UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Percussion Graphs : An Automated Approach for Percussion Composition

Louis Philippe Simões Castelo Branco Lopes

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Interacção e Conhecimento

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Percussion Graphs : An Automated Approach for Percussion Composition

Louis Philippe Simões Castelo Branco Lopes

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Paulo Jorge Vaz Cunha Dias Urbano

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Interacção e Conhecimento

2011

Acknowledgments

It has been a long road since I was freshman just entering college for the first time, and learning the proverbial ropes around campus (which I came to know like the back of my hand, after all these years). I know that I would have never accomplished any of this without the help of all my family and friends who molded me into the person I am today and in a way, I write this as a great big thank you for their continued influence on my life.

I would firstly like to thank my family, my mother and my father who have always watched over me and guided me my whole life, and of course my brother and his beautiful wife, who have always been there whenever I needed them.

I would also like to thank all my friends and coworkers here at LabMAg who helped me throughout my college career and thesis. I'd like to particularly thank André Bastos, João Costa, Geraldo Nascimento, Chrisitan Marques, Davide Nunes, Bruno Correia, Marco Lourenço, Gonçalo Cruchinho, Henrique Morais, Rui Flores, João Lobo, Frederico Miranda, João Lopes, Carlos Álvares, Carlos Teixeira, Diogo Serrano, Dinis Premji and Joaquim Tiago Reis.

I'd also like to thank my music teacher Emanuel Sousa who taught me everything I know and love about the guitar, music and musical composition. I'd also like to thank Sara Salazar, Francisco Gonçalves, André Batista and David Oliveira, who have given me some musical sanity along all my years playing the electric guitar.

I also couldn't forget my friends who have been with me since before I even dreamed of college life and most likely if it weren't for them I wouldn't even be here today, particularly Marlene Martins, Fabio Reis and my 10^{o} to 12^{o} grade teacher Paulo Gonçalves, who was more than a teacher but a great friend to all of us in our old computer science group.

Finally I would also like to thank my thesis coordinator and professor Paulo Urbano, who gave me the liberty to explore the theme of this work and was not only a great mentor but a great friend.

To all my Family and Friends.

Resumo

A composição musical é um processo artístico interessante, que envolve não só criatividade mas também experimentação. Neste trabalho propomos um sistema que integra vários conceitos de sistemas de música generativa, com um particular foco na integração do utilizador.

Ao contrário de outros sistemas de música generativa, principalmente os sistemas de música genética, ao qual o único papel que o utilizador tem sobre as músicas é de avaliador (avaliar a músicalidade de cada ritmo gerado), o nosso sistema irá permitir ao utilizador fazer parte do processo criativo e criar os seus próprios ritmos, que serão preponderantes depois no processo generativo musical.

Os principais objectivos deste trabalho foram:

- Sistema em que o utilizador seria mais do que um avaliador ou juiz rítmico e fazer parte do processo criativo músical;
- Fornecer ao utilizador múltiplas formas de gerar a música;
- A música gerada não deverá ser uma cacofonia de sons, mas apresentar alguma estrutura músical.

Introduzimos o conceito de ”Grafos de Percussão”, que é uma sistema de composição híbrido, misturando elementos interactivos (dependem do utilizador) com elementos mais automáticos ou generativos. Um grafo percursivo não é mais do que um grafo dirigido ao qual se associam, tanto aos nós como às conexões, ritmos percursivos. Cada nó está associado a um ritmo criado por um utilizador, enquanto cada conexão, representa uma sequência de ritmos. A ideia do grafo de percussão era criar um processo de composição intuitivo no qual o utilizador pudesse criar vários ritmos e interligá-los. A cada nó o utilizador irá associar um ritmo percursivo, enquanto os ritmos musicais associados ás conexões irão ser fruto de um processo generativo.

Os ritmos dos nós são criados pelo utilizador através de uma caixa de ritmos, que consiste numa grelha de 16 linhas por 16 colunas. As linhas representam os vários instrumentos de percussão disponíveis enquanto as colunas representam todos os tempos

disponíveis, neste caso um tempo é uma semicolcheia. Quando um ritmo é criado o utilizador terá que apenas transferir o ritmo para a aplicação ao qual este depois transformará num nó do grafo.

A cada conexão vai corresponder uma sequência de ritmos percursivos que representa a progressão musical entre o ritmo do seu nó inicial até ao ritmo do seu nó final. Este ritmo progressivo corresponde ao rasto do processo de optimização, que é formado pela sequência de ritmos obtidas em cada iteração do processo de optimização. Implementamos três técnicas de optimização para gerar os ritmos das conexões, que foram: Algoritmo Genético, Trepa Colinas e o Trepa Colinas Estocástico. Estes foram escolhidos pela capacidade de criar diversidade tanto a tamanho e natureza das sequências dos ritmos gerados.

Todos os optimizadores utilizaram a mesma função objectivo que é uma medida de semelhança de qualquer ritmo objectivo (ritmo final). Nomeadamente os instrumentos que são comuns aos dois ritmos, os tempos comuns (o tempo a que uma nota é tocada) e notas que são exactamente iguais tanto em instrumento como tempo. A função objectivo consoante estas características irá dar uma classificação entre 1 a 10, 10 sendo exactamente semelhante com o ritmo final. Logo como é de se esperar, os algoritmos generativos têm como função progredir os ritmos através desta avaliação rítmica. No entanto, o objectivo não é chegar ao ritmo final o mais eficientemente possível, mas de forma a que este seja capaz de criar ritmos intermédios que subjectivamente são ”valiosos”para o utilizador.

Pode existir várias situações em que uma sequência rítmica poderá ser demasiado longa ou repetitiva. Para combater esta situação introduzimos o conceito de filtros que permitem ao utilizador editar as várias conexões geradas, como por exemplo cortar uma parte rítmica da conexão (O Filtro Corte entre dois Pontos), eliminar todos os ritmos que têm um valor de avaliação menor do que X (O Filtro por Classificação), eliminar repetições (O Filtro por Repetição).

O processo de composição musical não se resume à geração do grafo percursivo. O grafo contém todos elementos da composição musical, mas a própria musica será o resultado de uma travessia do grafo. Conceptualizámos dois tipos de travessia, definindo dois nós como sendo dois extremos de um caminho (inicio e fim) ou então fazer uma travessia que passa por todos os ritmos uma única vez, problema análoga ao celebre caixeiro viajante.

Se o utilizador decidir definir dois extremos de um caminho, um processo que chamámos de ”Path Finding”, a aplicação retornará uma música (lista de ritmos) com todas as conexões intermédias que satisfazem um caminho entre esses dois extremos (caso seja possível). O utilizador terá à sua disposição 3 tipos de algoritmos de ”Path Finding”: Procura em Largura (Breath-first Search), Procura em Profundidade (Depth-first Search) e a Procura Aleatória (Random Search). Dependendo do grafo estas 3 opções podem

retornar resultados completamente diferentes, o que ajuda sempre na variedade musical. Neste tipo de travessias os nós podem ser repetidos.

Caso o utilizador escolher a opção do caixeiro viajante, o processo irá primeiro de tudo gerar conexões novas para criar um grafo totalmente conexo. Isto é necessário para que o algoritmo seja possível, e para não obrigar o utilizador a ter que interligar todos os nós manualmente. Quando o processo de criação do grafo terminar, calculamos um circuito usando um método de pesquisa local, o 2-Opt, utilizado com grande sucesso no problema clássico do caixeiro viajante.

Para que o utilizador seja capaz de utilizar todas estas características foi necessário criar um interface, onde o utilizador pudesse ter fácil acesso a todas a opções e características do programa. Esta interface permite:

- Criar os nós, manipulá-los, interliga-los com conexões;
- Ouvir conexões e nós;
- Escolher o algoritmo generativo de optimização para preencher as conexões;
- Gerar a lista de ritmos para as conexões existentes;
- Aplicar filtros nas conexões;
- Escolher o tipo de travessia;
- Definir os extremos para a travessia ponto a ponto e gerar o resultado;
- Obter a lista de ritmos do algoritmo caixeiro viajante e ouvir o resultado;
- Ouvir as Travessias.

Todos estes conceitos serão explicados e detalhados neste trabalho, incluindo detalhes de implementação, a investigação e experimentação do conceito de grafo de percussão e um manual de utilização do sistema. Este trabalho também incluí uma secção de discussão e uma apreciação musical dos resultados obtidos, como a musicalidade, tempo de geração e análise dos resultados do path finding. O sistema também foi testado com um grupo de utilizadores (músicos e não músicos) ao qual este trabalho também descreverá as experiências dos utilizadores e a sua apreciação musical.

Palavras-chave: Música, Percussão, Grafo, Generativo, Automação

Abstract

Musical composition is an interesting artistic process, which can involve a lot of experimentation with various known musical concepts. In this work we propose a system that tries to integrate various aspects of music generation and automation systems, but with a focus on integrating the user into the music creation process. Unlike systems who cast users into a musical arbiter role, our system generates music in accordance to user created music.

We named this system Percussion Graphs, and consists of a musical graph representation of user created rhythms and generated rhythms. A graph contains two main components: The Node component (which are User Created Rhythms) and a Connection component (which is a List of Generated Rhythms). The idea is to have users create rhythms, that can then be connected with each other. These connections are the rhythm generation process, and they consist of a slow progressive transformation process of one rhythm into another, in this case two user created rhythms.

A user will be able to control multiple aspects of the percussion graph such as node manipulation and creation, node connection, choosing the rhythm generating methodology from multiple available methodologies, traveling the percussion graph and hearing the generated solutions. We also implemented editing options, called filters, that will let the user trim and edit connections to his or her liking.

In this work we will explain on how the percussion graph and all of its features were implemented and how they work (internally and usability wise). We also talk about on what led us to follow this specific modus operandi. We also discuss the experiences of several users during their test run with the percussion graph system, and their thoughts on the generated rhythms and the system itself.

Keywords: Music, Percussion, Graph, Generative, Automation

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Considering that this was a research project, we believe it was in our best interests to explain on how we got to the idea of the percussion graph. The system wasn't born out of thin air and involved a lot of experimenting, testing and some creativity.

In the beginning of our research we started out by applying various ideas of Horowitz's [Hor95] evolving genetic drum patterns. Horowitz's idea was to use genetic algorithms to generate rhythms who would then be ranked by users, however we soon realized that this was somewhat of a tedious task and a more automated approach would be much more ideal.

The work of Yee-King[YK07] whose solution to the fitness evaluation (even though not the main focus of his work) was quite ingenious. The main idea was using a piece of music which was originally created by the user, which would then serve as the fitness for the evolutionary process (which in Yee-King's case was through Genetic Evolution specifically). As the song evolved (from a randomly created individual), it would try to be as closely similar to what the user had initially created, with each new generation. Meaning that during the later generations of the evolutionary process a musical structure would start to take form, as it would start to gain influences from the user's original creation. So in theory, if we could "recycle" all of the solutions obtained through the evolutionary process (especially the latter solutions) one could create variations upon the original piece or even literally create a song using these solutions.

Figure 1.1 shows how the initial stages of our prototype functioned. We initially started exclusively using the genetic algorithm model, which at the time was an extension of our Horowitz drum machine prototype, and extensively tested it for it's musicality. The system showed a lot of potential and was something from which we could build upon.

During one of our brainstorming sessions one particular idea started to take shape, which was to create a system that could read multiple user created rhythms and instead of starting the evolutionary process from a randomly generated music piece, it would start from another user created rhythm. What we had hoped to achieve with this, was create musical pieces that could change and slowly evolve from one rhythm to another. We

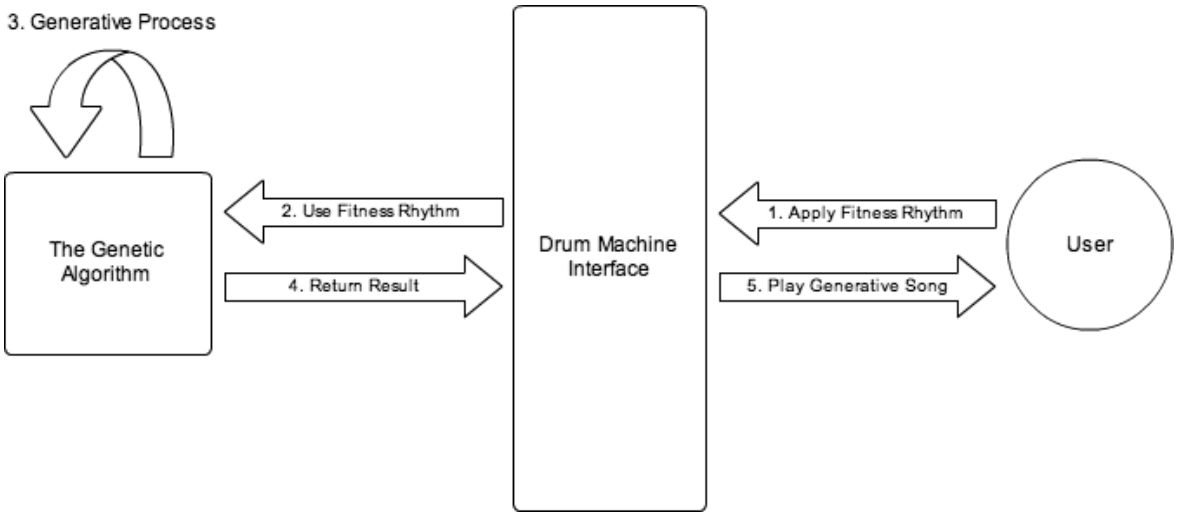

Figure 1.1: The First Iteration of the System

quickly created another prototype, one that would be capable of reading two user rhythms and generate a passage between rhythm 1 and rhythm 2. The results were quite interesting and prompt us to follow this idea even further.

Everything finally came to fruition once we tried to symbolize the previous process graphically. The idea of Nodes representing user created rhythms and their connections representing the generation process gave birth to the notion of percussion graph. A system that would allow the user to organize his rhythms and manage the generation processes, through a graphical interface. The user would also be able to listen to multiple connections by traveling through the percussion graph and listening to graphical paths.

The conceptualization of the fitness function was also an important aspect of the early research process as it would be the arbiter of how a song would develop throughout the various generative iterations. Various function ideas were explored through experimentation with NetLogo[Wil][TW04] which ranged from ambiguous to more defined comparison solutions. The final solution ended up being an extension of one of the fitness functions used in the NetLogo experiments.

As the research progressed, the concept of restricting the system to only an Evolutionary Model was a bit narrow minded. The real objective was to create automated music regardless of the methodology in the back system. Besides it would also add another layer of researchable value, comparing other system types to the evolutionary model. With that in mind we decided to end the fitness terminology when referring towards this evaluation process, as it just didn't make sense anymore. Objective Function was a much more appropriate name[LS11] as it symbolized the process relatively well.

The Percussion Graph system, which we will extensively explore and discuss all of its components in this thesis, is the fruit that came from this researching phase. The application can be seen in action at the LabMAg website (`http://bookmark.labmag.`

di.fc.ul.pt/?page_id=1966).

1.1 Motivation

One of the main reasons that went into developing this system, was to have an opportunity into combining two of our greatest passions, music and technology. The idea of creating a system that can create music, which to most is considered a form of art and by definition (according to Webster's Dictionary) art is the expression of human skill and imagination, can be thought as ludicrous. Our goal was never to erase the musician from the music creation process, but to help the musician into finding new forms of musicality and create tools that would allow him to express himself and also have the capability to even assist him in the music making process.

Considering the number of interactive applications that exist, primarily interactive games, its not hard to imagine a system that could use a musicians musical created content in ways that would change every time a user would play a specific level. This would not only grant a fresh new experience every time the user would re-play a specific sequence but also make that experience unique every time, adding a new level of value to the overall experience.

But also just the prospect of investigating a machine's own ability of creating art, something that is usually reserved to science fiction novels, is a spectacularly fascinating concept. In which to itself is worth of investigation, even if not out of curiosity.

1.2 Objectives

One of the main objectives of this work was to create an automated musical system, one that could create coherent percussion rhythms that would appeal to users and even help them in the music creation process. We also wanted to create a system where a user could focus his main attention into the creative aspects of music creation and not the monotonous aspects, such as option tweaking or even a rhythm rating system that we had with Horowitz's model.

We also wanted to make sure that the created system felt complete, with enough features that would make it compelling and interesting. Such as multiple generative algorithms, methods that would allow the user to travel the graph and listen to the generated content, methods that would allow the manipulation of the graph and a rhythm creation tool.

1.3 Contributions

The Percussion Graph system is an application that considers user created rhythms to create computerized automated music. The system is still pretty much in its primordial state and would be a great asset for anyone willing to work on it. The Percussion Graph's main features include:

- The objective function and rhythm ranking system
- The various optimization process algorithms
- The path finding algorithms
- The traveling salesman algorithm with 2-Opt
- Rhythm / Connection Filters
- The Percussion Graph Interface (Node Creation, Connection Creation, etc.)

Also it should be noted, until reaching the development of the percussion graph system, we created various prototypes from numerous ideas that we had conceptualized during various brainstorming sessions, although due to time constraints we were never able to really explore these ideas further, apart from creating a simple prototype application. Such prototypes included, Horowitz's Drum Machine, The BeatBox Sketcher (Image Sonification Application) and The Evo BeatBox (Automated Genetic Algorithm Drum Machine) from which two versions were created. All of these prototypes were built using NetLogo and are open to anyone who wishes to work on these ideas and further them.

1.4 Document Structure

This document is organized as follows:

- Chapter 2 - The State of the Art - In this chapter we will review some of the previous works accomplished in the field of generative and automated music systems.
- Chapter 3 - Percussion Graphs - In this chapter we will discuss and explain all of the components of a percussion graph and its editable features.
- Chapter 4 - Traveling the Percussion Graph - Chapter 4 explains how a user will be able to hear the generated rhythms by traveling a percussion graph, by applying a traveling method.
- Chapter 5 - The Percussion Graph Application - This chapter will explain the user interface and application, and how to use it.

- Chapter 6 - Experiments & Results - Chapter 6 details our personal musical appreciation of the system and various experiments and discussions that we conducted with multiple users.
- Chapter 7 - Conclusions & Future Work - Finally Chapter 7, consists of our final discussion of the application and a post-mortem view of the entire research project. We also discuss some of the future work of the system.

