invent new rules."
>
> [Rit06] – p.248

This way, the conceptual space would always be a subset of the logically possible set an artifact could be. Is there any way to further specify this set?

In one sense, the conceptual space is just some set of artifacts. But arguably it is often more interesting to know how the space is characterized – what are the rules that make certain artifacts an element of the conceptual space? As Ritchie points out, a set of artifacts may be infinite, but has a finite definition.

Next to sketching a model for creativity, Boden mentions in her work [Bod09] the notion of computer creativity and whether such a thing is possible. However, to her, answering this question seems less important than answering 'Could computers come up with ideas that at least *appear* to be creative?'.

With respect to Boden's description of creativity, several computer programs already exist that explore various conceptual spaces in acceptable ways. Boden uses the example of artifacts of AARON [Coh06], a drawing program. These artifacts appreciated in various galleries from unsuspecting visitors. Again, it is unclear here what is meant exactly by 'the conceptual space', but one could argue it makes sense to loosely define this space as 'the set of artifacts that AARON could create provided the current rule set of AARON'.

Using this example Boden claims that it seems to be that computers *can* be used to form ideas that acceptably appear to be creative, or at least, are able to generate artifacts that are 'surprising' using exploratory creativity.

As transformational creativity is not yet sufficiently defined, a possible rule-set for the conceptual space is adopted to suggest a way in which transformational creativity could be realised in a computer system in 1.2.

1.1.2 Interactivity

As already explained in 1.1, a jam session is not just about generating creative ideas constantly. The participating musicians actively influence each other by playing and listening. There is a dialogue in musical language much like people can have conversations in the spoken languages. In this improvisatory context, musicians participate in this dialogue by interacting with each other, and consequently influence the direction of the real-time composition.

The doctoral thesis of S. Bell called *Participatory art and computers* [Bel10] sets out to determine the essential characteristics of participatory works of art that use computer technology. Now, interactive artworks all have a distinct feature. With these artworks, most if not all knowledge which produces the output of the artwork is embedded in the artwork itself. For example, a piano is also an artifact which someone is able to interact with – but it is not considered as an interactive work of art, for the reason being that any interesting output of the artifact (music being played on the piano) is interesting because the participant has explicit knowledge to create something interesting with a piano. A computer system mimicking the behaviour of a musician is in that same sense not an interactive art work. However, as long as this difference is kept in mind, there are still helpful ideas in the thesis of Bell about how interactivity between two entities work in general.

The thesis proposes a system of analysis in which the principal characteristics are considered to be those which contribute to the degree and manner of control afforded to participants. This system of analysis is particularly useful as it is likely that it could be used to analyse the characteristic of directional control (the control of the improvisatory direction of the musical piece in a jam session) of a system.

Bell references the work of S. Cornock and E. Edmonds [CE73], which classifies systems in interactive situations as:

1. *static*: The system is unchanging – think of a traditional artwork like a painting.
2. *dynamic-passive*: The system is caused to change with time by the system itself or by the environment. However, participants have no control of the system and cannot alter anything – for example, kinetic art.

3. *dynamic-interactive*: Like a dynamic-passive system, the system can alter itself. But in the case of a dynamic-interactive system, participants do have some form of control over the artwork. This could be as simple as pressing a button to turn on some light in the artwork.
4. *varying dynamic-interactive*: A special case of a dynamic-interactive system where additionally the possibilities of interacting with the system are also altered.

It should be obvious that a computer system that plays music over time can not be a *static* system. It is impossible for an unchanging system to make different sounds over time, much like how a video cannot be a static system. Let's take the jam session from Cory Henry [Hen20] mentioned in 1.1 and look at it as if it was an interactive art system – in what category would a computer system mimicking it be?

From the video a couple of functions can be derived, from the standpoint of any musician that participated in the session:

1. Play while being dominant in deciding the direction of the piece.
2. Play while not being dominant in deciding the direction of the piece.
3. Give this directional control to other musician.
4. Take this directional control.

The first function is the simplest in terms of interactivity: for a minimal implementation of 1, the system just needs to play music in some predefined context, so that the other participants of the jam session would be able to follow along. It does not necessarily need any input from the participants.

Function 2 requires listening to the musician that is in control of the direction to the piece. It relies on the input of a participant – therefore, the system can never be *static* or even be *dynamic-passive*. At most, the function would require a *dynamic-interactive* system.

Additionally, the functions 3 and 4 combine the first two functions by being able to switch which of the two functions 2 and 1 to execute. Since the interaction of playing with the system while it's in control differs with playing with the system while a participant is control, a system that is *varying dynamic-interactive* would be needed [3].

1.2 Approaches

There are various algorithms that would lead to some interesting output with the goal of creativity and interactivity in mind, but which differ in the way they explore a conceptual space. As discussed in 1.1.1, it would not make sense to consider the conceptual space to be identical to the algorithm's search space, as transforming this conceptual space would be impossible. To put the different approaches in context to transformational creativity, the conceptual space is considered to be the possible output an approach can generate using its **current** rule-set. Here, what is meant by rule-set is not necessarily some explicitly coded set of rules in the algorithm, but rather rules that

[3]While the conceptual possibilities of implementing a varying dynamic-interactive will be discussed later, there will be no concrete example for it and it will not be experimented with.

could be derived from the output the algorithm produces. Below are some of these approaches to handling real-time improvisation and their different features.

1.2.1 Rule-based

Musicians that play in a jam session will know about and use some music theory to at least some extent. For example, even without knowing about scales or chords, a musician has at least an abstract idea of what sounds diatonic. In that regard, one might be inclined to map a model of music theory and to specify rules and constraints to ensure the output is closely described by music theory. Stochastic rules describe how to generate the next output depending on the state of the program. In a real-time context, you could create an algorithm using a temporal approach described by JIG [Gra01]. Here, every next output depends on the previous output, and is determined by the constraints of the musical model Grachten chose to implement.

Evidently, there should be a clear understanding of what would be considered desirable output before the output is being generated. In terms of exploring the conceptual search space (1.1.1), it would be relatively harder to find novel ideas that have not been found before, as most likely the creator of the algorithm is already familiar with the rules that generate the output, and therefore the possibilities of the output.

For a rule-based system to be *varying dynamic-interactive*, it is required to define some meta-rules that explicitly alter the rule-set the system uses to generate output. That would most likely mean to explicitly define the 4 functions any participating musician should be able to perform in 1.1.2.

1.2.2 Neural Network

In contrast, neural networks don't exhibit the requirement of having an explicit understanding on how to generate acceptable output. Instead, it solely relies on some large set of examples of desirable output. As an example, the program RoboMozart [Wee16] uses a LSTM neural network to generate music using MIDI input as a starting point. The creator of RoboMozart, J. Wheel, trained the neural network using 74 full songs of one artist in MIDI format. In the discussion of the project he mentioned that the network struggled to learn how to transition between musical structures (especially in longer output sequences), and that the model could greatly benefit from significantly increasing the size of the training data set.

One could argue that using a neural network seems like a very natural solution to this problem – after all, an analogy can be made with a neural network and a musician in the sense that it is often said they both need 'training' before it can perform well. Generally, a musician learns from experience – he has listened to a lot of music to know what musical ideas would be interesting for him to explore.

The more examples the neural network is trained with, the more the properties of the conceptual space of the algorithm are similar to the properties of the examples. A neural network performs well in creating new output that would be plausible to exist in a conceptual space where at least the training data is also a subset of.

That being said, when a neural network is trained so that the conceptual space is closely encoded in the training data, it might not be that exceptional in transformational creativity in the sense that it might be hard to realise the transformation of this conceptual space. There is no explicit definition of the rules for desired properties of the output, as they are encoded in the training data. To alter it, you might need to train the neural network with an additional set of different examples. Getting control over the change of conceptual space then is not so straight-forward.

1.2.3 Genetic Algorithm

Something can be said about using a genetic algorithm to generate harmony. After all, it is a common search heuristic and should be able to deliver some results considering exploring the conceptual space. In contrast to a rule-based system, it is not needed to specify constraints to how the output should be generated so it would ultimately lead to some desirable solution. Crossover and mutation techniques should lead to some interesting outcomes and the undesirable solutions would be filtered away by a fitness function. The important difference here is that the constraints would be specified for a solution rather than the generation of a solution. This more closely resembles the rule-set of the actual conceptual space. The fitness function controls the properties of the eventual generated output more directly than a rule-based system.

More directly, the crossover function could be able to control the combinational creativity of the program (as it quite literally) combines and mixes existing ideas, whereas the mutation function influences the amount of exploratory creativity – within a certain style of thought, new ideas are generated.

The processes of mutation and crossover also support the ability to develop thematic musical ideas, where one specific idea is generated, and then possibly altered or expanded upon later (by using the same individual multiple times, occasionally altering it by mutation and crossover) [4].

Among others, the following are some typical questions that need to be answered in order to implement a genetic algorithm:

1. What will be an individual?
2. What will the initial population consist of?
3. What will be the fitness function?

The answers to these questions might not be immediately clear for generating music in real time. The form of an individual would have to support that it could be used in real time, so it would at least need to provide smaller parts of the eventual generated music.
Typically, the initial population is filled with random instantiations of individuals. Depending on the form of the individual, these little parts of music could have a very large search space, also containing a lot of instances that generally speaking, would not sound very musical at all. For example, one could consider generating short melody phrases using a genetic algorithm. Use the 88 notes from a typical piano to build the melody phrases: Even without taking rhythm into account, for a melody phrase containing n notes, the search space would still have a size of 88^n. One can imagine that there would be far more phrases which would sound very random instead of musical. That being said, it could take a lot of iterations to evolve a randomly instantiated initial population

[4]The claim of a genetic algorithm supporting the ability of thematic development will be validated in the experiments (chapter 4).

into actual musical sounding parts. Considering the real-time aspect, it could be undesirable to calculate that much iterations as it takes some time to perform these calculations.
Some would also argue that a disadvantage of implementing a genetic algorithm here is that it is considered that a fitness function for complex problems is generally hard to define.

At its essence, a fitness function about generating music would be predominantly subjective, as people have different values when it comes down to music. With respect to the conceptual space talked about in 1.1.1, it is not a problem to make the transformations, but as Boden says,

> ”What’s difficult is to state our aesthetic values clearly enough to enable the program itself to make the evaluation at each generation.” [Bod09] – p.8

The program GenJam made by J. Biles generates melody phrases in a chord progression with an interactive evolutionary algorithm: it relies on a musician to evaluate mutated phrases. It avoids finding a suitable algorithm for providing fitness values typically found in a Genetic Algorithm, believing a (human) mentor is needed to dictate what is an interesting melody. The paper focuses on the interaction between musician and system needed for generating phrases:

> [...] GenJam’s design reflects the need to minimize the amount of listening required and to make the mentor’s interface as simple as possible. [Bil94] – p.135

There are however, commonly shared values that the majority of musicians agree on, so it is arguable that there could exist a fitness function that would be satisfactory to them. It is also worth noting that once the idea of using a genetic algorithm in a certain way has been established, it is relatively easy to modify the fitness function later for various specific use cases.

Even more interesting is the fact that the fitness function can be modified very directly. Consequently, as it resembles the rule-set for the actual algorithm’s conceptual space, the conceptual space can be modified very directly. That makes it possible to have a lot of control in the transformational creativeness of the program. The fitness function can also be altered for the sake of interactivity. For example, if two separate fitness functions are used for both playing dominantly (influencing the direction of the dialogue a lot) and playing supportive (not necessarily influencing the direction as much), the system could switch between these to create a sense of *varying dynamic-interactivity*.

However, a problem with genetic algorithms is that they can get computationally expensive relatively fast, depending on the fitness function complexity in combination with the population size and the amount of iterations. Especially in a real-time context this could lead to some trouble.

In conclusion, with regards to creativity and interaction, a genetic algorithm could be a very interesting algorithm to explore real-time improvisation with.

2 Description

The idea of a computer system participating in a jam session will be explored by looking at the point of view of a musician improvising in a jam session. For the sake of simplicity, it is assumed that there will be only two participants in a jam session: the musician, and the AI – the computer program. Also, it is assumed that the musician would be responsible for improvising a melody, and that the goal of the AI is to generate a corresponding harmony for it. What does a musician playing a melody like that need in order for him to be able to participate in a jam session? Only after answering that, it is possible to look at what a similar computer system would look like.