Chapter 2

State of the Art

The evolutionary and generative art world isn't a relatively new concept, and already has a great thriving community, who are passionate about art and its integration within an artificial computational mold. The solution to this problem isn't a one-way street, with the community presenting various creative solutions to the problem including swarm types, genetic types, multi-agent types, neural networking types, cellular automata types and hybrids (a system that uses multiple types).

The cellular automata model has been thoroughly used in generative music systems[BE05], of particular note is the work of Miranda, E.R. [Mir01] [Mir03] who used cellular automata for music composition and explored its usefulness for a musician or musical composer. Burraston, D. et al [BELM04] also explored the possibilities of applying the Game of Life into a MIDI domain, while Brown, A. [Bro05] explored the ability of composing and creating rhythms with coherent monophonic passages.

Gueret, C. et al.[GMS04] proposed a swarm system of musical ants who would travel a graph whose vertices would consist of notes and edges of the passages between notes, this work was even further developed by restricting the ants to the compositional style of baroque[GM07]. Although, swarm systems have also been used for music improvisation [Bla03] and not just music composition.

Neural Networks have proven to be good for fitness evaluation[JP98] methods and specifically identifying musical styles[MVW$^+$03], however Mozer, M.C. [Moz94] proposed a music composition neural network system that would compose music through the assimilation of various musical styles and their respective tendencies.

For the Multi-Agent types Eigenfeldt, A., proposed a system [Eig10][EP09] where agents would work together using negotiation tactics and compromise to form a singular music objective. Gimenes, M. et al. [GMJ05] proposed another type of multi-agent system using the meme concept [Daw76], where each agent would learn specific rhythms and spread them across other agents.

Considering though that our research consisted of a generative percussion system using genetic algorithms, we decided that we could explain some of the more influential

projects that influenced our own research, in more depth.

2.1 Horowitz's Genetic Algorithm Rhythms

Horowitz's system[Hor95], was one of the main influences of this research, where we even recreated the system for one of our prototypes (See Appendix A). The main objective of the system was to generate automated rhythms through a genetic algorithmic process. User's would then influence this genetic process by evaluating each generated rhythm with a "like" or "dislike".

Even though the system is using an interactive genetic algorithm approach[Smi91], the subjectivity and variability of a user's criteria can sometimes be ambiguous. For that matter, user evaluation wasn't the only determining factor for the fitness function. Objective functions, containing various rhythmic rules (such as density, beat repetition, etc.) would also influence and steer the evolutionary process into a more specific direction.

2.2 CONGA

CONGA[TI00] is another type of evolutionary music composition system developed by Tokui & Ida. However, unlike Horowitz's system, the CONGA system relies on both the Genetic Algorithm and Genetic Programming[Koz96] concepts for generating new rhythmic solutions, which had been previously proven successful[BV99][Bil94][JP98].

The CONGA system was designed taking into consideration three main aspects, the search domain, the genetic representation and the fitness evaluation. The search domain consisted of 4 to 16 measure rhythmic patterns (sequence of notes). The genetic representation combined the genetic algorithm (GA) and genetic programming (GP) approaches, where GA individuals represented short pieces of rhythmic patterns, while the GP expressed how these patterns were arranged in the musical structure. The Fitness Function consisted of a user evaluation approach, accompanied by a evaluation assistant module.

In a way the CONGA system can be thought as the next step of Horowitz's Design[Bil05] as it takes the musical creation even further by giving it a musical structure and not simply creating single standalone rhythmic patterns.

The evaluation assistant module is also an interesting approach on the user subjectiveness of the fitness criteria. Using a neural network, this module will reduce the GA population in accordance to a human subjective function and display only the fittest individuals to the user, lessening the burden by making the user rate a smaller population. The assistant will also take into consideration the GP individuals, and shorten the musical structure if the length gets subsequently to large.

2.3 The Evolving Drum Machine

Yee-King's Evolving Drum Machine[YK07], is a good example of how the evolutionary and generative process can be taken into consideration for the music creation process. In this system the user can simply input a target sound (or rhythm), which serve's as the rhythmic objective or the goal of the evolutionary process.

Each GA Individual will try to converge towards this rhythm, by comparing itself with the target rhythm. The more similar the GA individual is to the target rhythm, the higher its fitness value will be. However, the best rhythms (highest fit) of each evolutionary generation, that were generated since the beginning and the end (when a generated rhythm converges into the target rhythm) of the evolutionary process, will be the final musical solution.

Taking into account Yee-King's fitness evaluation, Horowitz's and Tokui's genetic model design, it is clearly obvious that these were the main inspirations that lead to the creation of this research.

2.4 Beatrix - The Amorphous Drum Ensemble

Beatrix[BS02] is a system that requires little to no user interference, and has the ability to be fully autonomous. Options such as adding personalized beats, voices and tempo changes, is something that is purely optional and not enforced.

The basis of the project was to model an African Polyrhythm[Lad] drum ensemble, where each drummer is distributed and autonomous and in which each drummer must coordinate their rhythm and timing amongst themselves.

Each drummer has a communication radius, for example, a beat produced by a drummer is broadcast to all of its neighbors (these being, drummers within the drummer's radius) much alike how sound travels through air. What this means is that drummers must physically communicate with each other in order to achieve a musical synchronicity. To add to this simulation, drummers also have a small percentage of not hearing what was played or communicated, by another drummer.

The rhythmic patterns are created by applying various genetic operators, although they are grounded to plausible percussion techniques, disallowing random and un-plausible rhythms. Beatrix contains a set of basis percussion patterns, which can then be combined and mixed creating various rhythmic solutions. These created patterns can then be mutated (dropping or adding a note) or rotated (shifting notes in the timeline), adding more rhythm variety.

2.5 GenJam

GenJam by Biles, John A.[Bil94] , is another type (and well known) of evolutionary system in which its objective was to model a novice jazz musician, who is learning how to improvise and perform live shows.

The GenJam works by using two types of populations one for measures and one for phrases as a way to build a solo. An individual in the measure population maps to sequences of MIDI events, while an individual in the phrase population maps to indexes of measures in the measure population. This makes it so that there is not just one single best measure or phrase. Achieving the perfect solo wasn't really the point of this work, but more of a way where the GenJam can apply various melodic ideas to any tune.

To improvise on a piece of music, GenJam first reads a progression file that contains information such as tempo and rhythmic style, the number of solo choruses it should take, and the chord progression. The system can also read MIDI sequences of piano, bass and drums (which were pre-generated). The GenJam will then improvise on this piece by building choruses of MIDI events obtained from members of the measure and phrase populations.

While listening to a solo, the user can then evaluate each musical portion with a binary operation, of g (good) or b (bad). Fitness is then determined by a counter, existent in each measure or phrase, that will increase or decrease depending on how many times the user types in a g or b, respectively.

Another interesting point of this work is that the GenJam feature 3 mode types: learning, breeding and demo. The learning mode consists of building the fitness values of the system without applying any genetic operators, where phrases are selected at random, ignoring fitness, and presented for feedback. Demo mode is intended to be a performance mode, where each phrase is selected in a tournament like fashion, where phrase fitness and their respective measure fitnesses are taken into account. The breeding mode is where the genetic operators come into play, in this mode half of the population will be replaced by new offspring and will await feedback by the user.

2.6 Coming Together

Another interesting system that applies a generative process to music making, is Coming Together by Eigenfeldt, A.[Eig10], which is a fully autonomous music creation system that creates music through cooperation and negotiation between virtual agents with predefined goals.

Each individual's desire is to develop a musically meaningful relationship with all other agents. What this mean is that an agent will try and generate a harmonically, melodically and rhythmically sound phrase, as if it had been composed by a human composer.

The idea is to have agents achieve a musical convergence by communicating and altering their musical output based on their beliefs.

The end result is then a negotiated solution between all of the agents, where a final consensus has been achieved. Although convergence is not always successful in this system, it does acknowledge this and restarts itself if a current progression isn't going favorably.

Chapter 3

Percussion Graphs

In this chapter we will introduce the concept of a percussion graph, and explain all of the ingredients that make this system come to life.

A percussion graph is a directed graph where we associate a percussion rhythm to each node and an ordered sequence of percussion rhythms to each edge. The definition of a percussion graph is very abstract and in principle an edge can be associated with any kind of sequence of percussion rhythms.

The percussion graph was a natural abstraction of our original idea, which was of users creating their own rhythms through a drum machine component and connecting these rhythms with each other. An automatic process would then simply ”fill in the gaps”, that is, it would generate a sequence of percussion rhythms for each directed edge. We thought it could be musically interesting to interpret the percussion graph edges as progressive transformations of musical rhythms (their origin nodes) into other rhythms (their end nodes).

The main goal of this research was to explore musical generative processes, which fitted in perfectly with our percussion graph system, allowing us the possibility of associating a rhythmic sequence through a graph's edge. Using a set of optimization processes we would autonomously create sequences of percussion rhythms and associate them with the multiple directed edges of the graph, given their respective initial and end nodes' rhythms. This progressive transformation of a rhythm can be the historical trace of an optimization process where we start from a rhythm and change it step by step trying to obtain the goal rhythm linked to the edge end node, in the smallest number of small transformation steps.

For this we applied and analyzed three different optimization algorithms: genetic algorithms, hill-climbing and stochastic hill-climbing. We could have explored more optimization options, but in regards of musical diversity, such as the length and the nature of the generated percussion sequences, we found that these 3 were enough.

We also found that some editing tools would prove useful, case a generated sequence would be too long or have a lot of repeating rhythms. This would allow the percussion

graph user to trim and form the rhythmic sequence into something more to his or her liking. We called these tools filter functions and are also described at the end of this chapter.

3.1 A Percussion Graph

A percussion graph is a directed graph where each node is associated with a percussion rhythm and each edge is associated with an ordered sequence of percussion rhythms. Figure 3.1 depicts a percussion graph composed of 4 nodes and 5 directed edges. Assume that each node has its own percussion beat and each edge has its own sequence of percussion beats - the musical components were just not represented in Figure 3.1 due to space constraints. The edges have costs that correspond to the number of percussion rhythms in the respective sequences.

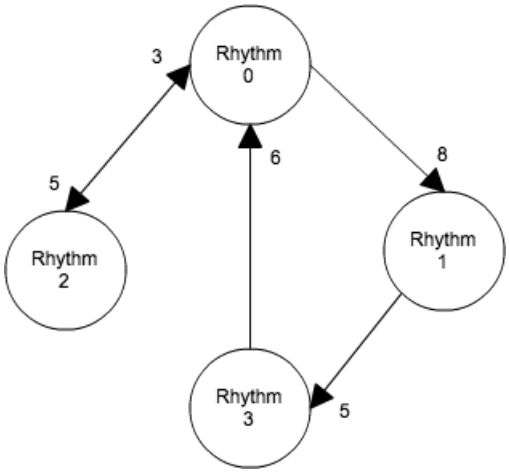

Figure 3.1: Percussion Graph Concept

3.1.1 The Node

The Node in the context of this work represents a rhythm (or a drum pattern). The idea behind this was to have a graphical representation of a rhythm within our system, that would help users grasp the percussion graph idea visually and not just conceptually.

A user will be able to input a rhythm through the aid of a Drum Machine, which is a small application that is capable of playing and creating rhythms, which can then be converted into nodes. The Drum Machine's musical timeline is represented by a vertical line that travels from left to right on the screen, when the line comes into contact with a note, that note is played. It should be noted however that the tempo (which is the speed of the musical timeline) does not have any influence on the rhythm itself, and is a direct specification of the Drum Machine component.

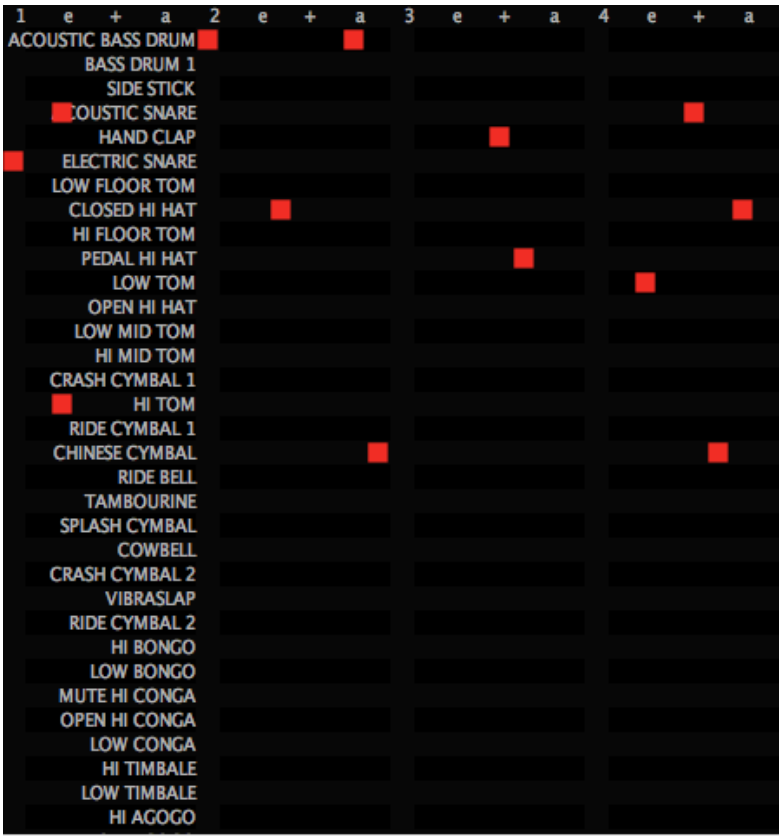

Figure 3.2: The NetLogo Drum Machine

For each node, the rhythm is represented by a note-map or beat-map (see Figure 3.3), which is a list of all the notes in the drum machine. Each note consists of a triple that are the representations of the Instrument, the Time and the Velocity.

- **The Instrument** to be played, is represented by an Integer that corresponds to a specific instrument code.
- **The Time** which an instrument is played, is represented by an Integer that corresponds to a specific play time code.
- **The Velocity** that an instrument is played, is represented by an Integer that varies between 0 and 100. For the purpose of this experiment velocity only has two states, played (value is equal to 100) or not played (value is equal to 0).

So in conclusion what should be retained when discussing the Node is that it represents a percussion rhythm, which internally is a note-map. A note-map is a drum pattern, like the one represented in Figure 3.2, and contains the information of the possible states (playing and non playing) of all the notes that the drum machine component is capable of offering.

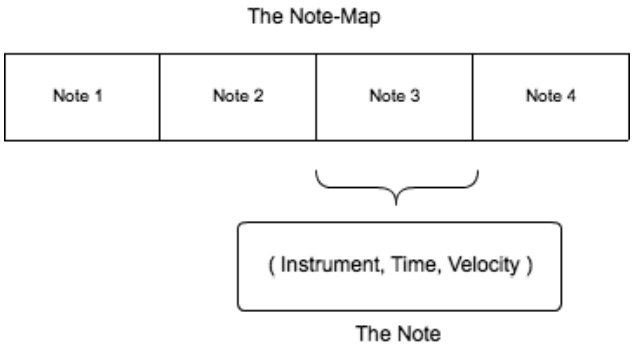

Figure 3.3: The Note-Map is the system's representation of a rhythm

3.1.2 The Connection - The Directed Edge

The Connection is a sequence of percussion rhythms. This sequence can be represented as a list of note-maps (see Figure 3.4) and will contain various rhythms which will be played in order of input.

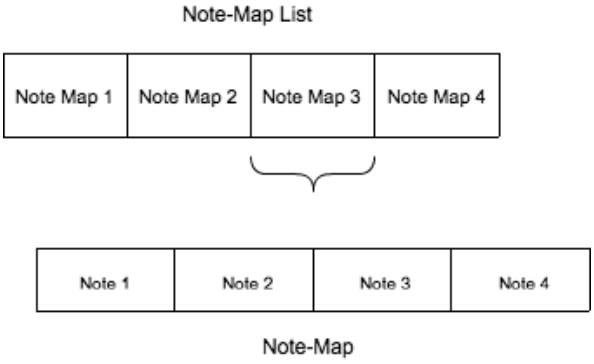

Figure 3.4: A Note-Map Sequence

A connection will start out empty initially, and will need to be filled with note-maps by an auxiliary function, which in the case of this research is done through our optimization processes.

3.2 Automated Connection - Generating Rhythms for the Directed Edge

In a percussion graph there are no restrictions regarding the association between the graph components and percussion rhythms: a node can be associated with any kind of percussion rhythm and an edge can be associated with any possible sequence of percussion rhythms.

For this research we envisioned a particular type of percussion graph, where each edge would be interpreted as a progressive transformation between two rhythms. What

this means is, if there is a directed edge between two nodes, it symbolizes the progressive transformation of the edge's origin node into the edge's end node. Which of course is something that could be done manually, but for the purpose of this work would be pointless. Instead, given the nodes' respective rhythm's and also the edges associated with them, the edges musical elements can be automatically generated through an optimization process, where the sequence of percussion rhythms is the trace left by these optimization processes, which represent the progressive rhythm transformation from the edge start node until reaching the edge end node.

In order to guide the optimization process we will have to define an objective function that will rank a rhythm in terms of how similar it is relatively to another rhythm (which is our goal). The objective function will always be the same for each optimization process.

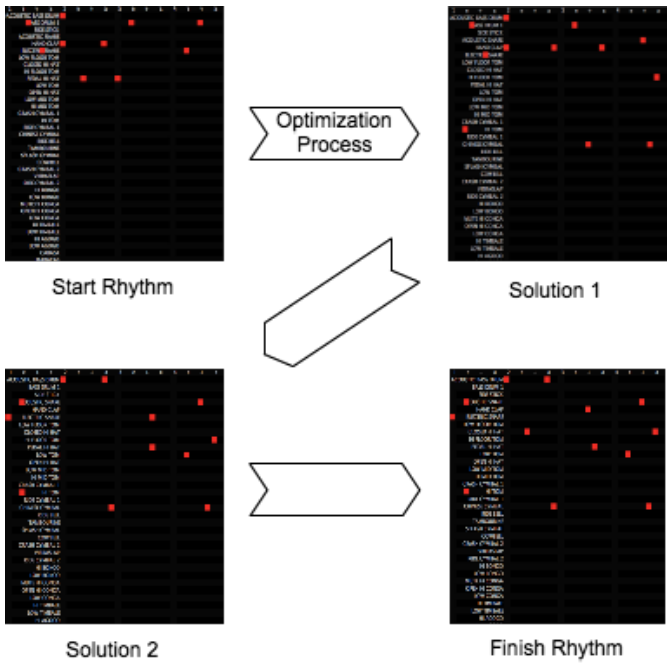

Figure 3.5: The Optimization Process

We also needed to define a way to stop the algorithm and we came up with two methods:

- The Natural Stop Condition: Stops the optimization process once the objective function gives any generated rhythm a value of X or more, X being defined by the user. This gives the user liberty to stop the process when a rhythm's similarity is sufficiently near the goal rhythm, and does not necessarily have to be a full exact equivalent.

- Sequence Length Stop Condition: Stops the optimization process based on the size of the sequence. This is useful in case an optimization process stalls or, is simply

taking to long to reach the Natural Stop Condition. In a way, this stop condition is a fail safe, in case the first stop condition is never achieved.

The sequence of rhythms which are obtained through the optimization processes will also be independent of which algorithm was used. We call these sequences Objective Maps, which are the consecutive solutions that are returned by our optimization processes. What this means is that, during the optimization process, an algorithm will progressively return solutions which will be kept by our objective map. These solutions will heavily vary depending on which algorithm was chosen, what the starting and ending rhythm was and a luck factor (or generative randomness).

Figure 3.5 shows a typical optimization process and its iterative solutions, note that for this particular example each iteration, the generated rhythms get relatively closer in similarity to the objective rhythm.

3.3 The Objective Function: Measuring the Rhythms Similarity

The main role of the Objective Function is to evaluate any rhythm relatively to an objective rhythm.

For this we need to determine the similarity between one rhythm relatively to another rhythm (rhythms are note-maps). The Objective Function is described in Figure 3.6.

$$f_{obj}(gen) = Similar(gen, obj)$$

Figure 3.6: The Objective Function

3.3.1 Measuring Rhythm Similarity

We think that they are two main component aspects for measuring similarity between any two rhythms. The first one being the number of active notes between the two note-maps, independently of the nature of the active notes. The second aspect being the similarity between the active notes of both rhythms. For that we'll take into account the number of coincident active notes, however we also take into account that it is important to keep track of notes that are only coincident in time or instrument.

The Similarity Function (Figure 3.7) is the weighted sum of the length similarity and the position similarity, thus giving the liberty to change their relative importance.

The Similarity function value is a function that ranges from 1 to 10 (10 being that two rhythms are exactly the same) and their relative importance of the Similarity by length and

$$Similar(r1, r2) = \frac{Sim_l(r1, r2) \times Lw}{Lw + NOw} + \frac{Sim_n(r1, r2) \times NOw}{Lw + NOw}$$

Figure 3.7: The Similar Function

position is given respectively by Lw and NOw. We experimented with these similarity weights and the best results obtained was simply giving the length a slightly lower weight (40-60 in favor of position) then the position weight, these values are the default weights.

Length Similarity

The term length refers to the number of notes that are played within a single note-map, for example if a rhythm has X playing notes the length will be X. The length similarity between any two rhythms will only depend on their lengths absolute difference as depicted in Figure 3.8. The farther away their lengths the lower the score.

$$Sim_l(r1, r2) = 10 \times \frac{1}{(|(length(r2) - length(r1)| + 1)}$$

Figure 3.8: The Length Similarity Function

The Length Similarity can be analyzed in Figure 3.9, and shows how rigorous this function can be: the score is immediately halved once the length difference is 1.

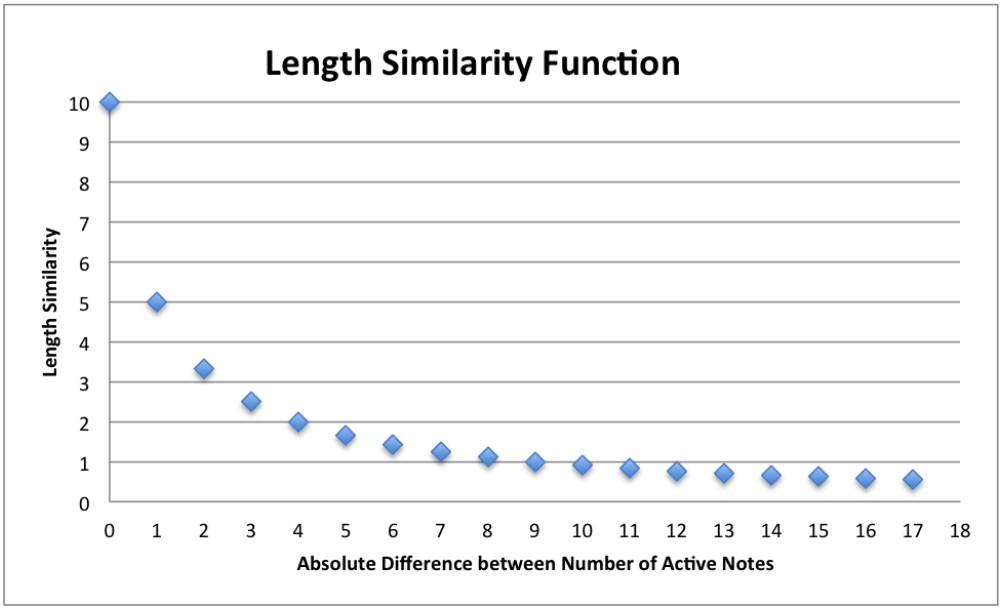

Figure 3.9: The Variation of the Length Similarity, given the absolute difference between rhythms' length

Notes' Similarity

The Notes' Similarity evaluation will compare the nature of the active notes between one rhythm (r1) and a target rhythm (r2), thus it is a measure of how similar Rhythm 1 is to Rhythm 2. As we stated earlier in section 3.3.1, this measure will consider three aspects: **Common Instruments**, **Common Times** and **Common Notes**.

- Common Instruments (**I**): The number of distinct instruments that are played in both rhythms. Note that, if Rhyhtm 1 has two active notes that play maracas and Rhythm 2 only has one active note with that instrument, that instrument will only be counted once.
- Common Times (**T**): The number of distinct times that are played in both rhythms. Repeated active notes with the same time, will be counted in the same fashion as in the Common Instruments.
- Common Notes (**T**): The number of active notes that appear in both Rhythms.x

Each of these aspects will correspond to a respective function: I(r1,r2), T(r1,r2) and N(r1,r2). The Similarity function will be normalization of the weighted sum of these three functions and again will be a value between 1 and 10, as depicted in Figure 3.10.

$$Sim_n(r1, r2) = 10 \times \frac{I(r1, r2) \times Iw + T(r1, r2) \times Tw + N(r1, r2) \times Nw}{numI(r2) \times Iw + numT(r2) \times Tw + length(r2) \times Nw}$$

Figure 3.10: The Notes' Similarity Evaluation

Its clear that the weight of the Common Note similarity should be significantly higher than the other two weights, and that both Common Instrument and Common Time should have an equivalent importance. By default we have chosen the value of 9 for the Common Note weight and the value of 1 for the Common Instrument and Common Time weights.

In order to normalize the function it was necessary to calculate the maximum possible value for the weighted sum (the numerator component of the formula). The maximum value is obtained when the first rhythm is exactly the same as the second rhythm. Note that we have to calculate the similarity of the first rhythm relatively to the second rhythm. Therefore, I(r1,r2) = numI(r2), T(r1,r2) = numT(r2) and N(r1,r2) = length(r2), where numI(r2) and numT(r2) corresponds respectively to the number of distinct Instruments and Times played in r2.

Illustration of the Similarity Calculation

We will illustrate the similarity function using as an example the two rhythms depicted in Figure 3.11.

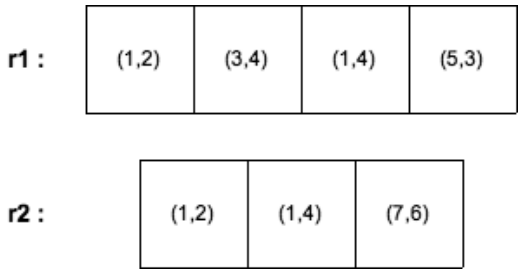

Figure 3.11: The Played Notes of Rhythm r1 and Rhythm r2

We will begin by calculating the length similarity, given that r1has four played notes and r2 has three played notes (length(r1) = 4 and length(r2) = 3), and following formula 3.8 we obtain the results depicted in Figure 3.12.

$$Sim_l(r1, r2) = 10 \times \frac{1}{(|(3-4|+1)} = 5$$

Figure 3.12: The result of Length Similarity between r1 & r2

After the length similarity calculation we pass onto the notes' similarity calculation. The intermediate calculations are:

- I(r1,r2) = 1, because there is only one common instrument between r1 & r2 (Instrument 1).
- T(r1,r2) = 2, because there are 2 common times between both rhythms (Time 2 & 4).
- N(r1,r2) = 2, because there are 2 notes ([1,2] & [1,4]) that belong to both rhythms.
- To calculate the normalizing factor, we need to determine the length (length(r2) = 3), the number of distinctive instruments (numI(r2) = 2) and the number of distinctive times (numT(r2) = 3).

Following formula 3.10 and given the default weights (Nw = 9 & Tw = Iw = 1) we obtain the results in Figure 3.13

$$Sim_n(r1, r2) = 10 \times \frac{1 \times 1 + 2 \times 1 + 2 \times 9}{2 \times 1 + 3 \times 1 + 3 \times 9} = 6.4$$

Figure 3.13: Notes' Similarity value of r1 relatively to r2

Finally, in order to calculate the global similarity of r1 relatively to r2, we will use the default values for the weight parameters(Lw = 0.4 & NOw = 0.6) and following formula 3.7 we obtain the result in Figure 3.14

$$Similar(r1, r2) = \frac{5 \times 0.4}{0.4 + 0.6} + \frac{6.4 \times 0.6}{0.4 + 0.6} = 5.84$$

Figure 3.14: The Similar Function result of r1 relatively to r2.

3.4 Optimization Processes

3.4.1 Genetic Evolution Process

The Genetic Evolution Algorithm is a search algorithm inspired by the natural process of evolution [Dar58]. The algorithm consists of creating populations of candidate solutions (called chromosomes) to the optimization problem and applying natural evolutionary tactics, such as crossovers, mutations and selections. The algorithm starts out by creating a population of randomized individuals who are then ranked by a fitness function. Depending on their rank the individual might be selected or not for the breeding phase. The breeding phase will create a new generation population, consequently re-iterating the ranking, selection and breeding phase on this new generation. This is done until the optimum solution is found.

The Genetic Evolution algorithm was the first optimization process to be developed for this research. According to Mitchell M. [Mit98] there are various ways to approach genetic algorithms (GA's), but the most common approach is by separating the algorithm in 3 phases:

- Fitness Application
- Selection
- Crossover and Mutation

The idea of the genetic model is to create a population of individuals who will then be ranked by a fitness function. Taking into consideration each individuals rank, a selection method is called. This method will pick a number of individuals (the individuals picked depend heavily on the selection type) that will go into the breeding phase. In the breeding phase two individuals will suffer a crossover and might have a chance of mutating, creating new individuals. This can be done Nth number of times from selection to breeding until reaching the population cap, whereas from there, each individual will be ranked again by the fitness function.

Implementation Specifics

Each Chromosome (or Individual) will be a note-map and the fitness will be our objective function (as defined in 3.6). The best ranked chromosome of each generation will be kept within the objective map.

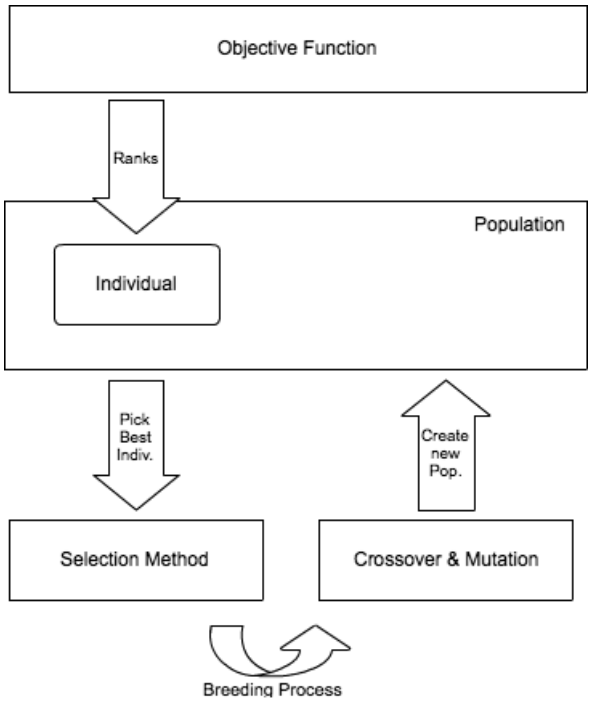

Figure 3.15: The Genetic Algorithm Evolution

We also needed a population to kickstart the genetic process. This process would need to take into consideration the starting node's rhythm, so we created a "special" mutation function, one that could mutate the starter rhythm, but still keep most of its elements intact. This starter mutation would then run N times (N being the population size), creating a population pool that could be used by the genetic algorithm.

Finally we needed to define the algorithms stop conditions. Because of the unpredictability of the natural stop condition, which by default is set to 10, we also defined a sequence length stop condition. By default the sequence length stop is set to 75 generations which means that, after 75 solutions were generated, and no solution has satisfied the natural stop condition, the algorithm stops and returns all the solutions that were obtained until generation 75.

When we were discussing the implementation of the genetic algorithm within our system, we pondered on the idea of implementing the genetic model ourselves, however a better solution was to just use an available working API. JGAP[Mef] included all of the options we needed to make this genetic model work, except a few minor exceptions.

Selection

Selection algorithms choose which chromosomes from a population are to breed for the next generation. If a particular chromosome is fitter than the other, doesn't necessarily mean it will be chosen. Depending on the algorithm the fitter chromosome might have a bigger or the same chance to breed than the less fit chromosome. But most of the time the

probability is always in favor of the fittest chromosome.

We ended picking the Roulette Selection method as it favors higher ranked chromosomes but not always. The Roulette Selection is a fitness-proportionate methodology[Mit98], where the fitness values of each chromosome influence the probability of selection. In the case of the roulette, each chromosome is assigned a slice of a circular ”roulette wheel”, where the size of the slice is proportional to the fitness that was given to each chromosome. The roulette is then spun N times (N being the number of the population) and the chromosome chosen by the ”wheel’s marker” is selected to breed.

1. Sum the total expected value of the individual’s fitness, and call this value T.
2. Repeat N times:
   (a) Choose a random integer r between 0 and T.
   (b) Loop through the individuals in the population, summing the expected values, until sum is greater than or equal to r. The individual whose expected value puts the sum over this limit is the one selected.

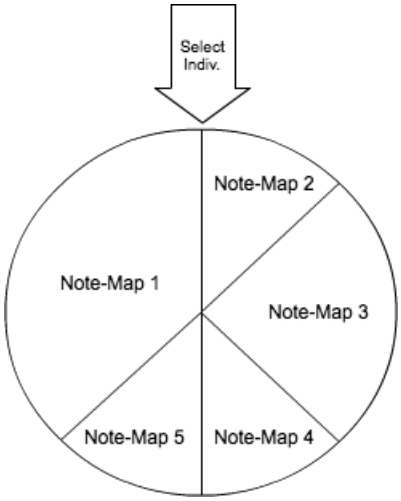

Figure 3.16: Representation of the Roulette Selection

Elitism

The Evolutionary process also uses elitism. Elitism forces the genetic algorithm to keep the best-ranked N number of chromosomes from one generation and automatically move them towards the next generation[Mit98]. This helps keep the population in-check and the chromosomes from worsening with each new generation. However Elitism has the side-effect of staling a population creating very similar chromosomes if left unchecked, so it is always important to keep the elitist population low compared to the whole.

For this research we decided to apply elitism to 25% of the population, which means that in a population of 500 chromosomes, the best 125 chromosomes would be picked for the next generation. However this should not be confused with the solution that is picked for the objective map, the solution is always one and only one chromosome, which is the best out of all the chromosomes of one generation (which in case of a tie is randomly picked).

During some experiments we did try to forgo elitism, however it proved to be difficult for the algorithm to converge towards it's goal, as it was always struggling with un-fit and fit solutions until stabilizing into a regular score after multiple generations. Musically, without elitism the sounds within the middle generations were always chaotic with new variations always occurring, however most of the time these variations were tame, with three to four notes shifting from one place to another.

Crossover

Once the individuals have been chosen, its time to apply the crossover. The crossover is a generic concept which consists of mixing the contents of two chromosomes as a means of generating two new chromosomes. Each chromosome (in this case the note-map) chosen for crossover is split in two, where the location of the cut is defined by a cutting point, subsequently each half is then joined with the other chromosome's halves. This will create two new individuals, each of them containing half of their parents genes. The cutting point of an individual's chromosome can be random or defined, in this case it is defined at the middle of a chromosome.

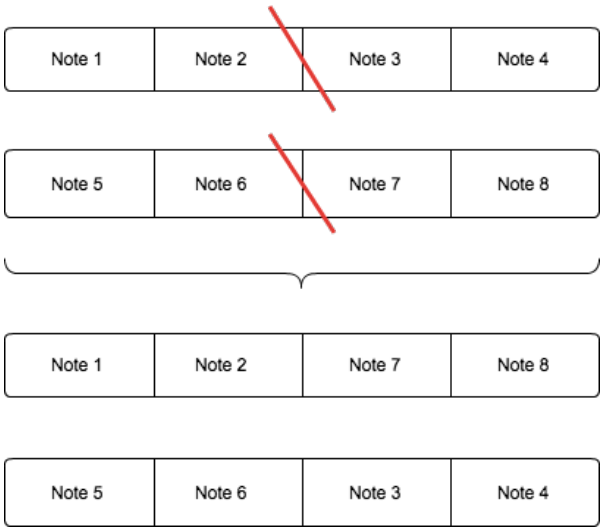

Figure 3.17: Representation of the Crossover Method

Figure 3.17 exemplifies how the crossover is applied within our system. The lines represent cutting points made upon each of the chromosomes. The results obtained through

this process still have a probability of suffering mutation, but afterwards are added to the new population.