2.1 Musician

Based on the observations of a jam session previously discussed in 1.1, rhythm keeps the musicians playing together in sync. Although rhythm is mostly highlighted by the drummer, all musicians are playing a role in defining the rhythm of the piece. This rhythm enables the musicians to interpret the music being played. It is very possible that in the jam session of Cory Henry, they have agreed on a tempo and time signature before playing, but it is also possible that someone started the jam session by playing a specific rhythm and the rest was able to hear which tempo and which time signature was played, as most musicians are generally pretty good in interpreting rhythm. It is possible in a musical piece that the tempo and time signature changes at some point – whether that is predefined or interpreted by the participants. However for simplicity here it will be considered a requirement of a jam session that the rhythm of the piece will be predefined.

Similarly, the musicians from the example jam session are playing in a specific key [5], to understand the implied harmony and melody. Much like rhythm, the key for any point in the musical piece could be either predefined or interpreted while playing. Again for simplicity, it will be considered a requirement of a jam session that the key of the piece will be predefined.

An obvious but important aspect of a jam session is that all participants should have an instrument that is able to communicate the music being played on it to the other participants. The instrument isn't limited to be a piano, or bass guitar – it can be someone's voice, or even a button on a computer keyboard: as long as the instrument can emit output that is interpretable as music by the other participants.

To summarize, the minimal environment of a jam session consists of the following rules:

1. There is a predefined rhythm that states the tempo and time signature of the improvised piece.
2. There is a predefined key in which the participants will play in.
3. All participants must be able to play music on an instrument that enables the other participants to interpret the instrument output as music.

2.2 System

As proposed in the introduction, the idea is to explore the possibility of a computer program which is able to participate in a jam session. It would be beyond the scope to assume that the computer system would be able to interact with conventional analogue instruments. It is not necessary to talk about analysing sound waves, or making a robot which can play the piano, for example.

To separate these concepts, some system is needed to simulate the environment of a jam session in a way that enables a 'simple' computer program to participate in the jam session. This system needs to have the same requirements as a jam session discussed in the previous section. Each

[5] In the case of the jam session of Cory Henry, it's $F^\sharp$ major – keys will be explained in 2.2.3.

individual requirement will be elaborated on in the following sections, with respect to the additional requirement that a computer system would have to be able to participate.

2.2.1 Rhythm

Since in this case we are dealing with a computer system, which is a discrete system, there has to be some form of note quantization to make the time resolution of notes in the rhythm finite.

Sometimes musicians are able to feel and detect this rhythm internally (much like the example from Cory Henry[Hen20]), but an external source can also be used. Think of a metronome or a conductor. When looking at a typical metronome, its functions are Tempo and Time signature.

Rhythm can now be made up of the following components:

1. **Tempo** $(\frac{\text{beats}}{\text{minute}})$
2. **Time signature** $(\frac{\text{beats}}{\text{measure}})$
3. **Note quantization** $(\frac{\text{notes}}{\text{beat}})$

With these values of rhythm defined, the following song structure can be made:

A song is made up of a growing array of beats. When the session is playing the song, it will go through this array of beats sequentially in such a way that there is a constant number of beats played per minute and where each beat takes an equal amount of time. The length of any beat in seconds can therefore be defined as $\frac{60}{\text{Tempo}}$.

Any sequential group of b beats can be called a measure. In the song, all beats must belong to a specific measure. Here, b is defined as the time signature.

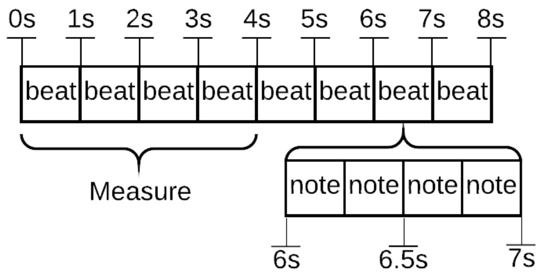


Figure 1: The first two measures of a song structure with Tempo = 60, Time signature = 4, Note quantization = 4. There are 4 beats in a measure, every beat is $\frac{60}{60} = 1$ second long, and contains 4 notes that each take $\frac{60}{4 * 60} = .25$ seconds.

This structure (an example can be seen in Figure 1) enables saving notes on a discrete location in the rhythm, relative to the beats and measures. To ensure that the other musician knows about this rhythmic context, a metronome tick sound can be played whenever a new beat is being played. Whenever a first beat of a measure is played, a different sound can be played to indicate a new measure.

2.2.2 Melody

In preparation for rule 3 talked about in 2.1, a structure for a melody will be implemented in the song structure. Later on in 2.2.4 it is discussed how the musician is able to input the melody in the structure.

Every beat consists of an array of n notes. These notes can also be undefined to indicate that no note should be played. When the session is playing a beat, it will go through all n notes sequentially in such a way that each note takes an equal amount of time. The length of a note in seconds is $\frac{60}{n \cdot \text{Tempo}}$. Here, the number of possible notes n is the note quantization value.

Using only this technique for structuring notes is insufficient if the length of the notes being played should also be preserved. Even though a structure that contains the length of notes can be realised [6], this structure will already suffice for implementing that the system would be able to easily understand in which part of the improvised piece the notes are being played. That does not mean that the length of a played note is not important in interpreting the melody, but it is not absolutely necessary. This way, a note in a melody happens only at one specific point in the song structure, and there can be only one note at a time, which is convenient for analysing the melody.

2.2.3 Key Signatures

In order for the system to contextualize the melody from a musical standpoint, it is needed to determine the key signature of the melody. A key signature can be viewed as an ordered subset of multiple musical notes $\{C, C^\sharp, D, D^\sharp, E, F, F^\sharp, G, G^\sharp, A, A^\sharp, B\}$. For example, the D major scale is defined as the subset $\{D, E, F^\sharp, G, A, B, C^\sharp\}$.

Another way to look at the key signature can be seen as a list of the intervals between its notes (the scale) and a starting note (the root). For example, the chromatic scale contains all notes, so all intervals of the scale are 1. The intervals of the whole-tone scale are all 2. Scales with different intervals are for example the ionian (major) scale, with intervals $\{2, 2, 1, 2, 2, 2(, 1)\}$. From the above it can be seen that a C-ionian scale would be built as follows:

$$C \xrightarrow{2} D \xrightarrow{2} E \xrightarrow{1} F \xrightarrow{2} G \xrightarrow{2} A \xrightarrow{2} B (\xrightarrow{1} C)$$

The D-ionian scale is built like this:

$$D \xrightarrow{2} E \xrightarrow{2} F^\sharp \xrightarrow{1} G \xrightarrow{2} A \xrightarrow{2} B \xrightarrow{2} C^\sharp (\xrightarrow{1} D)$$

When one says 'to play a melody in a D major', it is meant that the melody is being created by thinking about that key and its notes. Often that means that the melody notes are almost all notes of the key.

Some scales can be made by rotating the list of intervals. For example, if you rotate the ionian scale to the left 5 times, you get the aeolian (minor) scale with intervals $\{2, 1, 2, 2, 1, 2(, 2)\}$ The A-aeolian scale is built using these intervals:

$$A \xrightarrow{2} B \xrightarrow{1} C \xrightarrow{2} D \xrightarrow{2} E \xrightarrow{1} F \xrightarrow{2} G (\xrightarrow{2} A)$$

[6]For example, when a note ends, the note length can be saved in the song structure where the note has been initiated.

rotation	name	intervals
0	ionian	$\{2,2,1,2,2,2(,1)\}$
1	dorian	$\{2,1,2,2,2,1(,2)\}$
2	phrygian	$\{1,2,2,2,1,2(,2)\}$
3	lydian	$\{2,2,2,1,2,2(,1)\}$
4	mixolydian	$\{2,2,1,2,2,1(,2)\}$
5	aeolian	$\{2,1,2,2,1,2(,2)\}$
6	locrian	$\{1,2,2,1,2,2(,2)\}$

Table 1: All church modes.

Scales created by rotating the ionian scale are called church modes and are listed in Table 1.

These church modes are often used in music and provide a base of keys for the musician to pick from.

2.2.4 System Overview

Now that there are definitions for (predefined) rhythm and key signature, there is one rule left to implement:

> All participants must be able to play music on an instrument that enables the other participants to interpret the instrument output as music.

For the setup of one musicians playing a melody, and a computer agent playing some harmony, this means that:

1. **The musician is able to know the harmony played by the agent**
   For the musician, it would be convenient to hear the harmony, just like in an 'all-human' jam session. There are multiple (existing) ways of playing sound by a computer system. Of course, it is possible to play the frequencies of the harmony through speakers attached to the system. In this case, MIDI [MMA04] will be used to send notes to an instrument which is able to receive and play MIDI.

2. **The agent is able to know the melody played by the musician**
   The musician has to be able to play a melody on an instrument and provide it as input for the agent in some way. Other techniques can be used, but one of the simplest is to use a MIDI instrument that will automatically parse the musical notes played on the instrument and send it to the system using the MIDI protocol [MMA04]. A MIDI note consists of an integer value for its pitch, and an integer value for its velocity. When receiving a MIDI note, the system has to figure out where the note was played relative to the song structure. The note gets quantized according to the quantization amount:
   As all notes in the song structure have the same length $l = \dfrac{60}{n \cdot \text{Tempo}}$ (described in 2.2.1), an input note played at time t (in seconds, relative to the start time of the song) is the $\lfloor \frac{t}{l} \rceil^{\text{th}}$ note in the song (rounded to the nearest integer).

With respect to all previous definitions, Figure 2 shows a general overview of a system that creates an environment similar to a real jam session.

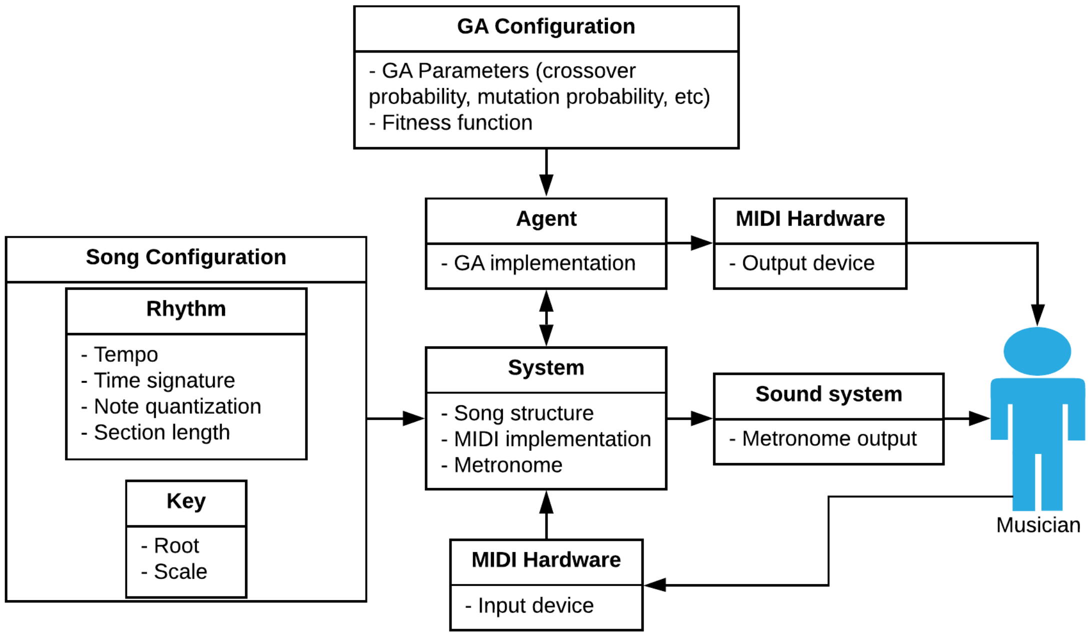


Figure 2: An overview of the different components of a (digital) jam session.

As the figure shows, there is a predefined song configuration that the system uses. The configuration contains information about the rhythm and the key signature. When the jam session is playing, the system will keep track of the song structure and listen to MIDI hardware to insert melody notes in the structure. A software metronome will be used to tell the musician about the time signature and the tempo.

The agent will use a genetic algorithm to continuously generate harmony and will output it to a MIDI instrument, so that the musician would be able to hear it as if someone else was actually physically playing the instrument. The agent will use the song structure (including the melody that has been played) as input for the genetic algorithm in some way.