Mutation

Mutation is a simple random percentage factor applied to a new-bred individual, which affects it's chromosome in a minor way[Mit98], such as changing a random 1 to 0 (or vice versa), or remove/add a bit for example.

For this experiment our standard mutation is simply the act of transforming a random playing note into a non-playing note or vice versa. The generic mutation function can influence a note in the following ways:

- **Play**: If that note is currently silent.
- **Silent**: If that note is currently playing.

In reality what it is doing is setting the Velocity to either 0 or to 100, which affects the notes being played, in this case its sonority.

However, it was necessary to create two types methods, on how this standard mutation algorithm would be applied onto our chromosomes. We needed one method that would kickstart the genetic process (as mentioned earlier), and another method that would be used during the evolutionary process itself. Its important to note that both of these methods would apply our standard mutation, although in different ways.

Initial Population Kickstart This method will take a note-map and select a random number between 0 and N (by default N = 5). For N times the function will choose one random position from the note-map and modify that note using the standard mutation function.

For the genetic algorithm specifically, this is done X times on the origin note-map, so as to create our initial population. This step is necessary to create a diversified initial population, so the chromosomes don't stale and the genetic process can come up with more diversified solutions. Usually the population initialization is randomized[Mit98], however in the specific case of this work it wouldn't make sense, because we want to keep some attributes of the origin note-map and not discard it completely.

It also should be noted that this method will work on any note-map type, and will in fact be used for some of our other optimization processes.

Evolutionary Method Mutation This method unlike the kickstart method, applies a probability of applying the standard mutation to every note during the evolutionary process. What this means is, that a note would have the probability of 1 in N chances of being

switched from play to silent or silent to play. By default N is set to 1000, so on average there is a probability that one within a 1000 notes will be mutated.

Also it is important to emphasize that this method is used during the evolutionary process itself.

3.4.2 Hill Climbing Process

The Hill Climbing Process uses a local search algorithm[MF04] (one of the various hill-climbing techniques) which focuses its search within a local neighborhood. The Local Search Algorithm goes as follows[MF04]:

1. Pick a solution from the search space and evaluate its merit. Define this as the current solution.

2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.

3. If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.

4. Repeat step 2 and 3 until the objective function gives a value that is at least equal to the user defined, natural stop condition (this step is specific to our research).

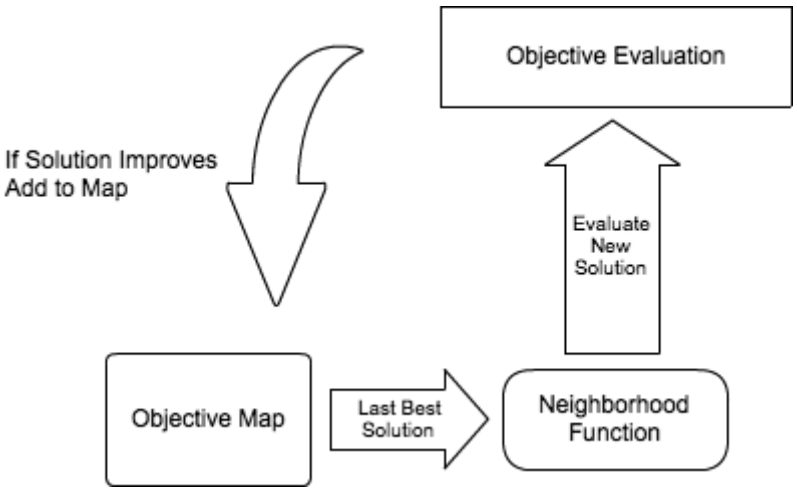

Figure 3.18: The Hill Climbing Process

First we had to define the neighborhood and the solution acceptance criteria. The neighborhood function was going to be variations of the last best solution or starting node (to kickstart the algorithm). Our Initial Population Kickstart method, which we created for our genetic algorithm component served perfectly well for our neighborhood function. The criteria for inserting a solution into our objective map was set to ”once the solution

improves". If a new generated solution satisfied this criteria, it would then be added to the objective map.

The process will always iterate over the last best note-map obtained, then apply the neighborhood function to it and finally rank it using the objective function, over and over until a better note-map is found. The iteration will run until a note-map reaches or passes the natural stop condition threshold (by default is set to 10). For this particular optimization process we did not set any length sequence stop condition, as it wasn't really necessary, because the algorithm worked fairly well with only the natural stop condition.

So in theory, once the optimization process has finished, we can expect a song that never repeats itself, thanks to the intrinsic nature of this algorithm (by only adding note-maps that are improvements of the precedent one). But on the other hand songs are much shorter, because their are less solutions entering the objective map, at least when compared to its genetic counterpart.

3.4.3 Stochastic Hill Climbing Process

The Stochastic Hill Climber is a modification of the Hill Climbing process, where the largest difference lies within the acceptance criteria. While the normal Hill Climbing method focuses on a more opportunistic approach of "Keep solution immediately, if best", the Stochastic's acceptance criteria is based on a probabilistic nature, where the rule of moving from a current solution v_c to a new solution v_n is based on a probability. The probabilistic formula for accepting a new solution focuses on maximizing the objective evaluation function[MF04], which means that a better ranked solution will garner a better chance of being picked than a lower ranked solution.

$$p = \frac{1}{1 + e^{\frac{eval(v_c) - eval(v_n)}{T}}}$$

Figure 3.19: The Probability of Acceptance

Analyzing Figure 3.19 the probability of acceptance depends heavily on the difference of merit between the score obtained by the last solution (v_c) and the score of the new solution (v_n) and of an additional parameter **T**. The T parameter influences the sway of the acceptance rate, for example the higher the T value is, the better the chance that a worse solution has of being accepted, the lower the T value the lower the chances are of a worse solution being accepted. The challenge of this algorithm is to find a T value "sweet spot" so to say, one that does not overcompensate worse solutions but also does not completely shut them down (if that were the case we'd have a normal Hill Climbing Algorithm).

To come up with a good T value we started out by randomly tweaking it and analyzing the objective maps solutions. We found that a T value between 0 and 0.1 were garnering the best results. So using an Excel spreadsheet we found a pretty good value of T = 0.07.

eval(v_c) - eval(v_n)	p
0.9	2.60743E-06
0.8	1.088E-05
0.7	4.53979E-05
0.6	0.000189406
0.5	0.000789866
0.4	0.003287661
0.3	0.013576917
0.2	0.054313266
0.1	0.19332137
0	0.5
-0.1	0.80667863
-0.2	0.945686734
-0.3	0.986423083
-0.4	0.996712339
-0.5	0.999210134

Table 3.1: Value of P with T = 0.07

However we did have another problem, which was of repetitiveness. When we experimented the algorithm we saw that a lot of the times our objective map would be flooded with the same ranked note-maps over and over, because the acceptance rate of two equivalent note-maps was at 50%, which by analyzing Table 3.1 we can clearly see this in effect, when eval(v_c) - eval(v_n) = 0. So to circumvent this we decided that if no changes were made to the note-map's rank the probability would be immediately ignored and pass to another solution. This allowed us to eliminate the repetitiveness of the final solution just like the normal hill climber.

The idea behind the stochastic hill climber is about creating a regressive possibility, which is counter to the normal hill climbing's "always best" kind of nature. This will create a more unpredictable song output, because the solution isn't merely trying to get towards its goal.

3.4.4 Optimization Filters

During the prototyping of the application we realized that a lot of the solutions that were returned by, most specifically the genetic optimization, contained a lot of repetitious patterns and very long objective maps, something that could saturate the listener from a overabundance of generated content. This, of course, was mostly due to the fact that we were using elitism in the genetic process. Although this was not the only reason, we also

wanted to give the user some simple editing tools, such as discarding unwanted parts of a solution.

Filters are methods of shortening the optimization solutions, which have probably grown too large for it to be usable. A filter will take care of shortening a solution by eliminating note-maps according to a user selected specification. For the purpose of this research we created 3 types of Filters:

- The Point to Point Cut
- The Rank Cut
- The Repetition Cut

Filters aren't mutually exclusive, meaning that if a user applies a Point to Point Cut he can also apply a Fitness or Repetition Cut if he so chooses. However, there are filters that do not make sense on certain optimization processes such as applying the repetition cut on a hill climbing solution, for example.

The Point to Point Cut

The Point to Point Cut is a filter method that cuts the N note-maps between a position X and a position Y of an objective map (Figure 3.20). This filter takes into consideration the position of a note-map within the objective map and will cut all note-maps whose position is equal or higher to X and equal or lower than Y. This filter should be used mostly as a form of editing tool, which will cut unwanted parts of the song.

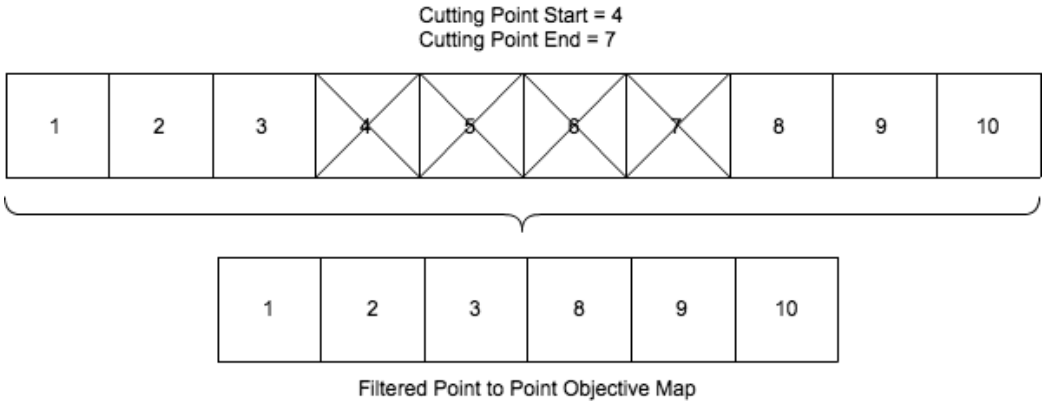

Figure 3.20: The Point to Point Cut Filter

The Rank Cut

The Rank Cut is another type of filter, which consists of eliminating all of the note-maps from an objective map, whose rank are not within a user specified interval. For example

if a user states that all note-maps must have a minimum rank of 5.5 and never exceed the rank of 8.5, all note-maps who are within this rank interval are kept while the rest is eliminated (see Figure 3.21).

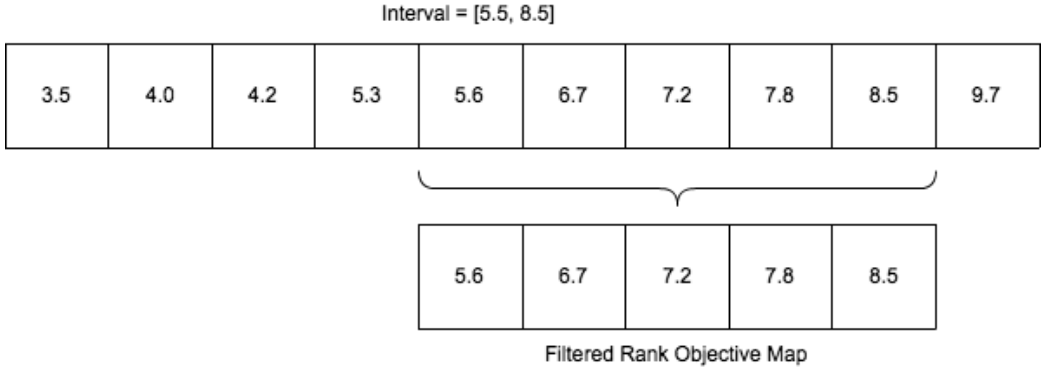

Figure 3.21: The Rank Cut Filter

It should be noted that this filter option will in fact work on a stochastic hill climbing solution, but because the note-map ranking is so unpredictable in this optimization method, the filtered solution may vary very much from the pre-filtered solution. So using this filter on a stochastic solution should be used with caution.

The Repetition Cut

The Repetition Cut is another rank filter, and like the name entails it will eliminate repetitive note-map rank scores that appear consecutively. The user can define a leeway of how many repetitions the filter can ignore, for example letting the objective map keep at least two consecutive notes ranked with the same value. This leeway exists because repetitiveness can sometimes be welcomed depending on which user is using the system.

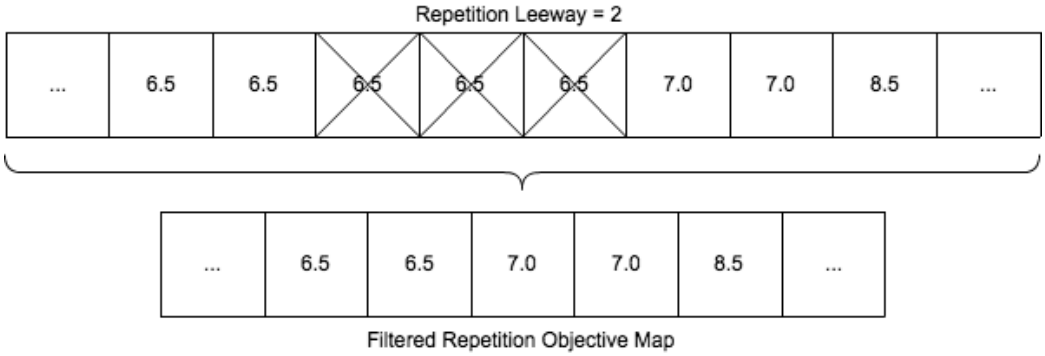

Figure 3.22: The Repetition Cut Filter

For both Hill Climbing methods repetition never occurs, thanks to the intrinsic nature of those optimization processes. So applying a Repetition Filter on either a Stochastic or a Normal Hill Climbing solution will not yield any sort of difference towards the original

solution. This filter is specifically targeted towards the genetic algorithm optimization, whose solutions are often long and repetitive. This filter will help reduce the excessive repetition into something the user might find less tedious.

Chapter 4

Traveling the Percussion Graph

When the idea of percussion graphs was discussed, one of the first things that had come to mind was the idea of a wandering percussionist, who would travel from node to node playing their respective connections along the way. The idea wasn't for the user to specifically define a path, but simply let a chosen algorithm to choose a path for him. A path will be a sequence of connections, that symbolizes the transition from the start node to end node.

To implement this idea we came up with two solutions. The user would be able to choose from two ways of traveling a percussion graph:

- By defining a Start and End Node
- Applying the Traveling Salesman Problem

In the first case, the user will set one of the graph nodes as a starting node and another node as an ending node and a path finding algorithm will then try to find a path, that can accomplish the task of going from start to end, which depending on the connections that were created will not always be possible. It should be noted that a path isn't obliged to visit all the graph nodes, however we believe that it would be musically interesting to allow nodes to be visited more than once. For this option we setup a user defined parameter imposing a limit on the number of allowed repetitions per node.

There are two possible outcomes of this type of path travel:

- A Solution is returned, either with repetition or no repetition;
- A Solution is not returned, because no path is available between start and end node.

The Traveling Salesman (TSP) is another travel type, where a song is created by visiting all the user created nodes once. Because creating a compatible TSP graph can be a somewhat tedious task for the user (creating a connection for every node for example), the algorithm will automatically generate connections for all user created nodes and will

assume that a graph is fully connected. The starting node will be out of the control of the user as it will be randomly selected.

4.1 Path Traveling Algorithms

There are several types of path finding algorithms that could have been chosen, however we settled on the following three:

- Breadth-first Search
- Depth-first Search
- Random Search

We chose these three algorithms at first for their simplicity. Due to the low complexity (in terms of number of nodes and connections) we were expecting from a percussion graph, it wasn't necessary to come up with a highly efficient and complex algorithm. This was not the only reason however, as these algorithms can actually produce enough musical diversity.

From the early conceptualization of the application we wanted to give the user full control over the connection creation process, so there is always a possibility of a user purposefully creating looping or impossible paths. Because of this, all of the path finding algorithms allow the user to define a number of repetitions allowed within a path.

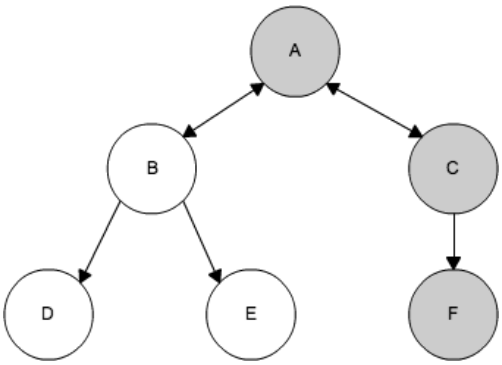

Figure 4.1: Example of a Path [A C F]

The Breadth-First Search (BFS) is an interesting option that will always return the shortest path (in terms of number of connections) if available. Also a curious note is that, if the repetition option is turned on and a path is available, the algorithm will never repeat a node!

The Depth-First Search (DFS) is also an interesting option, contrary to its BFS counterpart the DFS does allow repetition even if a path is available. Of course this heavily depends on the user created graph.

The Random Search is an option for users who enjoy a more experimental solution and unexpected path. This option will randomly choose one path that is available. This option does not guarantee it will reach the end node even if there is a path available, however it will always return a solution with all the places it has visited.

4.1.1 Breadth-first Search

The Breadth-first Search (BFS) is a graph search method that begins at the root and explores all of its neighboring nodes. Then subsequently for all the nearest nodes explores their neighbors and so on[RN02].