2.3 Agent

Now that there is an environment for the AI to understand the key and time signature, and it is specified how the musician and the AI can communicate music with each other, the other question can be answered: What and how is the computer agent supposed to play during the jam session? In the beginning of 2 it is already indicated that the agent should be able to harmonize the melody that is being improvised. How can harmonization be realised?

2.3.1 Harmony

Harmony is most often understood to be the product of combining multiple pitches with each other. Often a melody is harmonized by playing chords – combinations of notes played simultaneously – under the melody to accentuate a certain tonal structure. Chords are a useful way to give context to a melody.

Triads As chords are combinations of notes, the intervals between these notes are often what describes the chords. In this case an interval is the number of semitones above the root – this means that the root note has an interval of 0 and, for example, a perfect fifth has an interval of 7. Typically, the minimal form of a chord is called a triad, and consists of three unique notes. In western music, the intervals of 'consonant' sounding chords are mostly built by stacking thirds on top of each other. This can be either a minor third (3 semitones above a note) or a major third (4 semitones above a note). For example, a major triad is built by picking a root and then stacking a major third and a minor third on top of each other:

$$C \xrightarrow{\text{major third}} E \xrightarrow{\text{minor third}} G$$

Relative to the root, the intervals of a major triad are $\{0, 4, 7\}$. Table 2a shows the most important triad intervals and their corresponding names.

Additionally, there are triads called sustained triads, where the third of the triad is replaces with a second. These sustained triads are used to create some ambiguity (as it can be interpreted in multiple ways – it could be in a major or a minor context).

Extensions Chords can consist of more than three intervals, and are typically used to add more color. To extend a triad to have more than three intervals, simply stack more thirds. By stacking a major or minor third, the first extensions are the minor seventh (10 semitones above a note) and major seventh (11 semitones above a note). Additionally, more thirds can be stacked to include the ninth (14 semitones), the eleventh (17 semitones), and the thirteenth (21 semitones). After adding the thirteenth, stacking another third on the chord would result in a note 24 semitones above the root. However, that is just the same note as the root but two octaves up. Therefore, the most notes you can put in a chord using this structure (before you reach the root again) is 7 – the root, the third, the fifth, the seventh, the ninth, the eleventh and the thirteenth.

Aside from all the extensions in thirds, it is also possible to add the sixth to the chord. The name of a particular extension interval set comes from the highest interval we stack thirds up to. All extension intervals and their corresponding names can be found in Table 2b.

Alterations To create tension in a chord, some of the extensions are altered to be a half semitone up or down. For example, a C^9 (intervals $\{0, 4, 7, 10, 14\}$):

$$C \xrightarrow{\text{major third}} E \xrightarrow{\text{minor third}} G \xrightarrow{\text{minor third}} B\flat \xrightarrow{\text{major third}} D$$

Can be altered to become a $C^{7(\flat 9)}$ (intervals $\{0, 4, 7, 10, 13\}$):

$$C \xrightarrow{\text{major third}} E \xrightarrow{\text{minor third}} G \xrightarrow{\text{minor third}} B\flat \xrightarrow{\text{major third - 1}} D\flat$$

(a) Common names and intervals of a triad.

name	shorthand	interval set
minor	min	$\{3, 7\}$
major		$\{4, 7\}$
diminished	dim	$\{3, 6\}$
augmented	aug	$\{4, 8\}$
sus2	sus2	$\{2, 7\}$
sus4	sus2	$\{5, 7\}$

(b) Common names and intervals of extensions.

name	shorthand	interval set
6	6	$\{9\}$
minor 7	7	$\{10\}$
major 7	maj7	$\{11\}$
minor 9	9	$\{10, 14\}$
major 9	maj9	$\{11, 14\}$
minor 11	11	$\{10, 14, 17\}$
major 11	maj11	$\{11, 14, 17\}$
minor 13	13	$\{10, 14, 17, 21\}$
major 13	maj13	$\{11, 14, 17, 21\}$

Table 2: Note intervals in chords and their corresponding names. In triads, the name 'major' often gets omitted in naming the chord. In extensions, the name 'minor' often gets omitted in naming the chord.

These altered notes make the chord more dissonant, which will create a stronger resolution when the notes are resolved.

These alterations (and most upper extensions) are not used too often in most music genres, with the exception of jazz, which is known for its dense harmonic language.

Not all extensions can be altered to create tension. For example, the ♭11 is actually the same note as the third, and the ♯13 is actually just the minor 7.

Inversion One final chord feature is inversion. As chords are essentially their root and a list of intervals, we could choose to play these intervals in a different order. Playing intervals $\{0, 3, 7\}$ will sound about the same as $\{3, 7, 0\}$, but some notes will be played an octave up or down, which can be useful for voice leading. For a set of n intervals, there will be n inversions.

With the previously mentioned principles, a chord C can be defined as a 5-tuple

$$C(R, T, E, A, I)$$

$$\begin{aligned} \text{where: } & R : \text{root note}, R \in \{C, C^\sharp, D, D^\sharp, E, F, F^\sharp, G, G^\sharp, A, A^\sharp, B\} \\ & T : \text{chord type}, T \in \{\text{major}, \text{minor}, \text{diminished}, \text{augmented}, \text{sus2}, \text{sus4}\} \\ & E : \text{chord extension}, E \in \{^6, ^7, ^{\text{maj7}}, ^9, ^{\text{maj9}}, ^{11}, ^{\text{maj11}}, ^{13}, ^{\text{maj13}}\} \\ & A : \text{chord alterations}, A \subseteq \{\flat 9, \sharp 9, \sharp 11, \flat 13\} \\ & I : \text{chord inversion}, I \in \mathbb{N} \end{aligned}$$

For a chord of size n (which is to say, containing N notes), the number of possible values for each feature are found in Table 3.

feature	R	T	E	A	I
possible values	12	6	9	$2^4 = 24$	n (see 2.3.1)

Table 3: The amount of possible values for each chord feature.

That means there are $15.552n$ different chords of size n. As discussed in 2.3.1, triads are the smallest chords. According to 2.3.1, there is a limit of 7 in the number of notes (thirds) to add to a chord.

With this system, since there are $15.552n$ possible chords of size n for any $3 \leq N \leq 7$, this means that there are

$$\sum_{n=3}^{7} 15.552n = 388.800$$

possible chords. How would an agent be able to choose harmony for some improvised melody when there are so many chords to choose from?

2.3.2 Genetic Algorithm

An evolutionary approach is fitting to solve the problem of generating some harmony in real time as an analogy can be made between a genetic algorithm and the way a typical musician would use creativity to improvise over music. Generally, improvised musical ideas are often formed from experience (for example, having heard a typical musical idea like a chord progression in another song). Sometimes ideas are combined with others, and sometimes they receive minor changes to make them spicy. These ideas are evaluated according to some personal preference and ultimately the musician will pick one to play.

In a similar fashion, a genetic algorithm is used to crossover, mutate and evaluate solutions [7]. It can build a population (the experience of the musician), and try to find interesting combinations and alterations until it has found an idea that is worth playing according to some process of evaluation (comparable to the personal preference of the musician).

To harmonize the melody, the genetic algorithm could simply generate chords. However, generating individual chords and evaluating them will most likely produce a more random sounding movement. To say anything about the properties of multiple chords, the representation of an individual could be a chord progression, an array of chords. This way, the fitness function could prefer combinations of chords that give a sense of direction to the music. Building an initial population with these kind of chord progressions will hopefully make sure this property will (at least partially) be retained.

2.3.2.1 Population Seed To give the algorithm some sense of what chord progressions to use, a database of popular chord progressions [Hoo20] is used. The common chord progression database contains scale steps and possible chord modifications (modifications to chord type – major, minor, etc – and chord extension – maj7, min7, etc). The system will generate the chords corresponding to that progression for the current key signature.

[7]It is worth noting that a genetic algorithm is typically used to find the solution with the best fitness value. With music, there doesn't exist a highest-scoring magical idea for any situation. It is better to pick a solution with a 'good enough' fitness value. The lower this threshold, the more creative it will get. The higher the threshold, the more it will adhere to the rules of the fitness function

If the population size is larger than the size of progressions of the database, the remaining progressions contain random chord triads from the current key signature. For the random chord triads, a random note of the scale of the current key will be picked for every chord, where the chord will be built using the third and fifth (relative to the scale) from that note.

2.3.2.2 Mutate

The algorithm will mutate an individual in the population with a probability defined in the GA configuration. Mutating an individual here means 'changing one or more features of a chord progression'. As talked about in 1.2.3, being able to mutate solutions to find surprising ideas in the conceptual space is an important feature of a genetic algorithm. As the conceptual space is a subset of the actual logical search space of the algorithm, exploring the conceptual space is equivalent as exploring the logical search space, where the fitness function limits the eventual outcome to output that exists in the conceptual space.

To show off this feature, exploring the search space should be minimally restricted – the mutation function shouldn't have a particular bias on where to look for new and interesting solutions. Because the exploration isn't restricted so much, there is a chance of 'undesirable' solutions being generated, but it is the job of the fitness function to filter these solutions out.

If the genetic algorithm is evolving individuals in the form of a chord progression, that is, an array of chords, the search space of the algorithm (not to be confused with the conceptual space, as said in 1.1.1) can be visualised as a collection of all the possible chord progressions. In this case, as the system generates a chord progression for a specific section, the search space consists of all chord progressions of length N, where N is also the amount of measures in a section.

In this search space, it makes sense to think of chord progressions that have a lot in common are close to each other. Two chord progressions with exactly one different feature are direct neighbours. Since one individual is a chord progression, the set of features of an individual is actually the combination of the set of the features of its chords. The features of a chord are specified in 2.3.1 as a 5-tuple. If the number of possible values of a chord feature F_i is $S(F_i)$, for any chord progression P with N chords, the amount of neighbours of P would be:

$$N \sum_{i=1}^{5} S(F_i) - 1$$

As 2.3.1 described all the values for chords of size n, the calculated amount of neighbours of chord progression containing chords of size n is $N(11 + 5 + 8 + 23 + n - 1) = N(46 + n)$.

As seen in 2.3.1, the minimal size of a chord feature is 3, so for any search space for chord progressions of size N there are at least $N(46 + 3) = 49N$ neighbours. For example, for the typical chord progression size 4, any chord progression has at least 196 neighbours.

As said before, exploring the search space should be minimally constricted, so the algorithm should be able to at least reach all of a chord progression's neighbours when doing a mutation, with equal probability.

> Therefore, an individual will be mutated by changing a random chord feature of a random chord of its chord progression.

Now, one could argue that evolving a chord progression to only one of its neighbours is too slow, and that it would require too many iterations to reach chord progressions that sound much different from the initial one. After all, as described in 2.3.1 there are 388.800 possible chords to make with this system. That means for a chord progression with size N, there are $N^{388.800}$ possible chord progressions. It would require a significant amount of iterations to get to a relatively different place if only $49N$ neighbours are reachable each iteration. There is a risk of the chord progressions sounding too much like the initial chord progression the algorithm started with, which wouldn't be too much attributed to as creative.

That is why in addition to the possibility of changing a random chord feature, another mutation is introduced [8]. A chord progression can also be mutated by changing a random chord in the chord progression to be a random chord triad of the current scale. In addition to the last mutation, the crossover function will make sure that the algorithm is able to evolve a chord progression in bigger steps than just its neighbour.

2.3.2.3 Crossover The algorithm will try to mate two individuals with a probability defined in the GA configuration. To explore the search space with bigger steps, it would make sense for a crossover function to take half of the features of both individuals and combine them.

Crossover will produce two children for any two individuals, the 'mother' M and 'father' F in the population. Two children are created by mixing the first and second halves of the mother $M1-M2$ and the father $F1-F2$ in such a way that the children can be described as $M1-F2$ and $F1-M2$. If a chord progression is of uneven length, the first half will be one chord larger than the second half.

2.3.2.4 Fitness Function Based on the current state of the improvised piece S, the fitness function is supposed to evaluate some chord progression P by giving it a numeric value. The genetic algorithm will rate every chord progression in the population so it can select certain individuals to survive the next iteration. At minimum, the form of the fitness function should be $F_{itness} : (S, P) \rightarrow \mathbb{R}$.

To have a more fine-grained control of the transformational creativity of the program, the fitness function needs to be easily modifiable. A fitness function containing multiple smaller evaluation functions would be convenient. The smaller components could be modified without drastically changing the whole fitness function, but large modifications are also possible.

These components, called fitness evaluators, give a score according to specific aspects of the chord progression in relation to the current state. A higher score indicates a higher desirability of a feature of the chord progression. To make the fitness evaluators behave uniformly, the form of any fitness evaluator E would be $E : (S, P) \rightarrow [0, 1] \in \mathbb{R}$, where a score of 0 means lowest desirability, 1 indicates highest desirability, and where 0.5 shows indifference towards a progression.

The fitness function is able to use a specific combination of these fitness evaluators to calculate a total fitness value. Fitness evaluators are given a factor $x \geq 0$ to indicate their importance relative to the other evaluators.

[8] This is not the only measure we take to make sure the algorithm is able to explore the search space in bigger steps. Another will be discussed in 2.3.2.3.

For a specific configuration of n fitness evaluators and their corresponding factors $\{(E_1, x_1), (E_2, x_2), ..., (E_n, x_n)\}$, the fitness function averages the scores of all evaluators to find the fitness of an individual:

$$F_{itness}(S, P) = \frac{\sum_{i=1}^{n} x_i \cdot E_i(S, P)}{\sum_{i=1}^{n} x_i}$$

As this function calculates the average of the fitness evaluators, the domain of the fitness function would also be $F_{itness} : (S, P) \rightarrow [0, 1] \in \mathbb{R}$.

2.3.2.5 Selection

Every iteration, after calculating the fitness of every individual in the population, only a portion of individuals get selected to survive. The selection is different based on whether or not crossover and mutation happen.

The evolving process is illustrated in pseudo-code in algorithm 1. This algorithm has been heavily inspired from a Javascript implementation of a genetic algorithm called `genetic-js` [sub20].

Here, `MaybeMutate` is a function that returns either the original individual or a mutated version according to the mutationProbability talked about in 2.3.2.2. The crossoverProbability has been talked about previously in 2.3.2.3.

Algorithm 1: Selection algorithm for evolving the population

```
Result: The new evolved population
newPopulation = [ ];
while (newPopulation.size < oldPopulation.size) do
    if (crossoverProbability and newPopulation.size + 1 < oldPopulation.size) then
        individuals = CrossoverSelect(oldPopulation);
        newPopulation.add(Crossover(MaybeMutate(individuals)));
    else
        individual = MutationSelect(oldPopulation);
        newPopulation.add(MaybeMutate(individual));
    end
end
```

For crossover, two individuals need to be selected. For mutation, only one individual needs to be selected. The selection functions `MutationSelect` and `CrossoverSelect` are described below:

1. `MutationSelect`
   The function will select 3 random individuals and it will pick the one with the largest fitness value.

2. `CrossoverSelect`
   The function will select 4 random individuals and it will pick the two with the largest fitness values.

These selection functions are used in the `genetic-js` implementation. Other selection functions can be used and are open for exploration. For now, only these two will be considered.

2.3.2.6 Thresholds The genetic algorithm does not always need to find the most optimal chord progression for any given state. Just like having a conversation, it is rather unimportant to think about what would be the absolute best answer to give. Instead, there are multiple answers that would each lead to different interesting outcomes. Similarly, deciding to explicitly search for and use an optimal chord progression would be limiting the creativity (or randomness) of the algorithm. Therefore, there should be some configurable thresholds for when the algorithm should stop evolving a population of progressions.

There are two thresholds that influence what individual ultimately gets picked and whether the algorithm should continue evolving:

1. **Fitness Threshold**
   This threshold decides the lower limit of the fitness score of a chord progression that could potentially be used as output. As described in 2.3.2.4, the fitness function gives a score between 0 and 1, where 1 is most desirable. When the algorithm stops evolving, a random individual gets picked from the subset of all instances in the population that have a fitness score greater than the `fitness threshold`. Of course it could happen that there are no individuals that have a high enough fitness score; in that case, the individual with the highest score gets picked.

2. **Generation Threshold**
   Each iteration of the algorithm, the algorithm checks whether the population should continue to be evolved using a `generation threshold`. The `generation threshold` describes the percentage of individuals in the population that should have a higher value than the `fitness treshold` in order for the algorithm to stop earlier than its configured amount of iterations. A `generation threshold` with value 0.5 will stop the evolution process when half of the population has a fitness score larger or equal than the `fitness threshold`.

2.3.3 Real-time Harmonization

Now that it has been established how the agent would be able to produce chord progressions, the final question is what to do exactly with this ability.

Whenever the agent requires a chord progression, it will use the genetic algorithm to evolve a population of chord progressions for a specific section. The fitness evaluators described in the configuration (see 3) use the current state of the song, to evaluate new individuals. After the evolving process, the system will pick a random chord progression from the population that has a fitness value higher than the configured fitness threshold. If there is no such individual available, it will pick the individual with the highest possible value. This individual will get used as the chord progression for the section it was requested for, and it will get saved in the song structure.

In Table 4 it is described how the genetic algorithm will continuously be used to generate or alter the previous, current and future chord progressions of the piece in real-time. A section describes a set of consecutive measures that is as big as the configured length of chord progressions. In other words, for every chord in a progression, there is a measure in a section. Sections indicate the start of a new chord progression.

Section	Measure	Evolve section
s_{i-1}	2	s_{i-1}
	3	s_i
s_i	0	s_{i-1}
	1	s_i
	2	s_i
	3	s_{i+1}
s_{i+1}	0	s_i
	1	s_{i+1}

Table 4: The evaluation process of the system, around some section s, with 4 measures in a section. For every measure, it shows which section (chord progression) is being evaluated to possibly mutate into a better one.

It shows that for every section, its chord progression gets generated **before** it gets played (provided that the generation takes less than one measure). If a section has finished playing, it gets re-evaluated to see if the evaluation of the chord progression is still acceptable now that everything about that section is known. If, with the new information about the input, the evaluation gets below the set fitness threshold, the chord progression will be evolved until it is changed into a satisfactory one. This is to support the idea of a musician correcting himself after he has already played an idea. This way mistakes that happened due to the real-time aspect of the piece would not reflect in the evaluation of future sections. On all measures in between the first and the last, the system will evaluate the chord progression of the current section. Once the musician is playing a certain chord progression, he needs to constantly ask himself if it is still the right idea when the state of the song changes. If, according to his own evaluation, it isn't, he needs to either modify (mutate) his current idea as he is playing it, or go along with another idea that is at least satisfactory.

Constants Other than the configuration for the fitness function (the fitness evaluators and their corresponding factors), the other configurable constants that can influence the output of the algorithm are explained below.

size: The population size (the amount of individuals created in 2.3.2.1).

crossover: The chance crossover happens (as described in 2.3.2.3).

mutation: The chance mutation happens (as described in 2.3.2.2).

iterations: The iteration amount (the amount of times the population evolves by selection described in 2.3.2.5).

fittestAlwaysSurvives: Whether the individual with the highest fitness score survives the next iteration, even when it doesn't get selected.

MaxResults: The maximum size of the returned subset of all individuals of the population that have a fitness score greater than the `fitness threshold`, when evolution stops.

fitnessTreshold: Explained in 2.3.2.6.

generationThreshold: Explained in 2.3.2.6.

3 Implementation

To perform experiments with the proposal of this thesis an implementation was made in Javascript using the new webMIDI api that is growing in popularity. Javascript is not the fastest language in terms of execution (and definitely not the most time critical – Most timer functionality works with the Javascript Event Queue [uc20a]), so there is no guarantee something will run on an exact specified time. However, this problem is relatively mitigated with Javascript Web Workers [uc20b], a form of multithreading. Also, Javascript does make for fast development, and opens up possibilities to host the system online in the future to collect statistics about the performance of the system or the algorithm and tweak it.

To implement the genetic algorithm, an Javascript implementation was made which is heavily inspired from `genetic-js` [sub20].

3.1 Song Configuration

For the experiments we will assume that the tempo, time signature and note quantization will be constant throughout the session. In this case, the rhythm is configured as follows:

Tempo 95 (bpm)

Time signature 4 (beats per measure)

Note quantization 4 (notes per measure)

Section length 4 (measures per section)

For the experiments we will also assume that the key signature will remain constant throughout the session. When the key signature is relevant to the experiments, it will be shown in musical notation.

3.2 Genetic Algorithm Configuration

Constants

The following standard configuration constants (their definitions can be read in subsections of 2.3.3) will be used in the experiments, unless explicitly stated otherwise:

size: 30

crossover: 0.8

mutation: 0.5

iterations: 80

fittestAlwaysSurvives: true

MaxResults: 100

fitnessTreshold: 0.7

generationThreshold: 0.6

Evaluators

Evaluators are the components of the fitness function that is discussed in 2.3.2.4. These resemble the values of the 'AI musician': What makes a good chord progression given the previous part of the improvised piece containing a melody?

Below are the evaluators used in the experiments in Chapter 4. These form an example of what the components of the fitness function could look like, and the aim for these particular evaluators is to showcase the features of the genetic algorithm. However, there are undoubtedly many better evaluators for other specific use cases.

1. `Progression similarity`
   By giving a better evaluation on chord progressions that have just been played, the system will have a slight preference to play the same chord progression unless there is reason to play anything else. Most of the time during jam sessions, it is unusual to play a totally different chord progression every time. This evaluator will give a score equal to the proportion of matching notes of the chords of the previous progression. To illustrate, if two chord progressions are exactly the same, the score will be 1. If half of the chords are exactly the same whilst the other half does not share any notes, the score will be 0.5.

2. `Melody similarity`
   As the behaviour of this system should be to support the melody, a chord progression should be better if the chords of the progression contain notes that are also played in the melody at that time. When generating a new chord progression for a future section, there are not any notes yet for this evaluator to analyse. The need for a system arises to anticipate certain notes based on the history of the piece. For now, to keep things as simple and straightforward as possible, this evaluator will rate a chord progression based on its similarity of the melody that was played in the previous section.

   The similarity is measured by the percentage of melody notes shared by the chords of the progression when that particular chord would be played.

   This gives the result that this evaluator expects the melody for the next section to be about the same as the melody in the previous section. While still being interactive, there is definitely room for future improvement.

3. `Chord complexity`
   This evaluator rates a progression based on the complexity of its chords. Simpler chords often sound less dissonant than chords with lots of alterations.

   For any chord, its complexity is determined by its type, extension and alterations:

   (a) **Type**

   $$C_{complexity}(\text{type}) = \begin{cases} 1 & \text{if } type = \text{diminished} \\ 0.4 & \text{otherwise} \end{cases}$$

   (b) **Extensions**
   The value returned for any given extension can be seen in table 5.

extension	∅	6	7	maj7	9	maj9	11	maj11	13	maj13
value	0.5	0.4	0.3	0.6	0.7	0.8	0.9	0.8	1	1

Table 5: Complexity values for chord extensions in an individual.

(c) **Alterations**
The percentage of alterations compared to the maximum amount of possible alterations:

$$C_{omplexity}(alterations) = \begin{cases} 0 & \text{if } alterations.length = 0 \\ 1 & \text{otherwise} \end{cases}$$

With these complexity components defined, a chord complexity is then calculated by taking a weighted average of the complexity of the type, the extension, and the number of alterations:

$$\begin{aligned} C_{omplexity}(chord) = & (C_{omplexity}(chord.type) \\ & 2 \cdot C_{omplexity}(chord.extensions) \\ & 9 \cdot C_{omplexity}(chord.alterations)) \\ & \div 12 \end{aligned}$$

As individuals with a low chord complexity would be more desirable in this case, the value is inverted to give the final fitness value for individuals with n chords: $F_{itness}(individual) = 1 - (\sum_{i=1}^{n} C_{omplexity}(individual.chord_i) \div n)$.

4 Experiments

To determine some of the claims made in 1.2.3 and 2.3.2, some experiments will be conducted. The setup of the experiments is defined in 3, unless stated explicitly otherwise. All generated output used for the experiments and the implementation of the system can be found in appendix G.