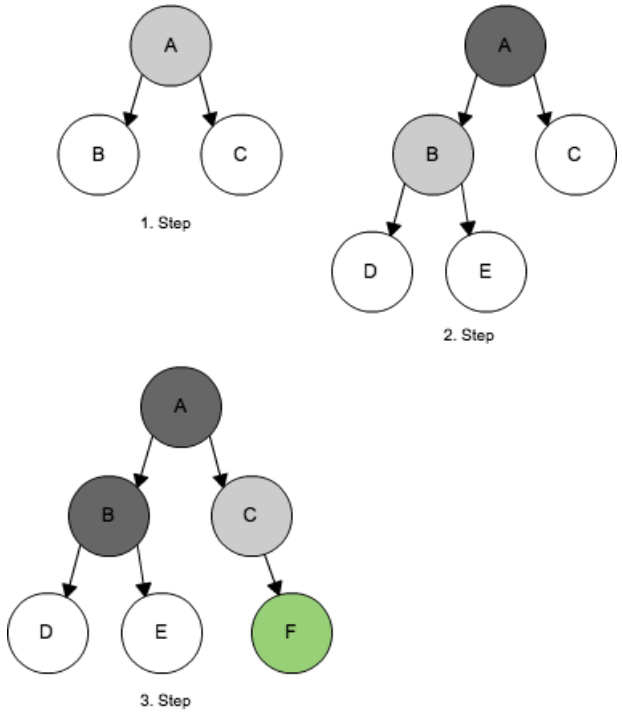

Figure 4.2: Simple Example of the BFS Algorithm on Figure 4.1 without Repetition

For the implementation of the method, a FIFO (First In, First Out) approach is usually used[RN02]:

A search node is a path from some state X to the start state S, e.g., (X B A S)
The state of the search node is the last state of the node, e.g., X
Let Q be a list of search nodes, e.g., ((X B A S), (C B A S) . . .)
Let S be the start state (Note that in the percussion graph case a state will be a note graph, which is different from a search node, which is a path).

1. Initialize Q with search node (S) as only entry

2. if Q is empty, fail. Else, pick the first element N from Q
3. if state(N) is goal, return N reversed (we have reached the goal)
4. (Otherwise) Remove N from Q
5. Find all the descendants of N if the satisfy the repetition constrain, i.e, no state can appear in N repeated more than the repetition limit, and create all the one-step extensions of N to each of the valid descendants.
6. Add the extended paths to end of Q
7. Go to step 2

Because we were using Java we used a LinkedList from the java.util library to be our FIFO queue. Another thing we had to have in mind was that we weren't searching for the end, but for the path that led there instead. So we created another LinkedList, that would hold the ongoing path.

Using Figure 4.2 & 4.1 as an example:

1. Insert Root Node into the Queue: [[A]]
   (a) Extend Node A - B, C
   (b) Insert New Paths into the Queue after removing path at the head.
2. Current List: [[B A] [C A]]. B is not Goal.
   (a) Extend Node B - D, E
   (b) Insert New Paths into the Queue after removing path at the head.
3. Current List: [[C A] [D B A] [E B A]]. C is not Goal
   (a) Extend Node C - F
   (b) Insert New Paths into the Queue after removing path at the head.
4. Current List: [[D B A] [E B A] [F C A]]. D is not Goal
   (a) Extend Node D.
   (b) D has no Neighbors. No Path is Added.
5. Current List: [[E B A] [F C A]]. E is not Goal
   (a) Extend Node E.

(b) E has no Neighbors. No Path is Added.

6. Current List: [[F C A]]. F is the Goal

7. Return path [A C F]

Also it is important to note that if node repetition is turned on (no matter how many repetitions are allowed) or off it won't cause a difference to the final solution, because the BFS will always get the shortest path (if possible) available.

4.1.2 Depth-first Search

The Depth-first Search (DFS) is another graph search method, which contrary to the BFS tries to explore each node as far as it possibly can. By starting at the root node, it will explore all the available connections of one of its neighbors. Once all of those paths are explored it will subsequently move on to the next neighbor[RN02]. This trait is shared with all the nodes in the tree.

The implementation of the DFS is almost exactly the same as the BFS, the main difference lies in its approach, while the BFS uses a FIFO approach the DFS uses a First in, Last out approach:

A search node is a path from some state X to the start state S, e.g., (X B A S)
The state of the search node is the last state of the node, e.g., X
Let Q be a list of search nodes, e.g., ((X B A S), (C B A S) . . .)
Let S be the start state

1. Initialize Q with search node (S) as only entry

2. if Q is empty, fail. Else, pick the first element from Q

3. if state(N) is goal, return N reversed (we have reached the goal)

4. (Otherwise) Remove N from Q

5. Find all the descendants of N that satisfy the repetition constraint, i.e, no state can appear in N repeated more than the repetition limit, and create all the one-step extensions of N to each of the valid descendants.

6. Add the extended paths to front of Q

7. Go to step 2

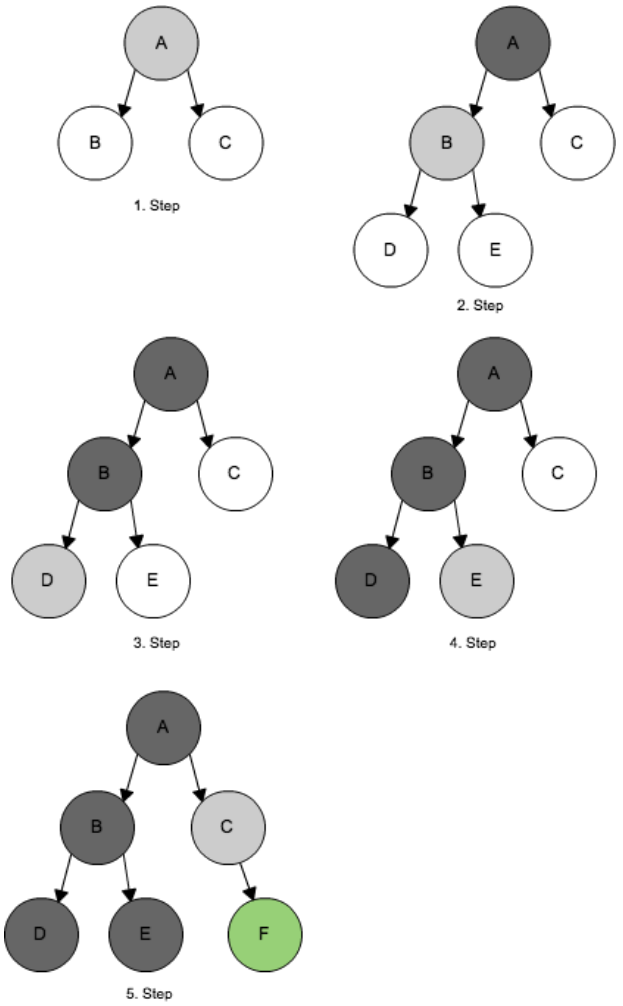

Figure 4.3: Simple Example of the DFS Algorithm on Figure 4.1 without repetition

We still used a LinkedList (to re-use most of the code already written), with the slight difference of instead of inserting a new path at the end of the list it would be inserted at the head.

Using Figure 4.3 & 4.1 as an example:

1. Insert Root Node at the head of the List: [[A]]. A is not Goal.
   (a) Remove First Path and Extend Node A - B, C
   (b) Insert New Paths at the Head of the List.
2. Current List: [[B A] [C A]]. B is not Goal.
   (a) Remove First Path and Extend Node B - D, E
   (b) Insert New Paths at the Head of the List

3. Current List: [[D B A] [E B A] [C A]]. D is not Goal.
   (a) Remove First Path and Extend Node D
   (b) D has no Neighbors. No Path is added to list.
4. Current List: [[E B A] [C A]]. E is not Goal.
   (a) Remove First Path and Extend Node E.
   (b) E has no Neighbors. No Path is added to list.
5. Current List [[C A]]. C is not Goal.
   (a) Remove First Path and Extend Node C - F.
   (b) Insert New Paths at the Head of the List.
6. Current List [[F C A]]. F is the Goal.
7. Return path [A C F]

Contrary to the BFS, the final solution of the DFS algorithm can suffer changes if repetition is turned on. However, this heavily depends on the graph. Using Figure 4.1 as an example, we could obtain the following result: [A B A C F], if we allowed 2 repetitions per node, because A will repeat twice, and once the algorithm finds A for the third time it will discard the current subtree and pass on to the next subtree list in the queue. Also it is important to note that, the order in which the tree is searched, also heavily influences the final result. In this case the DFS picked the subtree per alphabetical order, however if the algorithm had picked the subtree [C F] we would have obtained the same solution of the BFS algorithm ([A C F]).

4.1.3 Random Search

The Random Search algorithm is a completely different approach from the BFS and DFS algorithms. The random search's strategy is to randomly pick a neighboring node, travel to it and add it to our solution list:

A search node is a path from some state X to the start state S, e.g., (X B A S)
The state of the search node is the last state of the node, e.g., X
Let S be the start state

1. Initialize N with search node (S) as only entry
2. if N has a length equal to max-length or state(N) is goal or if N has no valid descendants, satisfying the repetition constraint, return N reversed.

3. (Otherwise) pick randomly one of the valid descendants and create a one-step extensions of N to the chosen descendant.

4. Go to step 2

This algorithm does present some unique particularities that are not existent in the other two path finding algorithms.

Firstly, repetition is always turned on in the random search. The reason why we chose to do this was because that, with repetition turned on, the algorithm has a much better chance of finding a solution from Start Node to End Node.

Another particularity is that, the user can define the maximum value of the random path length, which by default is set to the number of nodes of the current percussion graph. This option is necessary because of the unpredictability of this algorithm, and its easy tendency to enter an infinite loop.

There are 3 ways the random search algorithm will stop, by reaching the end node, by reaching the maximum list size or by reaching a dead end of the graph.

Using Figure 4.1 as a reference here some examples of the random search algorithm with a path length of 6 and repetition length of 3.

Example 1:

1. Insert Root Node into the List [A]

2. Pick Random Neighbor (Picked B) and add to List [B A]

3. Pick Random Neighbor (Picked D) and add to List [D B A]

4. Return Path: [A B D]

Example 2:

1. Insert Root Node into the List [A]

2. Pick Random Neighbor (Picked B) and add to List [B A]

3. Pick Random Neighbor (Picked A) and add to List [A B A]

4. Pick Random Neighbor (Picked C) and add to List [C A B A]

5. Pick Random Neighbor (Picked F) and add to List [F C A B A]

6. Return Path [A B A C F]

Example 3:

1. Insert Root Node into the List [A]
2. Pick Random Neighbor (Picked B) and add to List [B A]
3. Pick Random Neighbor (Picked A) and add to List [A B A]
4. Pick Random Neighbor (Picked C) and add to List [C A B A]
5. Pick Random Neighbor (Picked A) and add to List [A C A B A]
6. Pick Random Neighbor (Picked B) and add to List [B A C A B A]
7. Return Path [A B A C A B]

Its important to note that this algorithm will always return a solution, but it does not guarantee it will return a solution that goes from start node to end node. The reason for this, is because we wanted this algorithm to return something so that the user wouldn't feel the need to press the random button until it got the right solution. Besides we also wanted to add an element of unexpectedness with this algorithm.

4.2 Traveling Salesman with 2-Opt

When coming up with the concept of percussion graphs, one of the earlier decisions was to apply a Traveling Salesman Algorithm for one of the song generating solutions. Creating a song by following the shortest path and visiting all the nodes was a great idea, as it would be an easy way for the user to incorporate all of his created nodes into a song. It also adds another layer of unexpectedness and randomness to the final song as well, which is always interesting for experimentation!

The Traveling Salesman Problem (TSP), consists of a salesman who must visit N number o cities only once and then return home, using the shortest route possible[RN02][MF04].

Our TSP algorithm is composed of 3 phases:

1. The Initialization
2. The Nearest Neighbor Algorithm
3. The 2-Opt Optimization

The initialization phase consists of creating a fully connected graph using the current percussion graph nodes. The Nearest Neighbor phase will create a TSP solution out of the initialization graph, which will then be optimized by the 2-Opt algorithm.

There are multiple ways of solving the TSP[RN02][MF04], however we chose this methodology because it was a simpler and more elegant solution, and considering the size of our graphs it didn't really warrant a more complex algorithm.

4.2.1 Initializing the TSP Algorithm

First and foremost, this algorithm does not take into account any connections that were previously created by the user. Because of the complexity that is inherent to a TSP graph[MF04], it was decided that instead of making the user create a fully connected graph by hand, we might as well do it automatically as part of a initialization process.

So the initialization consists of generating connections for all of the nodes, so as to create a fully connected graph. This is needed because the value of each connection will be obtained through the length of the objective map. Meaning that the fewer note-maps an objective map has the shorter the path, and vice versa. It should be noted that before running the algorithm, the user must create at least 2 or more nodes, for it to start.

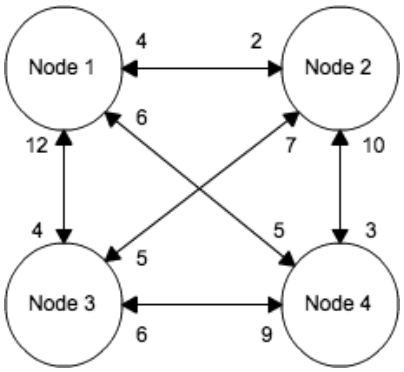

Figure 4.4: Fully Connected Percussion Graph with Connection Length

4.2.2 The Nearest Neighbor Algorithm

Once the initialization process has completed, we apply a nearest neighbor algorithm on the graph[BG05]. This algorithm will start by randomly choosing a graph node. Then at every subsequent iteration the algorithm will search for the node that still hasn't been inserted into the solution and that is closest to the last node added (see Figure 4.5).

However, most of the time this is far from the optimal solution, and its only the start of our optimization process. Once the nearest neighbor solution is outputted we'll apply the 2-Opt optimization which will give us a local optima.

4.2.3 The 2-Opt Optimization

The 2-Opt is a TSP optimization process first described by Croes[Cro58], and consists of optimizing a TSP tour. A tour is the journey between all the nodes of the TSP graph,

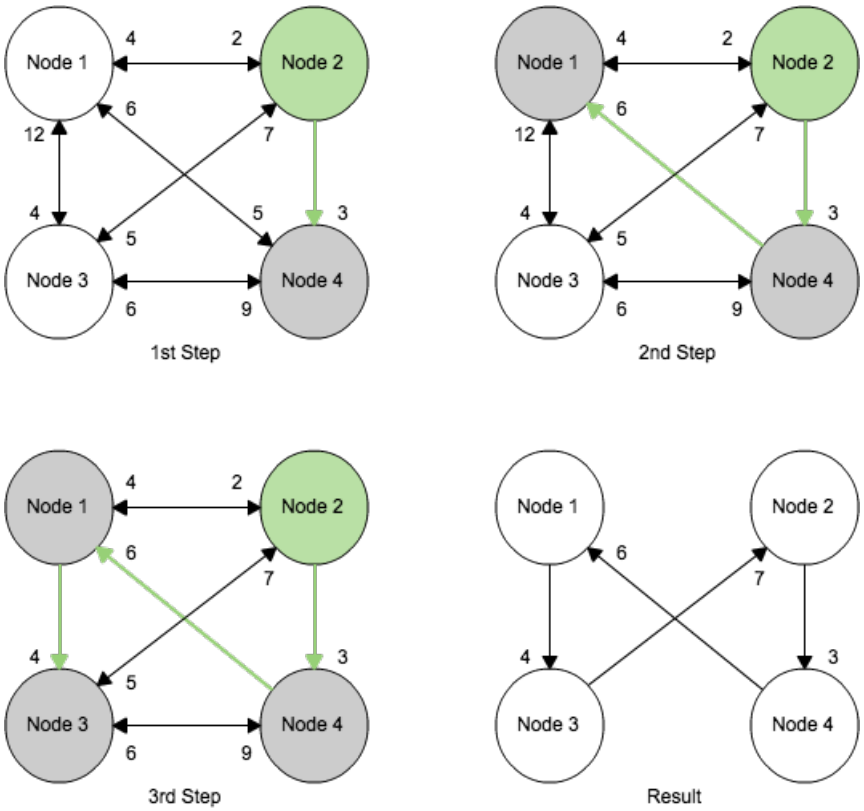

Figure 4.5: Step By Step of Nearest Neighbor Using Figure 4.4

in our case the starting tour will be the TSP solution obtained by our nearest neighbor algorithm.

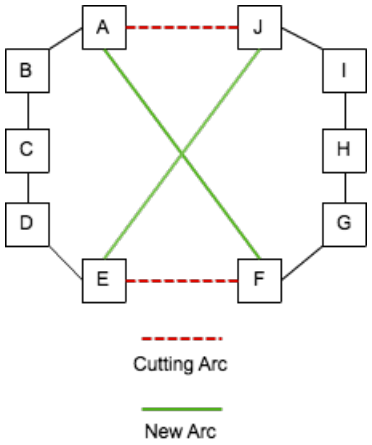

Figure 4.6: The 2-Exchange Operator

The 2-Opt Optimization is a form of local search[BG05], where the edge-exchange neighborhoods for a single route, are a set of tours that can be obtained from our initial Nearest Neighbor tour, by replacing 2 of its edges by another set of 2 edges. The algorithm will try to lower the cost of the tour by replacing two of its edges by two other edges and

will iterate until no further improvement on the tour cost is possible (see Figure 4.6).

For the implementation of the 2-Opt we found in the work of Misevicius A., et al.[MOŠŽ07] a nice pseudocode algorithm on which we could base our 2-Opt function.

```
function 2-opt(p);
// input: p – initial (starting) solution; output: p• – resulting (locally optimal) solution
begin
  p• := p;
  repeat
    p := p•;
    Δmin := 0;  // Δmin denotes the minimum difference in the objective function values
    for i := 1 to n – 2 do
      for j := i + 2 to n – 1 + Sign(i – 1) do begin
        Δ := z( p ⊕ φij ) – z(p);
        if Δ < Δmin then begin Δmin := Δ; k := i; l := j end
      end;  // for
    if Δmin < 0 then p• := p ⊕ φkl   // move from the current solution to a new one
  until Δmin = 0;
  return p•
end.
```

Figure 4.7: 2-Opt Pseudocode

Lets start by describing the 2-Opt pseudocode (Figure 4.7) gradually, so as to have a universal understanding on how this works within our system. The algorithm requires an already created tour of the graph, which as previously mentioned will be the solution from our nearest neighbor algorithm.