To analyse all the output in the real-time jam session, it is implemented that whenever the jam session stops, an output file is generated. This output file contains the configuration (the time and key signature and the genetic algorithm configuration), the melody played by the musician, and the chords played by the agent. Additional features of the session consist of the execution time of the genetic algorithm each time a progression needed to be generated, fitness value of every generated progression that was at some point used in the system, and the average execution time of the genetic algorithm. The output file is saved in LilyPond format [tea20]. LilyPond is used to generate a MIDI file, which is loaded into MuseScore to generate a PDF. The musical notation that is seen in the experiments are excerpts of the PDF's generated by MuseScore.

4.1 Thematic Development

Previously it was reasoned that a genetic algorithm would be able to possess the capability of building on a particular idea. In the context of an improvisation, a musical idea is thematical if it is a recurring idea within the piece. Thematic development then is the altering of this recurrent idea by doing some permutations on it.

With the `ProgressionSimilarity` evaluator, a preference is given to the chord progression that is played previously. By giving similar chord progressions a higher fitness score, these individuals would survive longer when evolving, thus giving them more chance to mutate. Examples of recurring chord progressions can be seen in appendices A, B and C. For instance, one occurrence can be seen in figure 3.

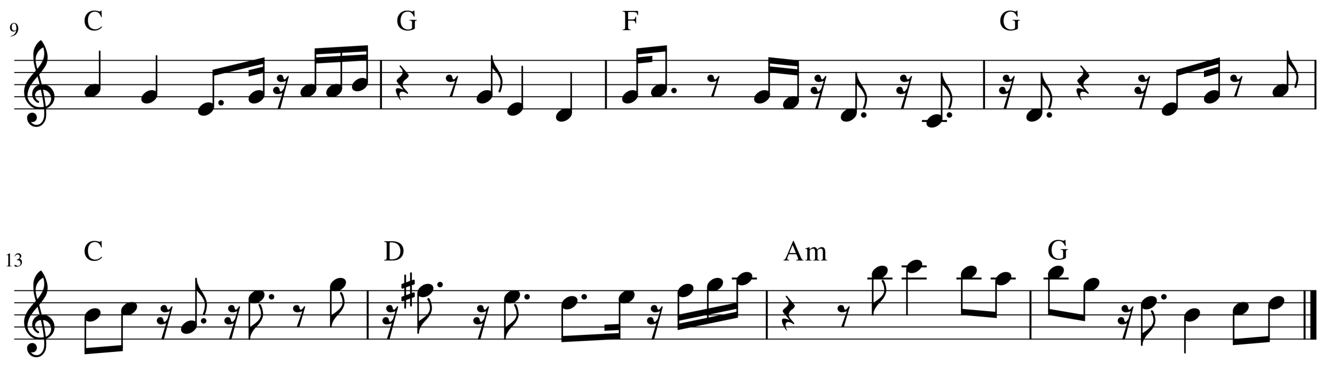


Figure 3: An excerpt of the program output (complete piece can be seen in appendix A) showing the development of a chord progression.

The notes of the chords of the progressions $C - G - F - G$ (measure 9-12) and $C - D - Am - G$ (measure 13-16) are very similar. As seen from 6, the similarity of these progressions is 75%.

progression	**chord 1**	**chord 2**	**chord 3**	**chord 4**
$C - G - F - G$	$C\ (C - E - G)$	$G\ (G - B - D)$	$F\ (F - A - C)$	$G\ (G - B - D)$
$C - D - Am - G$	$C\ (C - E - G)$	$D\ (D - F\sharp - A)$	$Am\ (A - C - E)$	$G\ (G - B - D)$
shared notes	$\frac{3}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$

Table 6: Similarity of chord notes for the two consecutive progressions from appendix A starting at measure 9.

4.2 Chord Complexity

To verify that the fitness function has direct control over specific features of the generated output, the usage of the `Chord complexity` evaluator is analysed to see if the feature of chord complexity of the output can be controlled.

In Figure 4 the `Chord complexity` evaluator has been given a factor 0, so that it would not affect the fitness score anymore. Without this evaluator, a significant increase in altered notes and complex chords can be seen.

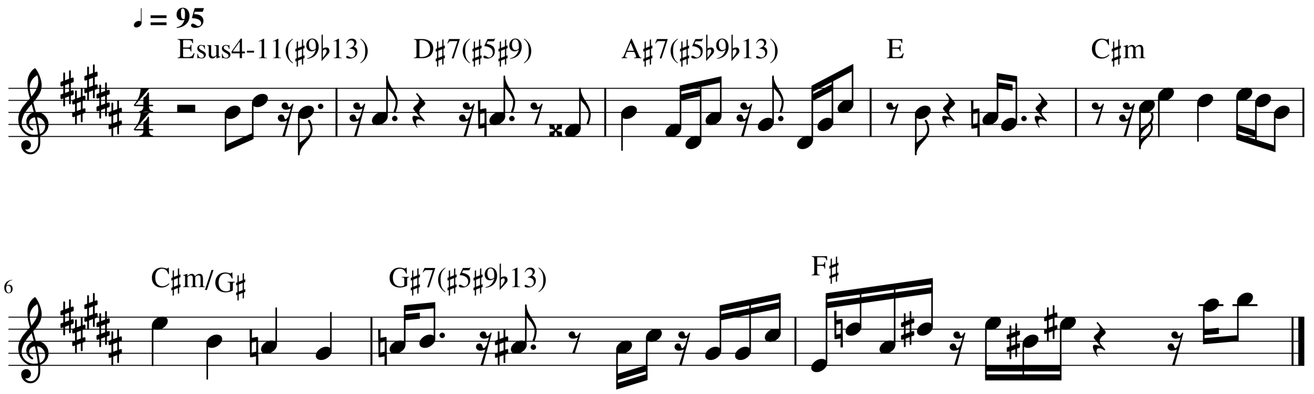


Figure 4: An excerpt of the program output without the `Chord complexity` evaluator.

Compared to the examples generated in the appendix (A, B and C) where the `Chord complexity` evaluator is present, there is an obvious difference in the complexity of the generated chords.

To support this above example, the diagram in figure 5 shows that for multiple sessions, the chord complexity continues to stay constant as long as the `Chord complexity` evaluator affects the fitness score.

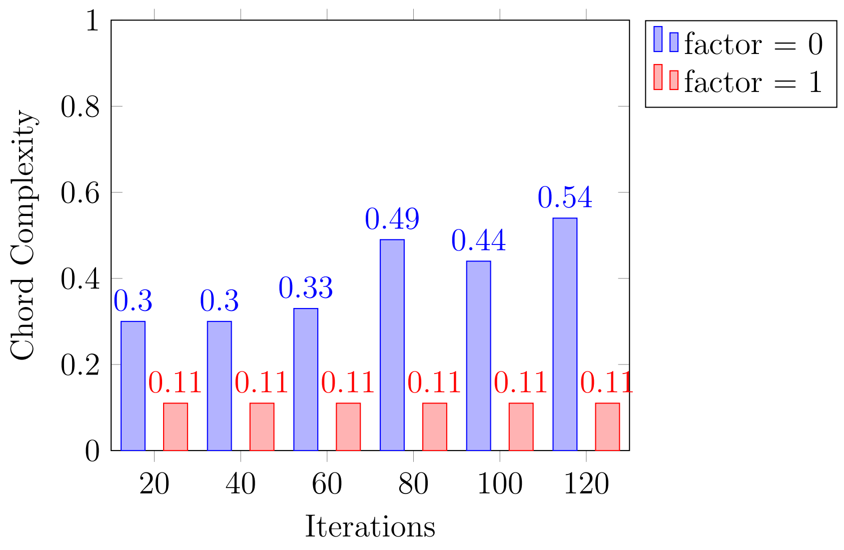


Figure 5: For a couple of different iteration values, the diagram shows the average chord complexity over five 10-section arrangements where the factor of the `Chord complexity` evaluator is 1 (the weight of the evaluator ensures that it significantly influences the outcome of the fitness function) and 0 (the evaluator has no influence on the the fitness function). The chord complexity is determined by the `Chord complexity` evaluator described in 3. Data can be found at appendix F.

It can also be seen that without the `Chord complexity` evaluator, the chord complexity grows with a larger iteration amount. Because the algorithm has more chance to mutate a particular individual when it has to perform more iterations, it makes sense that the average chord complexity also grows (as the mutation process can add alterations and change the extension and type of a chord). The fact that there are some discrepancies in the what should arguably be a linear growth of the complexity can be explained by the influence of the `Progression similarity` evaluator. This evaluator gives a preference to using only notes of the previous chord progression. If the session started with a chord progression with a low average chord complexity, the agent tends to stay close to these notes, therefore giving a lower average chord complexity to the arrangement. This can further be seen by the relatively high standard deviation of the data in appendix F where the factor = 0, compared to factor = 1.

With the `Chord complexity` evaluator active (factor = 1), chords with less complexity become more desirable for the genetic algorithm. Without any significant reason for other evaluators to choose an individual that has more complex chords, it is clear that the algorithm tries to find chord progressions with a minimal chord complexity, thus controlling the feature of chord complexity in the output of the program.

4.3 Interactivity

In 3 it has been discussed that in order to have close to real-time interactivity which would make musical sense, an evaluator must exist that would be able to make assumptions about the musician's meaning of a combination of input notes. For now it is not necessary to go into details about making these assumptions, and therefore the simpler `Melody similarity` evaluator has been introduced. Even though this evaluator does not make the agent react quickly to input (it takes a whole section before the input is considered) it does give some indication if the system can be interactive to some extent.

By having the human musician arpeggiate the notes of some chord progression, it is expected that a very similar chord progression should be played in the next section.

In figure 6, the melody is arpeggiating the notes of the chord progression $C - G - G^{\sharp} - A^{\sharp}$. It can be seen that the first chord progression is very different from the arpeggiated input notes: $Dm - Fm - Bdim - Dsus2$, and it sounds very dissonant. If it was totally up to the `Progression similarity` evaluator, the next section would have produced a similar chord progression. However, as predicted, the `Melody similarity` evaluator makes sure that in the next section the chord progression changes to something that resembles the input more closely: $C - G - Fm - Bdim$. The first two chords in the progression have changed to the intended chords. What is interesting is that the intended $G^{\sharp}$ and $A^{\sharp}$ chords have been approached as Fm and $Bdim$ respectively. The notes of these chords are very similar, and the intention of the arpeggiated notes are put into a different perspective. Now, the melody sounds more like it highlights the chord progression $C - G - Fm7 - G7(\sharp 9)$, which is an interesting sound in itself.

It can be argued that the human participant does have a lot of control over the chord progression the agent produces, but not so much that the agent cannot come up with interesting ideas for itself.

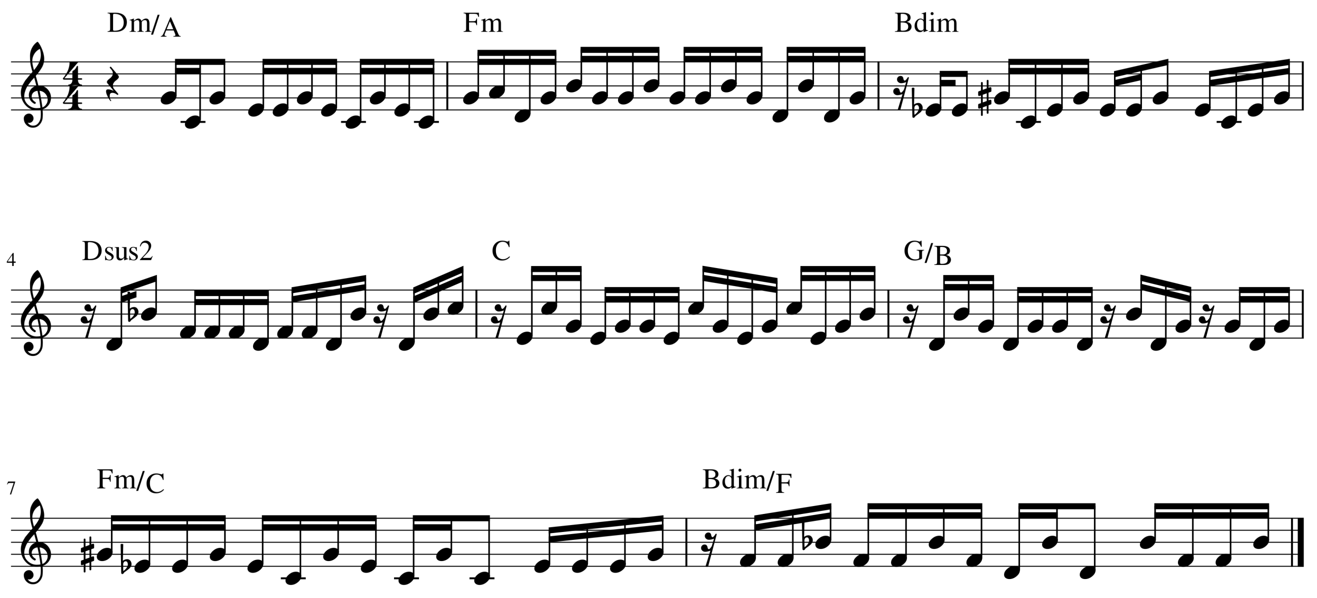


Figure 6: An excerpt of the program output (complete piece can be seen in appendix D) where the melody arpeggiates the chord progression $C - G - G^\sharp - A^\sharp$.

4.4 Execution Time

As another claim was that the execution time of a genetic algorithm might be too much with a complex fitness function, considering the real-time context. Here, the execution time is defined as the time from the moment the system requests a new chord progression from the genetic algorithm until the moment it is being placed in the song structure.

As discussed in 2.3.2.6, there is a `generation threshold` that stops evolving the population once a certain percentage of the population has surpassed the acceptable fitness value (the `fitness threshold`). Because at some measures the system is only re-evaluating an already existing progression instead of generating a new one (as defined in 2.3.3), the progression will have a high probability of being acceptable again (using the current evaluators, of course). The algorithm will then stop evolving population almost instantly, e.g. the algorithm will not spend any time on more iterations. The following experiments will consider this setup in addition to one where the `generation threshold` is not taken into account [9].

The configuration of the genetic algorithm will be considered as well as the population size and the amount of iterations (times that the population evolves).

Population and Iterations

Figure 7 shows a graph of how the population size and the iteration amount influence the execution time directly. The plots marked with a cross show the time for increasing population, the plots marked with a circle show the time for increasing iterations, and the plots marked with a triangle show the time for both increasing population and iterations.

[9] To force the system into going through all the iterations, a `generation threshold` of > 1 can be used.

Population size and Iteration amount on Execution time

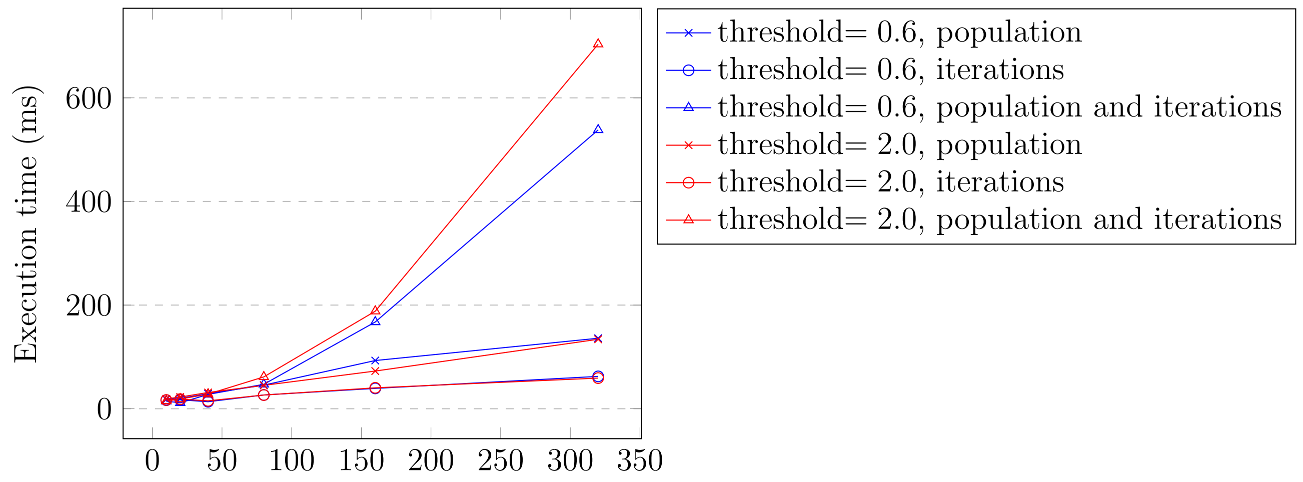


Figure 7: The graph shows the execution time for a growing population size and iteration amount, increased in steps of $10 \cdot 2^n$. The x-axis is defined as the population size, iteration amount, or both, as described in each plot. Red plots use `generation threshold`$= 0.6$, blue plots use `generation threshold`$= 2$. Data can be found at appendix E.

To get more insight into the execution time of the system, the blue plots show the execution time for a `generation threshold` of 0.6. The red plots show the execution time for `generation threshold` $= 2$.

Generally, the graph shows that the execution time increases as either the population size or the iteration amount increases. Individually, the population size and the iteration amount seem to increase the execution time linearly, and the `generation threshold` doesn't seem to make any significant difference. When both the population and iterations are increased, the execution time seems to increase exponentially. This would make sense, as the algorithm needs to spend time performing more evolutions on more individuals. Also not very surprising is that the graph shows that when the `generation threshold` is lower, the execution time is lower. After all, it is reasoned that the algorithm will just perform less evolutions.

5 Discussion

A real-time improvising agent that uses a genetic algorithm this particular way to create harmony provides an interesting perspective to look at improvisational creativity and interactivity. For one, it becomes clear how a genetic algorithm can be used to search for creative ideas according to Boden's model:

Combinational creativity can be achieved by implementing a crossover function to find unfamiliar combinations in already existing ideas.

Exploratory creativity of the agent can be influenced by the mutation function – it tries to find novel ideas that exist in the agent's conceptual space by making logical alterations of existing ideas.

Transformational creativity can be realised by incorporating a process that continually makes changes to the configuration of the fitness function. This way, the agent's conceptual space (and consequently, the agent's style of thought) is something that can be altered directly.

Reasoning and experimenting with the idea of using a genetic algorithm to generate improvised harmony in real time also gave insight as to how different levels of interactivity described by Cornock and Edmonds could be realised. While, for any interactive system that produces music, a *static* system would be impossible, the following other classifications can be incorporated in the system as follows:

dynamic-passive: The genetic algorithm can be used to find individuals according to a fitness function that evaluates them only using internal rules or output generated by the system itself.

dynamic-interactive: The outcome of the fitness function would have to be influenced by input of the participant(s). In this specific case, the fitness function evaluates individuals according to features of the melody played by the participant.

varying dynamic-passive: Here, the fitness function itself would need the possibility to be altered by the system, and even possibly by the participant(s). The changing of the fitness function over time influences the generated output, in such a way that the possibilities of interacting with the system can be different.

It can be seen from the experiments that a fitness function that evaluates individuals for various desired features is able to control these features in the generated output. The different components of the fitness function that are called fitness evaluators could be any function that rates an individual a number between 0 and 1 using the current state of the song. With the corresponding factors, it is possible to control the importance of a specific evaluator compared to the others.

The way that the agent uses the algorithm – evolving a chord progression every measure to generate harmony in real time – is in some way limiting the reactivity of the agent. If the musician would play melody notes which would be desirable to have them influence the chord of the next measure, the agent wouldn't be able to deal with it in time. However, the system is not limited to evaluating its state only every measure. In theory, the agent can react as quickly as it takes for the genetic algorithm to evolve and generate a fit individual. It did make more sense to limit the agent to use the algorithm only every measure, from the fact that it was easier to implement and to experiment with [10]. The advantage of limiting the agent this way is that as long as the execution time of the algorithm doesn't exceed the time it takes for a measure to finish, the typical high execution time of a genetic algorithm is unnoticeable.

5.1 Further Research

There are obviously several areas to expand on, either to improve the proposed system, or to explore new relatable ideas.

One particularly important area to explore is the potential of this system to be *varying dynamic-interactive*. As discussed in 1.2.3, it could be interesting to implement some process that modifies the fitness function according to what is happening to the melody. The fitness function can be altered by adding, replacing or removing one or more fitness evaluators. A more subtle way of altering the function is to only alter the weights of specific fitness evaluators.

The potential of specific fitness evaluators has only been briefly touched on, and it would be really interesting to experiment with different fitness evaluators, and to find out how combinations of these evaluators result in a fitness function that creates specific features in the eventual output.

The fitness evaluators could even incorporate other algorithms for the benefits of their properties, like using a trained neural network to generate a certain fitness value for the evaluator [11]. Of course, the usage of fitness evaluators is limited by the real-time aspect in the sense that very complex evaluators will increase the execution time, possibly to the point where the algorithm simply is too slow to generate and play interactive output.

To make this system able to react more quickly to user input, it would be of value to look at finding another way for the agent to use the genetic algorithm effectively. If the algorithm is used continuously, an improvement in the agent's reaction time could potentially be made. As said before, in theory, the reaction time of the agent could be as fast as the execution time of the genetic algorithm.

In terms of execution time, what could help is of course optimizing the system. Using a more time-critical language, and reducing the proposed object structure of the song and an individual (a chord progression) are two solutions. The less information is contained in these structures, the faster it is to navigate through them, and to generate multiple generations of them. However, this would only decrease the execution time by some constant value, whereas increasingly complex evaluators would increase the execution time exponentially. The experiments show in 4.4 that the average execution time can be diminished by the `generation threshold`. It is possible that an even better threshold exists than the one used in the experiments.

Another aspect that could be explored further is the performance of playing. The aim of this system is to generate an improvised harmony for an improvised real-time melody. The output has been

[10] It was easy to analyse generated output because it was known at which exact time a chord progression was evaluated.

[11] Other possibilities of combining these approaches have been considered as well. For example, C. Chen and R. Miikkulainen [CCJC01] are creating melodies using evolving recurrent neural networks: A genetic algorithm is evolving different neural networks in order to control the explorative capabilities of the neural network generating the eventual melodies.

made relatively simple by just playing the chords every measure where there is a chord change. Real musicians on the other hand, don't necessarily play just one chord in this predictable manner. This area has actually already been researched somewhat: some of the previously referenced papers are about generating output for a specific chord progression [Bil94][Gra01]. They could be used in combination with this generation of harmony.

Some small improvement would be to incorporate the song configuration (the rhythm and key) in the song structure itself. This way, these parameters could be made variable, so that the system could switch keys or even time signatures during the improvised piece. That would also make it possible to incorporate key and beat detection for the melody. There are multiple studies that are already able to predict a key [ZKG05] and the downbeat [Col03] of a melody in real-time. This would open up possibilities of not having to specify a key or rhythm beforehand. After all, most real musicians would be able to play along with a song by guessing the key and rhythm themselves. As discussed in 2.2.2, the length of input MIDI notes has not been considered, but the system could undoubtedly benefit from being able to analyse the melody in a more detailed manner.

Finally, the analysis of the jam session of Cory Henry in 1.1.1 highlighted the fact that the communication between the participating musicians is not at all limited to their instruments. There would be a lot of potential to make a system like this even more interactive by analysing these communication techniques and trying to incorporate them in the system. Verbal and visual cues can be much more explicit about the improvisation, perhaps because these cues can happen without affecting the music directly. For example, imagine a part of the jam session where there is a slow build-up of tension. By adding more tension with their instruments, the musicians are able to communicate the amount of build-up over time. However, it is not immediately clear how the musicians would communicate the start of a new section using just their instruments. By using verbal cues, this is easily communicated.

6 Conclusions

In 1.1.1 Boden's model for creativity is used to categorize how *creative* ideas are formed, and how they could be formed by a computer system. The approach of generating ideas by using a genetic algorithm was picked, and by reasoning about Boden's model it was shown that multiple levels of creativity could be incorporated in such a system. A real-time improvising system using a genetic algorithm can be creatively interactive when the musician interacting with the system is able have control over the transformational creativity of the system – that can happen when the musician is able to provide input that will trigger some kind of alteration to the fitness function of the genetic algorithm.

A real jam session was analysed to find some key features that would give some insight in how a creative computer system would be able to participate in a jam session. It was pointed out that the control of the musical direction of the improvisational piece was a recurring and important concept. With this in mind, Cornock and Edmonds' work on interactivity has been used to find the categorization of interactivity in any jam session. This way, in order for a computer system to mimic this, the requirement could be made that the system would have to aim to be as interactive. It has been reasoned that such a system would at least have to be *dynamic-interactive*, and to provide the concept of directional control of a song, the system would have to be *varying dynamic-interactive*.