An iteration will consist of comparing tours by previewing the value of a tour that is obtained by exchanging each possible arc and comparing it with the old tour value.

Δ_{min} serves as the controller variable verifying if after an entire iteration the graph can be improved or not. It is initialized to 0 and subsequently at the start of each iteration. If a lower tour is found thanks to a particular arc exchange, Δ_{min} will obtain the value of Δ, which is the difference between the value of the current tour and the comparing tour.

The variable **i** will be the ”main” comparison node, in this case a placeholder for us to then verify each of the other arcs that are connected to this placeholder. The variable **j** will be the nodes unto which **i** will compare possible arc exchanges. Figure 4.8 shows how the comparison process progresses. Note that node E does not compare to node F because the exchange of neighboring arcs never yield any improvement to the tour.

Once an improved tour is found, its then a matter of rearranging all of the inside nodes and update the graph with the new solution. This is easily accomplished by simply swapping the inside nodes between **i + 1** and **j** (see Figure 4.9).

This entire process is iterated multiple times until Δ_{min} yields no changes (Δ_{min} = 0 for an entire cycle). Once the algorithm is finished it will return the play path, which in this case is the most optimized tour. Then its just a matter of loading this into the drum

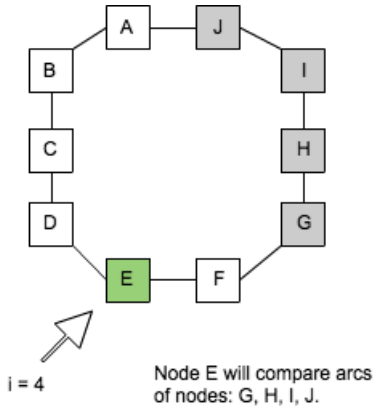

Figure 4.8: 2-Opt Comparing Nodes

machine so that the user can listen to its output and progression.

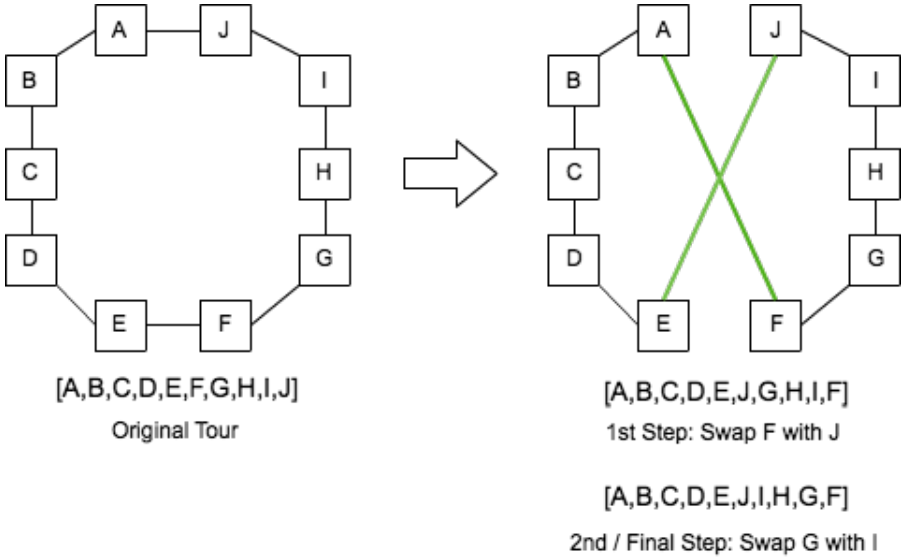

Figure 4.9: Creating a New Tour

Chapter 5

The Percussion Graph Application

In this chapter we will discuss how we were able to put all the conceptual ideas explained above into a working prototype application. Here we will cover all the aspects that are necessary to enable the user into creating and manipulating his own percussion graph, such as creating nodes, connections, generate songs, apply filters and get a song path. We would also like to point out that the system can be seen in action in our research website page (`http://bookmark.labmag.di.fc.ul.pt/?page_id=1966`).

The application consists of three main components, the Drum Machine Application, the Backend Process Component and the Percussion Graph Interface (Figure 5.1).

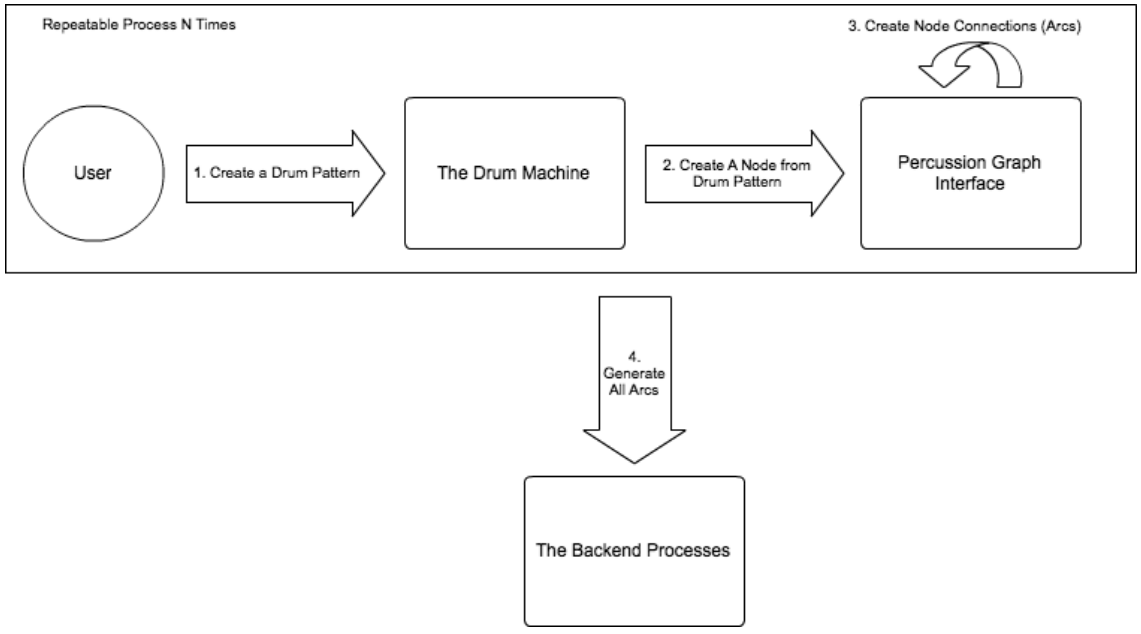

Figure 5.1: The Application Flow (A User Creates Nodes and Connects Arcs)

- **Drum Machine Application**: The Musical Interface between the user and the application. This is where the user will create rhythms, which can later be transferred into the application as nodes.

- **Percussion Graph Interface**: This is the Graph Interface, where the user defines the flow of the music. Here the user can create nodes, and create connections by linking two nodes with each other. The user can also define paths, define the optimization process to be used and apply filters to already optimized solutions.

- **The Backend Process Component**: This component will take care of all the Optimization, Path Finding and TSP Algorithm Processes (See Chapter 3 & 4).

5.1 The Drum Machine

The Drum Machine serves as the music player and the musical interface. This is the application where the user will create rhythms and be able to hear the musical output of connections, paths or even re-hear nodes that had been previously created. In a way the drum machine serves as the musical link between the user and the percussion graph system.

For the drum machine, we wanted to have something that would be a step up from the Netlogo prototype (See Appendix A). With much more believable drum sounds and a more responsive and robust interface.

The Monkey Machine by Pekka Kauppila[Kau] was a perfect fit for the experiment, as it combined all the aspects mentioned above in a nice easy to use package, and also included an XML import / export option which would greatly facilitate its communication with the other application components.

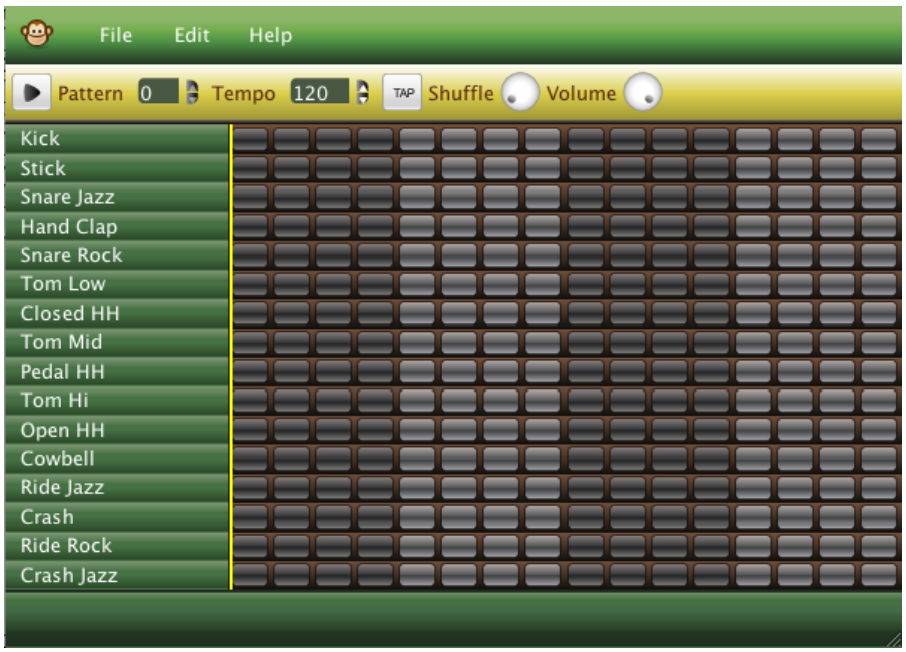

Figure 5.2: The Monkey Machine Interface

Figure 5.3: Monkey Machine Playing a Drum Pattern

The Drum Machine consists of a 16x16 grid, where each line of the grid represents a percussion instrument and each column represents the time in which the instrument is played (Figure 5.2).

The available instruments range from Bass Drum (a.k.a. Kick Drum), Snare Drum, Hand Claps, Toms, Cowbell, etc. The instrument sounds are provided through two VST's (Virtual Studio Technology) called Natural Studio[Lim] and Hydrogen[Hyd]. The Drum Machine also takes care of the entire MIDI mapping allowing for the right instruments to be played through the VSTs.

The timeline is a standard 4/4 notation, divided in eight notes, this means that the user can play an instrument up to 16 times in a measure. However the drum machine was limited to a single user inputted measure, the rest of the measures appear from the solutions obtained from the optimization process.

Another limitation applied to the drum machine was the velocity factor. Velocity is the force in which an instrument is played (in the MIDI standard). The drum machine allows two note forms, high velocity note (ex. hitting the drum harder) and a medium velocity note (ex. hitting the drum moderately) we ignored the latter option as another form of simplifying the evolutionary process. Though in a future implementation it would be possible to rework the note-map to fit into this 3 velocity model and could possibly add more depth to the generated songs.

5.1.1 The XML Input and Output

One of the main advantages of using the Monkey Drum Machine application was the ability to import and export drum patterns to XML. This feature greatly facilitated the communication between the user interface and the drum machine.

When a user creates a drum pattern, the drum machine will export it into an XML format to the Percussion Graph Component. The application will then parse the XML and create a Node with that pattern. The created pattern can be reloaded into the drum machine, in case the user forgets what he had created or simply wants to re-hear the Node's associated rhythm. To do this, the user only needs to click on the Node in the graphical interface, and the associated rhythm will load into the drum machine. Here is an example of an XML pattern (XML of the pattern in Figure 5.3):

```
<?xml version="1.0" encoding="UTF-8"?>
<style version="2.0">
   <kit>GMkit.h2drumkit</kit>
   <bpm>120</bpm>
   <shuffle>0</shuffle>
   <patterns>
      <pattern id="0" length="16">
         <drum id="0">1000000000000000</drum>
         <drum id="1">0000000000000010</drum>
         <drum id="2">0001000100100100</drum>
         <drum id="5">0100000001000000</drum>
         <drum id="6">0000000000100000</drum>
         <drum id="7">0011110000000010</drum>
         <drum id="8">0000001000000000</drum>
         <drum id="11">0000000001000100</drum>
         <drum id="12">0010000000000000</drum>
         <drum id="13">0000010000000000</drum>
         <drum id="15">0010000000010000</drum>
      </pattern>
   </patterns>
</style>
```

- The Drum ids represent each instrument in the drum machine (16 in this case), while the 1s and 0s represent if an instrument is silent or not, in that particular moment in time.
- The Pattern tag indicates the number of the patterns. A song can have multiple patterns, which is usually the case of most of the music that is generated by the optimization processes.

- The BPM tag refers to the beats per minute, which isn't used by any of the components, however the user can alter it by hand to speed up the song.
- The Shuffle tag is never used by the application and is turned off by default (the user can alter it, but the application ignores it).
- The Kit tag refers to which drum kit the drum machine will be using, by default and as a way to reduce the complexity of the other components the drum machine always uses the default GMKit by Hydrogen and it's sound banks.

When a connection is loaded into the drum machine, the system will generate an XML very similar to this one, but with likely more patterns than the one showed above (which is a one pattern song), and import it into the monkey machine.

5.2 The Percussion Graph Interface

The percussion graph interface is the main application interface and will enable percussion graph creation and manipulation to the user. This component serves as the other main interface component of the application. The available user interactions are:

- Node Creation.
- Node Manipulation (move the Nodes around the display graph).
- Node Deletion.
- Load a Node into the Drum Machine (play the drum pattern associated to that Node).
- Connection Creation (create a connection between Nodes).
- Connection Deletion.
- Load a Connection into the Drum Machine (load the associated generated song into the drum machine).
- Select the Optimization Process.
- Start the Optimization Process for all connections that are existent within the user created graph.
- Calculate and play the TSP version of the graph (Traveling Salesman Problem with 2-Opt).
- Define a Start and Ending Node for path traveling.
- Search & Play a Path Between the Start and End Node.
- Apply Connection Filters, such as Point-to-Point Filter, Rank Filter or Repetition Filter.

- Clear the Drum Machine.

The interface in itself was kept simple as it was not the main focus of this research. It was created merely as an inevitable solution to the problem of graph creation and manipulation and as an easy and intuitive way of creating percussion graphs. The visualization of the graph is also extremely helpful as it helps keep the musical progression in check, such as loading both nodes of a connection and the connection itself so as to compare the similarities and the music tendency.

Figure 5.4 show's a screenshot of the main panel of the Percussion Graph Interface. The left panel represents the options menu, here the user can find most of the graph manipulation tools at their disposal, plus options that effect the Drum Machine (such as clearing all of it's patterns). The right panel is the drawing board, this is where a user can create/remove nodes or connections, manipulate the graph or even access the filter options (by right clicking a connection).

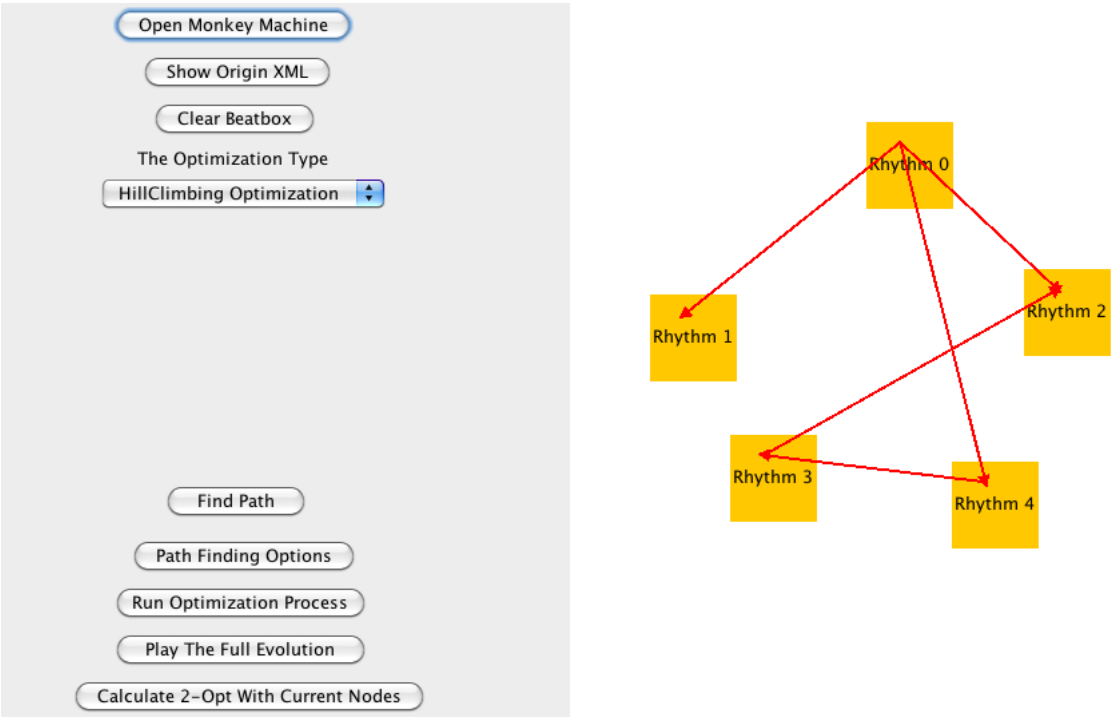

Figure 5.4: The Percussion Graph Interface

All of the interface was created using the Java Swing Library and with the aid of the Netbean's Interface builder.

5.2.1 The Drawing Board

The drawing board is the right hand side of the user Interface. This is where the user will be able to define the flow of the graph, by creating nodes, connecting them and manipulating them. The user is also able to load nodes and connections (once the optimization process finishes) into the drum machine to analyze and hear the musical output that was generated by the selected optimization process.

Creating a Node and a Connection

To create a node a user simply inserts a rhythm into the Drum Machine component and then by right clicking on an empty space on the drawing board, select the option ”Add Current Beat as Node” from the popup menu. This will create a new Node with the current rhythm in the drum machine.

To create a Connection a user needs to create at least two or more nodes. Usually it is recommended to create all nodes beforehand and then apply the connections, however it is not obligatory. A connection is created by right clicking on the user's intended start node and selecting the ”Set Connection” option from the popup menu. The user then only needs to click and drag the mouse from the start node towards the intended ending node. Once the button is released a new connection is created, going from the starting node towards the ending node.