A computer system has been modeled out that is *dynamic-interactive* in the sense that the human musician participating in the jam session has some form of control over the output of the computer system, and it has been reasoned in 1.2.3 that this system could potentially become *varying dynamic-interactive* if it would incorporate some way of continuously altering the fitness function during the session, either by the system itself, the state of the song, or even by the human musician.

It is also explored that even with a complex fitness function, the computer system is able to generate output within a reasonable amount of time. The execution time of the genetic algorithm is reduced by introducing certain thresholds explained in 2.3.2.6. Additionally, while the system produces output in real time, way the chord progressions are generated as defined in 2.3.3 – using a genetic algorithm for each measure – makes it so that generating a chord progression may take as long as the duration of a whole measure.

However, in the experiments of 4.3 it is argued that this feature makes it difficult to see how to react quickly to a changing melody. At most, the system could only directly react to input that has happened one measure ago (because the agent evaluates the state of the arrangement every measure). Even though it is not yet clear on how to configure this system in such a way that the agent can immediately react to a melody that is being played to alter its chord progression (as seen in the `Melody Similarity` evaluator in 3), there are definitely possibilities that potentially handle this. For example, certain fitness evaluators can be made which focus on specific melodious patterns and prefer chord progressions that would be able to resolve the melody in the future. It is shown in the experiments (4.3) that in any case the system is at least interactive to a certain extent, and that it could have the potential to be more interactive with respect to the melody.

Acknowledgements

I would like to thank the Computer Science institute of Leiden, LIACS, for guiding me through the general process of creating this bachelor thesis. Even with the advent of the COVID-19 virus, they have managed to continue to guide me exceptionally well using online virtual methods.

Secondly, my gratitude goes towards Maarten Lamers for reading this thesis and providing it with very constructive commentary. It presented me with insights in improving on some minor but important details which would have been overlooked otherwise.

In addition, the never-ending support of my parents has been invaluable to me as they continually provided me a listening ear to talk to which enabled me to create a comprehensive mental model of the aspects talked about.

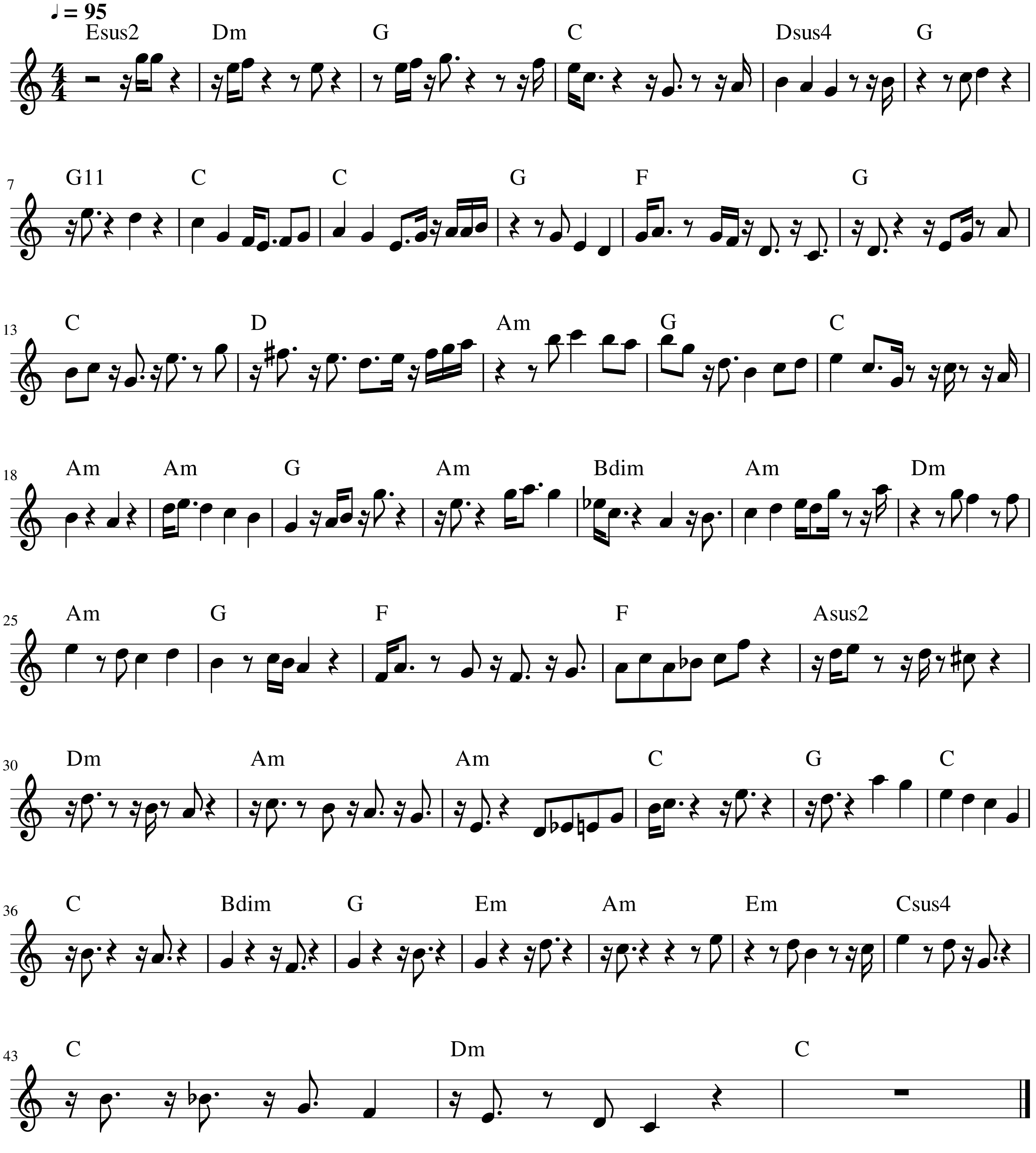
Lastly, I would like to mention the exceptional supervision I received by Edwin van der Heide. Without taking over the control of the direction of the thesis, he was able to spark my interest in especially the aspect of interactivity between human and computer and the importance of it regarding my goal of making a real-time improvisational system.

Appendices

GaConfiguration:
size: 30
crossover: 0.9
mutation: 0.4
iterations: 80
fittestAlwaysSurvives: true
maxResults: 500
fitnessThreshold: 0.7
generationThreshold: 0.6

A Example Output 1

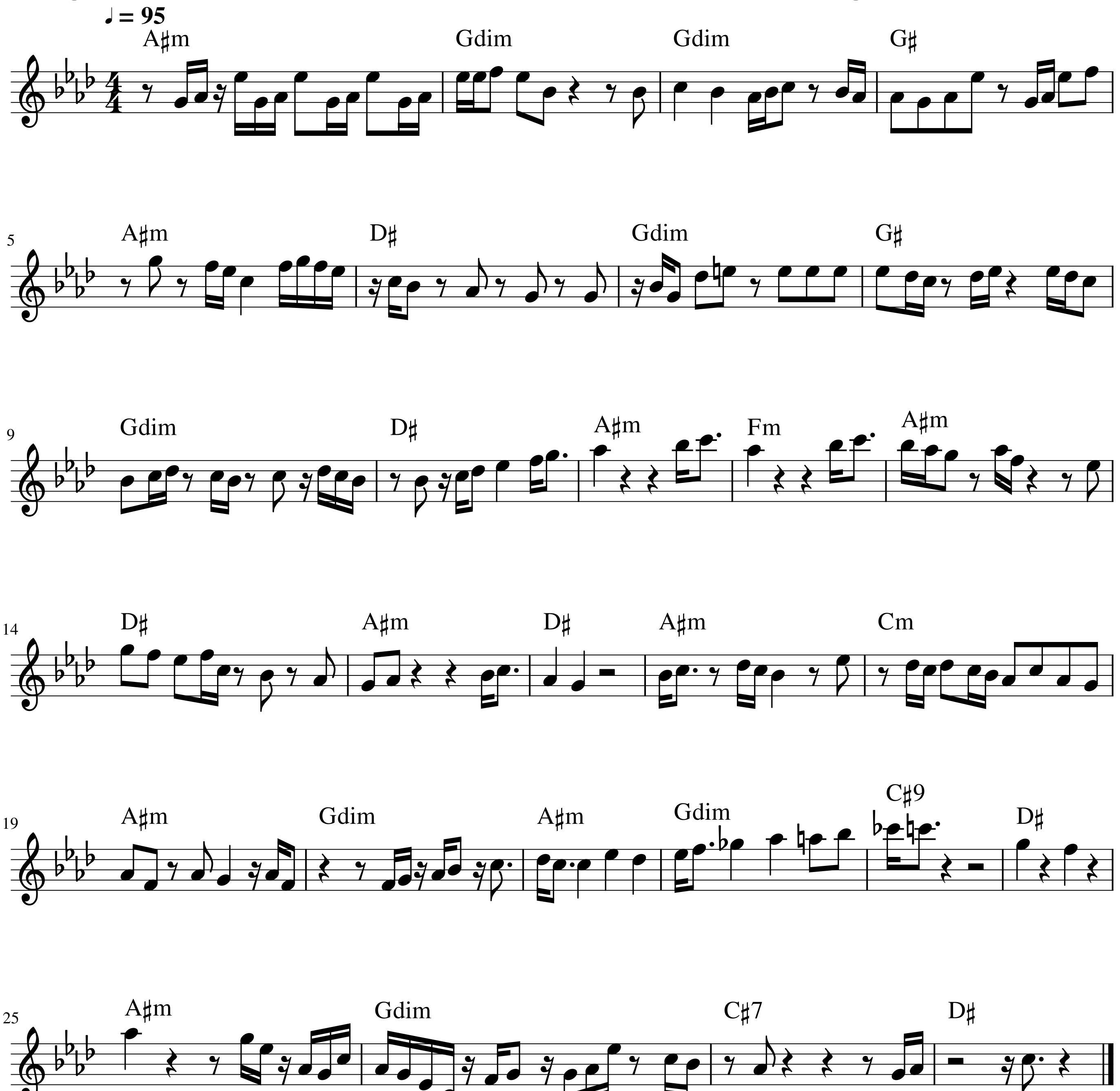
average execution time: 9.416666666666666ms



GaConfiguration:
size: 40
crossover: 0.4
mutation: 0.6
iterations: 80
fittestAlwaysSurvives: true
maxResults: 500
fitnessThreshold: 0.8
generationThreshold: 0.7

B Example Output 2

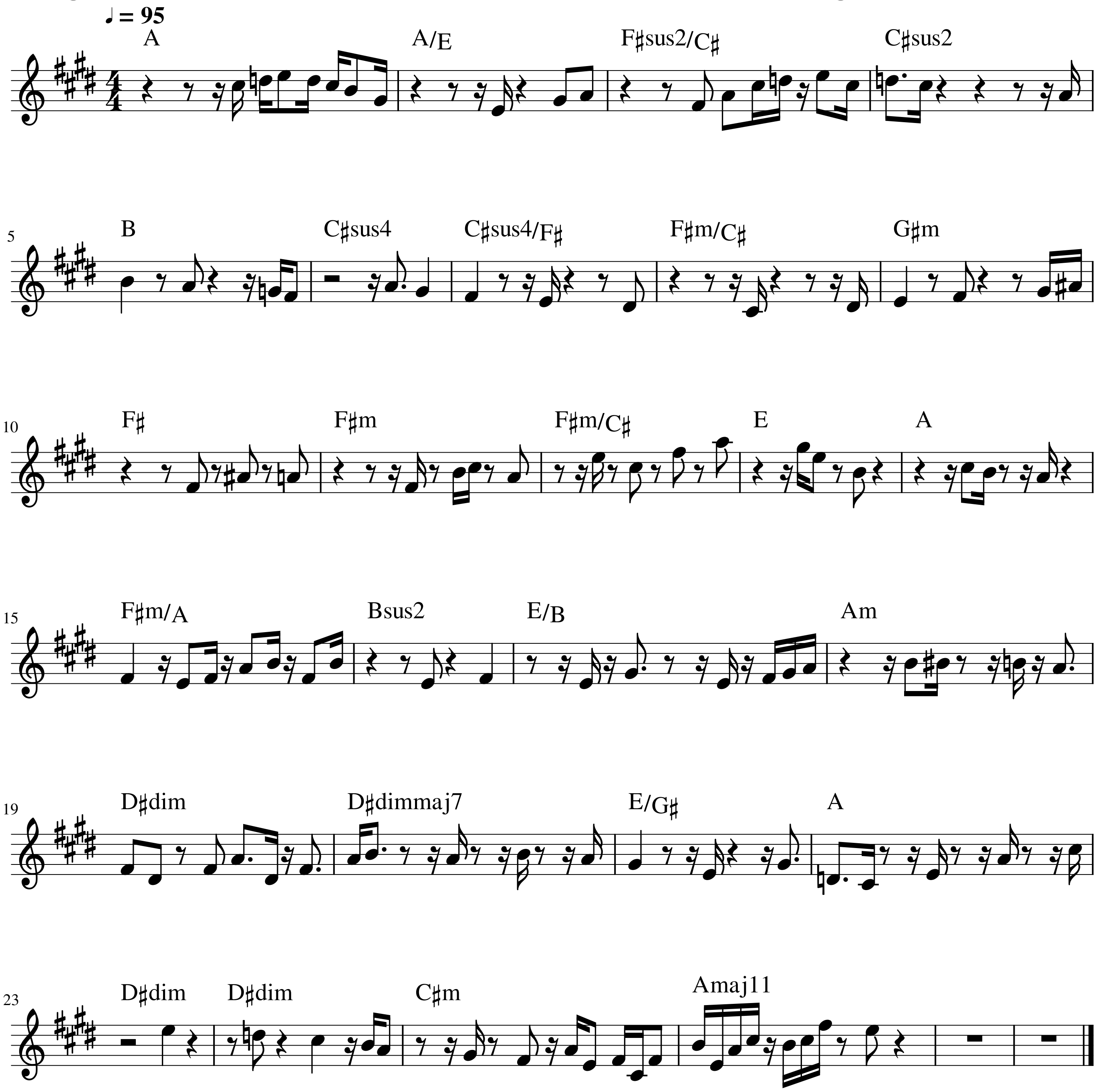
average execution time: 19.767857142857142ms



C Example Output 3

GaConfiguration:
size: 40
crossover: 0.4
mutation: 0.6
iterations: 80
fittestAlwaysSurvives: true
maxResults: 500
fitnessThreshold: 0.8
generationThreshold: 0.7

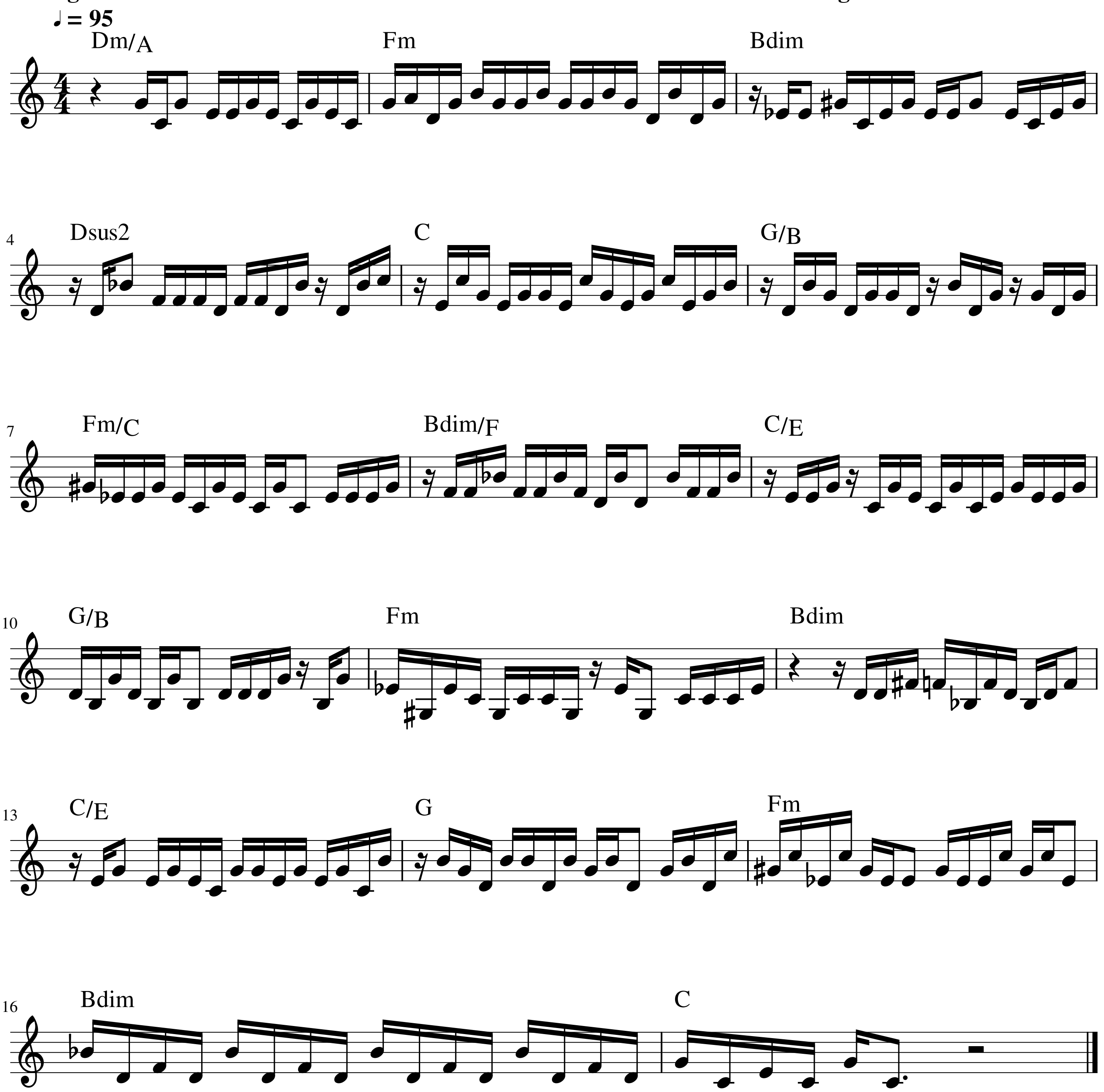
average execution time: 19.75ms



D Example Interactivity

GaConfiguration:
size: 30
crossover: 0.8
mutation: 0.5
iterations: 120
fittestAlwaysSurvives: true
maxResults: 100
fitnessThreshold: 0.8
generationThreshold: 0.7

average execution time: 48.35ms



E Execution Time Averages

	n	10	20	40	80	160	320
threshold = 0.6	**population**	17.8	19.9	29.1	45.5	93.0	136.0
	iteration	16.5	17.2	13.5	26.5	39.2	62.4
	both	14.6	11.4	27.6	47.1	167.4	538.1
threshold = 2.0	**population**	19.0	22.5	31.3	45.1	72.7	134.0
	iteration	16.8	18.5	15.2	26.4	40.3	59.1
	both	15.1	18.6	28.2	61.4	188.2	703.6

Table 7: The average execution time over a 10-section arrangement in milliseconds for different sizes of population and iterations for `generation thresholds` 0.6 (evolution stops when 60% of the population has a fitness score higher than the `fitness threshold`) and 2.0 (evolution always continues for the amount of iterations specified). Actual output can be found in appendix G.

F Chord Complexity Averages

	iterations	20	40	60	80	100	120
factor = 0	1	0.27	0.37	0.39	0.32	0.41	0.57
	2	0.34	0.31	0.36	0.55	0.57	0.55
	3	0.33	0.26	0.30	0.53	0.46	0.47
	4	0.29	0.34	0.38	0.64	0.39	0.46
	5	0.29	0.22	0.21	0.39	0.37	0.68
	average	0.30	0.30	0.33	0.48	0.44	0.54
factor = 1	1	0.10	0.11	0.11	0.11	0.11	0.12
	2	0.11	0.11	0.11	0.11	0.12	0.12
	3	0.10	0.11	0.11	0.11	0.12	0.11
	4	0.11	0.11	0.11	0.11	0.11	0.11
	5	0.11	0.12	0.11	0.11	0.11	0.11
	average	0.11	0.11	0.11	0.11	0.11	0.11

Table 8: Average chord complexity over five 10-section arrangements for different iteration amounts where the factor of the `Chord Complexity` evaluator is 1 and 0. The chord complexity is determined by the `Chord Complexity` evaluator described in 3. Actual output can be found in appendix G.

G Source Code

Source code that generated the outcome of the experiments, and the outcome of the experiments (LilyPond files, midi and MuseScore pdf's) can be found on `https://github.com/jeroenwzelf/MelodyHarmonizer`.

References

[Bel10] Stephen C.D. Bell. *Participatory art and computers: identifying, analysing and composing the characteristics of works of participatory art that use computer technology.* PhD thesis, Loughborough University of Technology, November 2010.

[Bil94] John A Biles. GenJam: A genetic algorithm for generating jazz solos. In *ICMC*, volume 94, pages 131–137, 1994.

[Bod09] Margaret Boden. Creativity in a nutshell. *Think*, 5:83–96, 09 2009.

[CCJC01] Risto Miikkulainen Chun-Chi J. Chen. Creating Melodies with Evolving Recurrent Neural Networks. In *International Joint Conference on Neural Networks*, 2001.

[CE73] Stroud Cornock and Ernest Edmonds. The creative process where the artist is amplified or superseded by the computer. *Leonardo*, 6(1):11–16, 1973.

[Coh06] Harold Cohen. AARON, Colorist: from Expert System to Expert. . *University of California at San Diego*, October 2006.

[Col03] Nicholas M. Collins. *Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems.* PhD thesis, Faculty of Music, University of Cambridge, 2003.

[DJP$^+$20] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music, 2020.

[Dri16] Jonathan Driedger. *Processing Music Signals Using Audio Decomposition Techniques.* PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[Gra01] Maarten Grachten. JIG: an Approach to Computational Jazz Improvisation. Master's thesis, Artificial Intelligence Institute of Barcelona, Spain and University of Groningen, Netherlands, January 2001.

[Gua09] Enric Guaus. *Audio content processing for automatic music genre classification: descriptors, databases, and classifiers.* PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2009.

[Hen20] Cory Henry. Cory Henry Workshop in Rio. https://www.youtube.com/watch?v=8cfpQdPY2pM, 2020. Accessed: 2020-07-23.

[Hoo20] HookTheory. Popular Chord Progressions. https://www.hooktheory.com/theorytab/common-chord-progressions, 2020. Accessed: 2020-07-01.

[LM17] Yi Luo and Nima Mesgarani. TasNet: time-domain audio separation network for real-time, single-channel speech separation. *CoRR*, abs/1711.00541, 2017.

[MMA04] MMA (The MIDI Manufacturers Association), Los Angeles, CA. *The Complete MIDI 1.0 Detailed Specification*, 2004.

[OTKN12] Noriko Otani, Katsutoshi Tadokoro, Satoshi Kurihara, and Masayuki Numao. Generation of Chord Progression Using Harmony Search Algorithm for a Constructive

Adaptive User Interface. In *PRICAI 2012: Trends in Artificial Intelligence*, Berlin, Heidelberg, 2012. Springer.

[Rit06] Graeme Ritchie. The transformational creativity hypothesis. *New Generation Computing*, 24:241–266, September 2006.

[sub20] subprotocol. Advanced genetic and evolutionary algorithm library written in Javascript. `https://github.com/subprotocol/genetic-js`, 2020. Accessed: 2020-07-31.

[tea20] LilyPond team. Music engraving program. `https://lilypond.org/index.html`, 2020. Accessed: 2020-07-31.

[uc20a] MDN contributors. The Javascript Concurrency Model. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`, 2020. Accessed: 2020-07-20.

[uc20b] MDN contributors. The Javascript Web Worker API. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`, 2020. Accessed: 2020-07-20.

[Wee16] Joseph Weel. RoboMozart: Generating music using LSTM networks trained per-tick on a MIDI collection with short music segments as output. Master's thesis, Faculty of Science, University of Amsterdam, June 2016.

[ZKG05] Yongwei Zhu, Mohan Kankanhalli, and Sheng Gao. Music Key Detection for Musical Audio. In *MultiMedia Modeling Conference*, pages 30– 37, 02 2005.