When a new connection is created its color will be red, this symbolizes that this particular connection still has no optimization solution associated with it. Once an optimization process is run, the connection will turn green symbolizing that it can now be loaded into the Drum Machine or a filter can be applied.

Deleting a Connection or a Node is achieved by simply right clicking it and selecting delete from the popup menu. It should be noted however, that if a user deletes a Node that is currently being affected by one or more connections (either being a Start Node or End Node) those will also be subsequently deleted.

Applying a Filter

A Filter can only be applied on a optimized connection, which as previously stated will appear green if that is the case. To apply a filter the user must right click an optimized connection and select the apply filter from the popup menu. The Filter Popup Panel will appear (see Figure 5.5), and by default will choose the Point to Point cut. The user however, can choose which filter can be applied by simply using the drop down menu found on the filter panel.

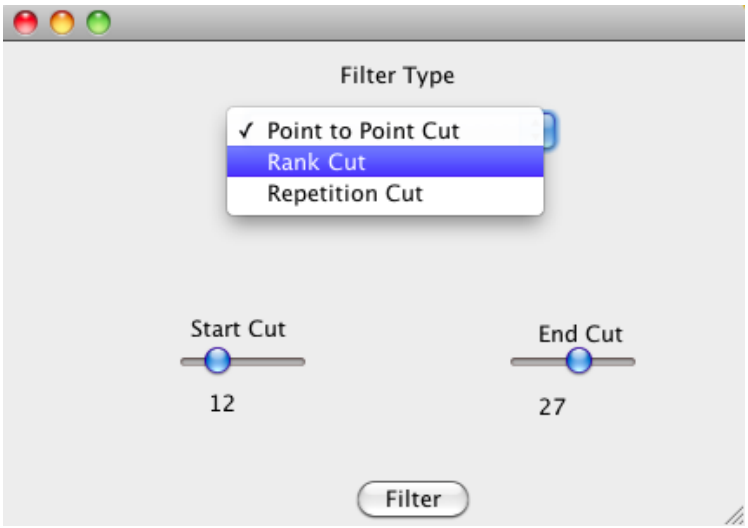

Figure 5.5: The Filter Panel

Defining a Start and End Node for Path Finding

Finding a song path starts out by defining a Path Starting Node and a Path Ending Node, which are unique Node statuses that can only occur once. To do this the user simply right clicks the intended path starting node or path ending node and selecting their respective choice on the popup menu (see Figure 5.6).

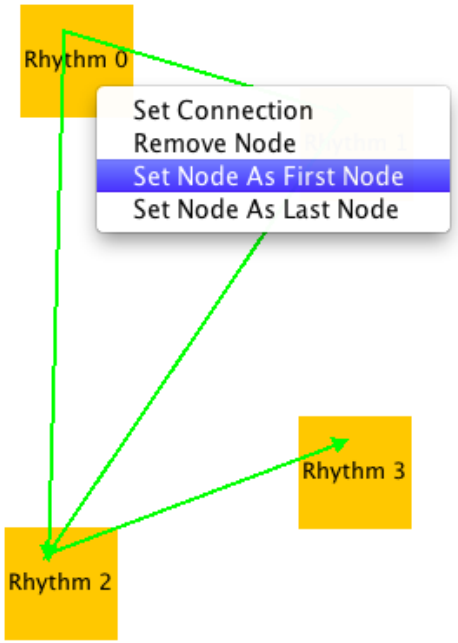

Figure 5.6: Selecting a Path Starting Node or a Path Ending Node

When a Path Start Node is selected, the Node's color will turn green symbolizing

the start of the path, while the Path End Node will turn red symbolizing the end of the path (see Figure 5.7). Also a Node can never be a a Path Start Node and Path End Node simultaneously or can their be more than one Start or End Path Node.

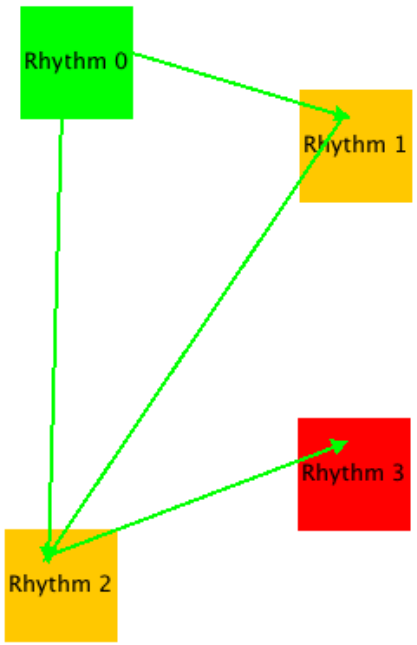

Figure 5.7: Defined Graph Path

To remove path status from a node a user must simply right click on the node and re-choose the option a second time and it will remove the current status, returning the node back to normal.

5.2.2 The Options Panel

The Options Panel (see Figure 5.8) lets the user apply global graph options that will influence or be influenced by the entire user created graph in some way. Such as starting the optimization process on all created connections, selecting the optimization process, finding a song path, defining the song path options and calculate the 2-Opt TSP song path. The Options Panel also offers a few simple Drum Machine controls such as starting the drum machine, clearing the current drum machine pattern and viewing the current drum pattern's XML.

Optimization Process Options

Applying optimization is a pretty straightforward process, the user must have at least one connection created to be able to run the optimization. The type of optimization can be chosen in the drop down menu on the options panel (see Figure 5.8). When the User

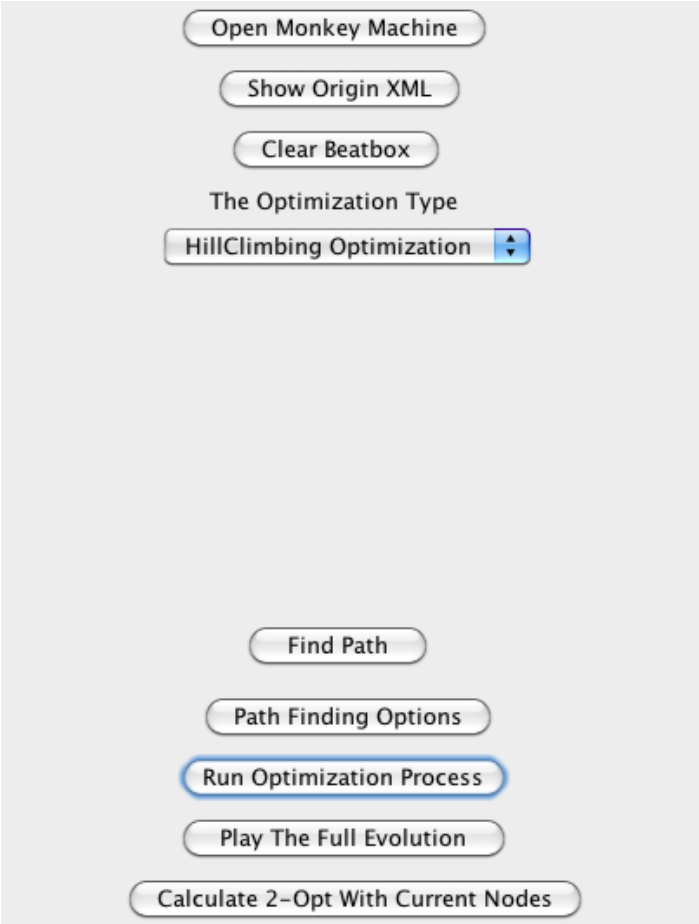

Figure 5.8: The Options Panel

runs the Optimization Process the application will run the selected optimization on all the connections existent in the user created graph. If a user creates another connection once the optimization process has finished, the process will only optimize the new connection. However, if the user changes the Optimization Type after the connections have already been optimized, every connection will suffer a re-optimization under the new selected Optimization Type, removing the previous optimization solutions and replacing them with the new ones.

Path Finding Options

The Path Finding Options can be accessed through the ”Path Finding Options” button, located on the options panel. By default the path finding search is set to the Breadth-First Search Algorithm without repetitions. Once a user is satisfied with the path options, simply clicking the ”Confirm Changes” button will save the new path finding options.

To run the path finding algorithm, the user simply clicks on the ”find path” button, which will then generate a path by following the chosen algorithm. Once this is done, it will directly load all of the rhythms found within the chosen path into the drum machine.

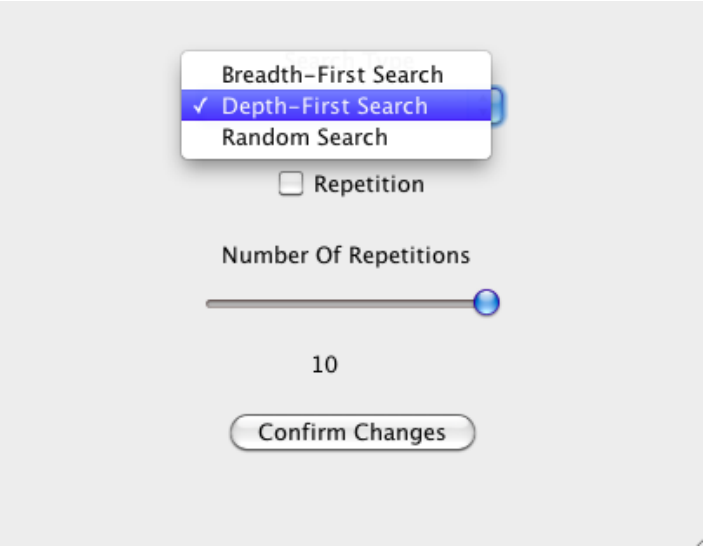

Figure 5.9: The Path Options Panel

However, even though it is not necessary to optimize the connections to load a path, it will provide nothing into the drum machine as there are no solutions in the connections. So it is always good to remember to optimize connections before playing with the percussion graph!

The Path Algorithm will return errors for the following, if there is no Start Path and End Path Node, if there is no available path to the End Path Node (depends on the algorithm) and if there are no Nodes or Connections existent in the Drawing Board.

The TSP 2-Opt Path

This option does not force the user to create connections, however the user does need to create 2 or more nodes for this algorithm to run. This feature can take some time to finish as it will try and create a fully connected graph using all the user created nodes, so that the algorithm can then apply the Nearest Neighbor and 2-Opt algorithms.

To start the process the user only needs to push the ”Calculate 2-Opt with Current Nodes” button and it will run, once the process finishes the result will be loaded into the Drum Machine. The selected path can be seen in the java command console (just like its path finding counterpart), for experimental and result feedback.

5.2.3 Backend Process Component

Most of what this component does has already been explained in Chapter 3 & 4, this component takes care of optimizing all the connections created by a user, either Hill Climbing, Stochastic Hill Climbing or Genetic, and all the path finding processes, either Breath-First, Depth-First or Random.

We felt the need to separate this into a component as it symbolizes the ”backend” of the application where most of the processes such as Path Finding and Optimization take place. This Component is fully autonomous once the user starts the path or optimizing processes and will then return the result back to the front-end application.

For the Genetic Algorithm implementation we used an API called JGAP[Mef] which is a Genetic Algorithm API featuring all of the selection, mutation, crossover and elitism methods mentioned. The Stochastic and Hill Climbing was implemented without the aid of the API and implemented from scratch. However all three algorithms do share some code, such as the Objective Function which is the same for all three algorithms, the starter mutation/randomization which is used by all three algorithms as well, the note-map and objective-map representations and finally of course the XML drum machine parser which helps load the solutions into the drum machine. The Path Finding algorithms were all implemented from scratch, including the 2-Opt algorithm.

Chapter 6

Experiments & Results

In this chapter we discuss the various experiments that were conducted on the system and review on the results that were obtained. We also analyze and discuss the various results that were obtained during our user testing phase.

6.1 Application & Musical Appreciation

In this section we will discuss the main differences between optimization processes (how a connection is generated) and analyze their musical output. We will also test the various travel algorithms and discuss how they fare using different graph types.

6.1.1 The Optimization Processes

For this group of experiments, we tested all the optimizations using the same graph with all the same connections and nodes. We did this for three distinct graphs and then analyzed their output and optimization time.

Genetic

During the first experiment we noticed that the genetic algorithm does present a more erratic behavior. Playing rhythms that are completely distinct from one generation to the next, however after a few generations it does have tendency to stabilize. Unfortunately a very common trend of this algorithm is the incapability to converge under at least 100 generations, which unfortunately hinders the algorithm quite significantly. We also tried changing the mutation rate from 1000 to 10 which we thought would provoke a change, however that wasn't the case. The scores obtained were quite similar to the ones with the lower mutation, something that was quite interesting.

We also tried it without any elitism whatsoever and obtained as was expected some pretty chaotic results. The rank usually shifted back and forth, until eventually stabilizing around the scores within the 4 to 7 interval. Musically some connections were very erratic,

with up to 10 different note changes between some generations, while other connections were less chaotic but the erratic sudden shift between rhythms was still present.

The interesting progressive pattern of the genetic algorithm despite whatever options are chosen, either with elitism or not, or with a higher mutation rate, it always finds a stabilizing point or generation in this case, where from there it will heavily rely on its mutation function. The main difference between them is that with using elitism the stabilizing point is reached much faster with a subsequently higher ranked individual, while without elitism it is reached much slower and usually with a much lower ranked individual.

The other problem with the genetic algorithm is that its convergence rate is very slow. We believe that this is due to the fact that there are so many random factors that weight in on this algorithm. Such as the roulette selection, the randomness of the mutation and even the crossover method, can create a convergence impediment.

Hill Climbing

Immediately after analyzing two connections we noticed the efficiency of this algorithm, it is very straightforward and when playing songs it seems to focus heavily on commons and exact notes, which was to be expected. The changes between rhythms are very minimalistic, usually adding or removing up 1 to 4 notes every pattern change. In terms of length, Hill Climbing presented the shortest lengthen songs, when compared to the other two methods.

In terms of musicality, the outputted results were very surprising. For example in our first experiment the connection that went from node 2 to node 1 had only 20 patterns and every pattern was genuinely interesting. After asking for some second opinions from colleagues about the musicality of the rhythms, most of them were surprised to learn that all of the rhythms had actually been generated by an algorithm. Its also interesting to note that the best connections musically were usually the ones that were quick to converge, although it was not always the case but a very frequent one.

Personally this algorithm was the most efficient in terms of convergence, with only a few instances where the algorithm actually struggled to complete its cycle, although in the end it converged. Playing with the objective functions Note and Length Similarity weights, we also noted that if the length similarity is particularly low, the hill climbing method struggles to converge and will definitely take more time to process. Compared to the other algorithms, hill climbing on average was the fastest in generating connections and with the shorter connection length.

It's hard to determine if luck is the only factor in generating good musical solution outputs, however its interesting to see that the system is actually capable of creating rhythms that can actually surprise someone who underestimates the systems musicality.

Stochastic Hill Climbing

When we started experimenting with the stochastic hill climbing, we had several problems from the get go. One was that the algorithm took just too long to converge, before the experiment the only way to stop the process was to generate a note-map with a rank of 10. One of the available solutions was to lower the threshold to a value around 9 and 9.5, however the side effect of this was that we would be eliminating the possibility of this algorithm converging. So the next solution we decided to use was to, like the genetic algorithm, limit the algorithm to n number of iterations, which was much more successful.

Interestingly enough, when we conducted our first experiment all of the connections converged when we ran it with the iteration threshold. Which to some extent showed us how luck based this algorithm really is, there were a lot of situations where a solution was better than its previous solution, however failed to pass the probability test.

Listening to the outputted results were quite interesting. We noticed a lot of similarities with the normal hill climbing algorithm, which were expected. Although the music seemed to progress at a much slower pace, for example a rhythm would only shift a single note from one rhythm to the next. During our musical appreciation phases we literally thought the algorithm was creating repetition, however the real reason was that rhythms were so similar in structure that it sounded like the algorithm was repeating the same pattern, when it actually wasn't.

Also just like its hill climbing counterpart, the best connections rhythmically were the short length ones. A phenomenon which is particularly interesting. Another interesting particularity was that, because the patterns change so slowly with each iteration, the earlier and latter generated rhythms are but small variations of user created rhythms, something that could be interesting to play with.

Once we toyed with the objective functions evaluation weights a particular interesting phenomenon did appear. Setting the length similarity particularly low in comparison to its position evaluation, created a convergence abnormality which we dubbed as flooding. Flooding consists of flooding the drum machine with notes and then analyzing each of these notes and verifying if any of them are exactly equal with the ending nodes rhythm. What we believe this does is, thanks to the limited restrictiveness of the length similarity, facilitate the algorithm into finding exact position notes more easily by simply testing every possible solution available.

When the algorithm does not engage in the flooding tactic, the outputted results are very interesting. Although this is a situation that mainly occurs when the length similarity is set to a particularly low weight, and was an interesting observation.

Discussion

During each experiment it was interesting to see the different variations of each algorithm, although we were a bit disappointed with the genetic algorithm's performance. Because

of the low convergence rate when compared with the other two algorithms, although this is a criticism on the algorithm's performance rather then the actual music itself. Flooding was also something that surprised us, although musically very chaotic it was an interesting convergence phenomenon.

In terms of convergence rate, the clear winner was the normal hill climbing method, which reaches convergence 100% of the time. While the stochastic and the genetic have a much more difficult time to converge.

Evaluating musicality is of course a relatively subjective component as it depends highly on its listener, in this case our very own opinions. Its also hard to simply say, this algorithm was better than the other one, because in reality all of the algorithms generated something that we enjoyed and something that we didn't and because of this we cannot simply state what was better, just that the three do things that are relatively different from one another.

In the case of the hill climbing method, we really enjoyed the constant progression of the rhythm which were simple and very straightforward. Although some can say that the algorithm was maybe to fast to converge, we believe that the length was just right most of the time, not to long and not to short. Another interesting observation was that, while the stochastic hill climber suffered from flooding, if the evaluation weight was low, the normal hill climbing did not, at least in all of our conducted tests. There were a few instances however, where a pattern was seemingly cluttered with notes but never like its stochastic counterpart.

The genetic algorithm, musically, was the most distinct out of all the three. The patterns were always very different and truly original compared to both hill climbing methods, although it suffers a lot from excessive repetition, something that luckily filters take care of. The best genetic rhythms are clearly the earlier generations, simply because the latter generations tend to stabilize themselves creating a monotonous sound. We actually tried lowering the maximum number of generations (25 to be exact) and obtained much more interesting results than higher generation ones. Of course with this kind of limitation the genetic algorithm would never be able to converge, unless being extremely lucky, though the obtained results was something to take into account.

The stochastic hill climbing was the most difficult one to work with and which, unfortunately, had to suffer some limitations during our experiments. However once we started to hear the results, they were actually quite fascinating. The rhythm progression of this algorithm was the complete opposite of that of the genetic, its progression seemed very orderly and minimalistic. The flooding, although an interesting phenomenon, really destroyed the musicality of the whole connection, but it's something that can easily be fixed by applying a filter, or just simply using the default values for the length and position evaluation.

In the end we believe that musically each algorithm did provide something different

for our ears, and that it was an experience to listen to. Its interesting to see how much each algorithm relies on the same objective function, but how different they are musically. Theoretically a different objective function could have helped make the genetic and stochastic converge much more frequently, however converging wasn't really the main objective of the work, and by analyzing the musicality it really didn't factor in as much to the generated rhythms as we thought it would, and could have even probably worsen them, in the case of the genetic algorithm specifically.

6.1.2 Travel Algorithms

We tested the various traveling algorithms with the repetition option, as doing so without wouldn't yield much difference between them, with of course, the exception of the random traveling algorithm.

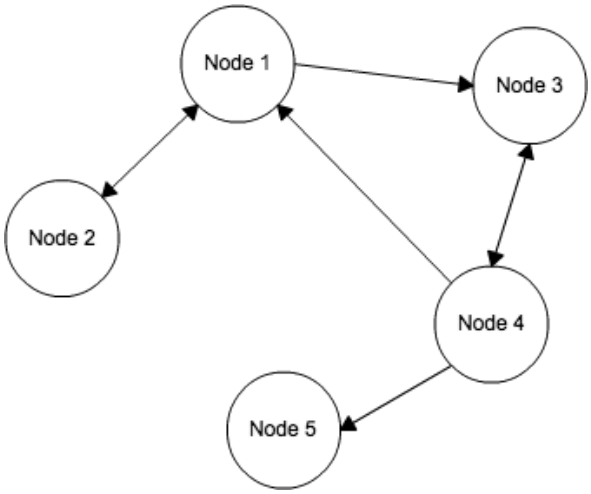

Figure 6.1: Reference Graph

Breadth-first Search

The Breadth-first Search is a straightforward algorithm, even if repetitions are turned on this algorithm will never repeat a node, simply because of the nature of the algorithm. For example during our test with Figure 6.1, the path we obtained between Node 1 and Node 5 was 1 - 3 - 4 - 5. What this does musically is shorter length songs without any connection repetition, and by rule paths obtained by Breadth-first are always the shortest paths.

Depth-first Search

The Depth-first search unlike the Breadth-first presents a much more varied array of solutions, and is not always the best path solution (if repetitions are turned on). For example

using the graph (Figure 6.1) above and setting Node 1 and Node 5 as our first and end node respectively and a repetition of 3, we would get the following path: 1 - 2 - 1 - 2 - 1 - 3 - 4 - 3 - 4 - 3 - 4 - 5. Notice how 1 and 3 repeat three times before moving forward towards a new node, what this does musically is create a connection repetition still managing to reach the end. Unlike the Breadth-first, this algorithm can create node repetitiveness, however this heavily depends on the type of graph. A good example is taking the same graph (Figure 6.1) and create a new connection between Node 2 and 5. If this was the case the outputted solution would be 1 - 2 - 5 or 1 - 2 - 1 - 2 - 1 - 2 - 5, depending on the nodes position in the search tree.

Musically this graph is unpredictable and predictable, what we mean by this is that its predictable because we know it will reach our final node (if a solution is available), but unpredictable because once we listen we're never quite sure where it will go next.

Random Search

The Random Search is a chaotic search algorithm, it will simply choose a random node and travel towards it no matter how many times. By default the random search can only travel to as many connections as there are nodes, however the user can define how many connections it can travel.

Depending on the graph, this algorithm can generate a new path every time and its unpredictability can be its biggest asset. Musically this algorithm is completely unpredictable, reaching the end node is already quite a feat. During our experiments we had cases where the music was extremely long with over 200 drum patterns and other cases where it was the complete opposite, using the same graph.

Traveling Salesman with 2-Opt

The Traveling Salesman path is also an interesting path option, and is clearly the most complex out of all the others. Although we did find this algorithm was the easiest to use, it only required nodes and everything else took care of itself.

Musically the song is very long, because it includes all the nodes from the graph. The more nodes a graph has, the bigger the length of the song. This is also true for its processing time, which can be a burden, especially if its the genetic algorithm. This is why we suggest the hill climbing optimization, as it is the fastest to process, when trying out this path travel.

6.2 User Experiences

We tested the application with four users, two with a musical background and two without a musical background. The User's were allowed to ask for guidance during these tests, as

the interface was not what was being tested, but the musical system, however they were free to return feedback on what could be improved on it.

Each User was given a sheet explaining what was going to be their course of action during the experiments and an explanation of all the features and algorithms available on the system.

6.2.1 Experiment Description

The Users were asked to create a percussion graph with rhythms composed by them. We also imposed some restrictions on the graph creation phase: it should have at least four nodes and a number of connections that were at least half the number of nodes.

Once a graph was created users were asked to try out at least 2 optimization algorithms, one hill climbing and the other genetic in order to listen to each optimizations musical output. We also suggested that a user try traveling the graph using one of the path finding algorithms and the 2-Opt and listen to them. They were also asked to try out the available filters and evaluate them.

Users were asked to discuss and comment as they used the application and at the end of the experiment. What we were looking for in these experiments was the usability of the application as a musical device, if the output and graphic concept made sense, if the available tools were enough, if the algorithms execution time were too long and of course musical appreciation.

6.2.2 User Evaluation

The application was tested with four users who have never used the system. During the entirety of the experiments users had complete control over the system and could freely ask any questions, if they felt the need to. For the purpose of this experiment the genetic algorithm was set to maximum of 100 possible generations (to reduce wait time), while both algorithms stoppage rank was set to 10 (if an individual ranked 10 appears the algorithm stops).

User 1

This particular user has been a musician for almost a decade and plays the english concertina, the portuguese ukelele and the guitar.

During the graph creation process, the user took some time into creating the rhythms into something that he'd like and created four nodes with one connection going through every created node, (node 1 to node 2, node 2 to node 3, etc.), which would allow only one possible path solution. The user enjoyed the system, however he did comment that

the interface could undergo a few tweaks and a much nicer appearance, as it was a bit crude and not very robust.

After the optimization process the user commented that it was still relatively ”quick”, although it could be a bit of a hassle after the nth optimization. He also wondered if creating a larger graph would have taken longer, to which we answered that yes, if the number of connections would be higher. The User listened to a few connections and was quite surprised with the output, as some of the rhythms generated were much to his liking, even suggesting that it would be a nice tool for the single musician jam session.

We asked the user to create a path song, which he did generating a simple path from node 1 to node 4, using the BFS search. The user commented that it was a pretty useful feature, as it helps define a sequence of rhythms and randomizes passages from one rhythm to the next, however the user did suggest that the interface should show the connection that is currently being played on the graph, a bit like a map that shows the progression of the song on the graph.

The User was now asked to generate the genetic optimization on the connections. This process took significantly longer than its hill climbing counterpart, which the user said was something that would probably need to be fixed. After hearing a couple connections the user was still impressed, however he commented that it had a lot of repetitiveness and the changes were a lot more rough than its hill climbing counterpart, though it was not necessarily a bad thing.

We commented how the user could use filters to eliminate repetitiveness and edit some of the connections, which he then went and tried on a few connection. He thought it was very interesting and gave much more control over the connections.

During the final discussions the user overall, did enjoy the system very much, he thought a system like this showed a lot of promise and should be continued, however more investment should be made into the interface and a more complex drum machine (musical interface) such as more measures and options.

User 2

This user has also been a musician for a decade and has been playing the electric guitar since the age of 15.

We first asked the user to create a percussion graph. The user took some care in creating his rhythm nodes as it was probably what took the largest chunk of time of the experiment. The user ended creating five nodes, all with at least one connection with Rhythm 0 branching a connection with Rhythm 1 and 2. The user commented that the percussion graph was a pretty good idea to represent the passage of rhythms and that branching rhythms was a interesting concept.

After the Hill Climbing optimization, the user was asked to listen to some of the

connections and comment. The user enjoyed some of the rhythms the algorithm created and the fact that there was no repetitiveness, however he did comment that some of the middle solutions were a bit chaotic.

The user was also asked to create a path using one of the path finding algorithms available. The user chose the BFS search and after listening to some part of the song thought it was an interesting concept and would be a good jam instrument, if various graph paths could be stored and played ”a la carte”.

The user also tried the 2-Opt TSP option, which he thought did take a bit to generate. Though the global concept is a good idea, the user explained that he had enjoyed more the path option as the wait time was far less, and the user felt a bit lost while listening to the 2-Opt solution, as it had a vast amount of content.

Finally the user was asked to use the genetic optimization. Once the user listened to a connection using this algorithm he immediately said that he was enjoying it more than the hill climbing method. Specifying that the drum pattern changes were much smoother, and retracted his earlier comment on repetitiveness which was actually improving the song more than hindering it. The user was also happy to use some of the filters, which he felt was a good way of handling song editing if they ever got to chaotic.

During the global appreciation discussion, he commented how he enjoyed creating the musical graph and that it was a nice concept for representing music. As for the algorithms, the user thoroughly enjoyed the genetic algorithm and suggested adding more rule specific algorithms such as creating objective functions who could rate generated rhythms based on particular music styles. Other user suggestions were, improving the interface, more specifically having the drum machine in a window with the rest of the interface and implement a .mp3 or .ogg exporter that would allow exporting generated songs or samples from the system.

User 3

This was the first experiment we were going to do with a user who was not a professional musician, although this user has played the guitar, but only leisurely.

The user started by creating four nodes, which were relatively hastily created, and connected each of the nodes, coincidently just like User 1.

The user was asked to use the hill climbing optimization and then listen to some of the connections. After hearing a couple of connections, the user enjoyed a lot of the solutions, but some of the middle rhythms sounded a bit chaotic and out of place, commenting that it also might be from some of the rhythms the user had inputted.

The user jumped straight into the 2-Opt, which he thought was pretty interesting as it gave a global option of using every created rhythm on the drawing board. The user suggested that it would be a nice feature to save these global songs or even samples from

it, much like the suggestion from user 2.

The user was then asked to use the genetic optimization and listen to at least 2 connections. One of the first comments made by the user was that the music sounded much more sporadic than its hill climbing counterpart and that repetition was a bit more abundant. The user then applied a repetition filter on one of the connections and reheard the filtered connection, which he thought had bettered substantially. In the end however, the user preferred the Hill Climbing Method.

In the end the user was pretty satisfied with the application, where the main complaints were for the user interface. In terms of musical appreciation, the user thought even though some solutions were a bit chaotic, the application did show a lot of promise.

User 4

This user does not have a musical background and has never played an instrument.

Like all the other experiments, we asked the user to create his percussion graph. The user took especially care of creating his nodes and experimented with the system quite extensively, even going so far as creating a node and deleting it. The user also took care of how the nodes were connected, by listening to his creations and connecting accordingly. Interestingly, the user created four nodes and connected them just like user 1 and 3, however he then connected the last node with first node creating a circular graph. During this process the user seemed genuinely satisfied in creating his percussion graph, as he tested with the drum machine and rearranged the nodes on the drawing board.

The user was then asked to optimize his connections and listen to the generated content. The user felt that in some connections the music didn't stray much from his original rhythm, while on other connections the algorithm generated some pretty chaotic patterns. We suggested that he could use a filtering option to eliminate those unwanted rhythms, which he tried on various connections.

The user was then asked to try out the 2-Opt option. Once the user listened to the result, he felt a bit overwhelmed with the quantity of rhythms presented and lost because he didn't know where the rhythm was progressing towards. We asked if an indication on the drawing board, symbolizing the rhythms current location would help the user, which he replied that it would definitely help. The user also suggested that in the drum machine, notes that appear in the nodes should be distinguished from generated notes, so as to visualize what the system is doing.

For the genetic part of the experiment, the user wanted to create a brand new graph. The user while very experimental with his rhythms, never created more than one connection per node. Once the user listened to his new connections he felt that the genetic brought more to the table then its hill climbing counterpart, however he was quick to say that it may have been because of the more complicated rhythms he had created with the

hill climbing method. The user explained the difference was that the genetic was a much more unexpected method, where changes could happen much more drastically then the hill climbing, but at the same time it felt controlled and natural.

During the final global appreciation of the application the user also explained that the hill climbing method was also a bit unpredictable when in the process of optimizing. For example in the genetic optimization no matter what the results are the user will always know that once it reaches 100 it will stop, while in the hill climbing method the algorithm only stops once it reaches 10, making the wait much more unpredictable. The user felt that the filters were also an important part of the application, as it also helped the user keep some control over the rhythms and trim them to his liking. This user also suggested that saving rhythms and loading them would be a useful future asset for the application. All in all, we feel that this user genuinely enjoyed his time with the application.

6.2.3 User Experiments - Post-Mortem

It was particularly interesting conducting these user experiments as certain ideas were shared among different users and it was interesting to analyze these tendencies. Given that four users is far from a total estimate, it does give us a glimpse of the prospect of the application and not to mention the vast number of suggestions that we will take into account, if future iterations of the research persists.

It was particularly interesting how 3 out of 4 users decided to make direct graphs, each node having a single connection with no altering paths and going from one node to the next. Even though the users were never restricted to how their percussion graph should be created (apart from the minimum number of nodes and connections), it seemed that their tendency was to prioritize creating a graph path that could maximize all of their created nodes to the fullest.

Another particular interesting point was that the user who seemed to enjoy the application the most was not even a musician. His interest came mostly from experimenting the drum machine and then listening to the generated results. Even though the other users did enjoy the system, we felt that this user thoroughly liked the application and what it had to offer.

Other particularities were most users favored the genetic method except one user, which interestingly was the only one who gave less thought about his created rhythms. Interestingly users did enjoy some repetition in their songs, which went against some of our initial user research data, and it was one of the reasons why our genetic algorithm was so favored. Musically the system did quite well apart from a few hiccups here and there, some users were in fact quite surprised with what the system had created.

The biggest complaint from all the users was of course the user interface, as it still is considerably crude and hard on the eyes, so to speak. Even though this was not a usability experiment and research we did take some of these criticisms into consideration. Ideas

such as following the graph path as the song plays was a particularly good idea. The pattern differentiation idea, which consists of coloring notes from user created patterns differently from notes that were generated, allowing the user to visualize the convergence of a song. Another highly requested feature was the system having an ability to export songs into .mp3s, so that users would actually be able to use rhythms that were generated on other projects.

All in all, we believe that all the experiments went well and that users genuinely enjoyed their experience with the system, which convinces us that there might actually be some future in the percussion graph idea.

Chapter 7

Conclusions & Future Work

7.1 Post-Mortem

Looking back at our objectives and what we had set out to accomplish, we believe that we were able to achieve these goals, even though the musical output wasn't always perfect, most of the time it was actually quite surprising and very musically sane. Which was our first objective that we had set out to accomplish.

We also believe that we were able to achieve our second goal of integrating the user into the music creation process, while still maintaining a generative and automated music approach. The fact that users were able to influence the generative process through their own creations, was something that went exactly with what he had set out to accomplish.

Finally one of our last objectives was to create a system that felt complete, with enough features that could create enough musical diversity and be compelling and interesting. As a whole we believe the system we developed was quite complete, three optimization processes, three path finding algorithms, one traveling salesman algorithm, the graphical interface, the drum machine, filters and path options. Pretty much everything that we wanted to implement since the early conceptualization of this idea.

We would also like to note that the whole conceptualization phase was one of the most creative processes and most fulfilling of this entire research. The project really allowed us to think outside of the box and simply test concepts and solutions that we had never tried before.

Personally we think that the weakest part of the system was the genetic algorithm, even though it was a great success with most users it didn't accomplish what we wanted it to accomplish, the algorithm was slow, complex, memory heavy and most of all it simply didn't converge all that well. In retrospect this was the algorithm that took us the most time to work on and it really didn't come together as well as we were expecting. Although we might be a bit harsh considering that musically the Genetic proved to be the favorite amongst users for its erratic behavior and originality, and in the end what we were really exploring was the musicality of the algorithms more than its computational performance.

The other problem specifically considering the genetic algorithm was that an external API was used and even though it saved us immense amount of time, we wonder if different results would have been obtained performance wise, if we had chosen another API or created a genetic algorithm from scratch.

The Hill Climbing and the Stochastic Hill Climber had a much better fare performance wise than its genetic counterpart. The stochastic hill climber was actually one of our personal favorites since the musical output was so undeterminable and the shifting between rhythms flowed so naturally.

In terms of path traveling, the start and end node travel proved the most successful among us and the users, simply because the TSP travel took too much time to generate a solution, and the generated solution was way too big even with the 2-Opt optimization. Due to the exaggerated number of rhythms in the TSP, we felt that we had lost a bit of control over the content, not to mention a bit lost within the music itself. We wonder if this could have been fixed with a ”current connection being played” reference on the percussion graph, while the TSP solution was being played.

The user experiments were the highlight of the whole research, we believe that most users were underestimating the potential of the system and were quite taken by surprise when the application came up with rhythms that weren't just a cacophony of percussion sounds but were actually real coherent drum patterns. Not to mention that most users shared some pretty good ideas on how we could improve the current system. We were also pleasantly surprised by how easily the users understood the graph concept and what we were trying to convey visually, as we had some reservations to how the whole graph concept would actually be viewed in practice.

All in all we believe that we succeeded in our venture, even though it wasn't the perfect application it did show some promise, especially after hearing the user reactions. If the idea had been conceptualized right at the beginning of the year, we believe that the application would have been free of some of the quirks it finds itself having, but as a first take on a system of this type we are proud of what has been achieved.

7.2 Future Work

Considering that this system was still in its prototype phase, we believe that there is still a lot of work and improvement to be had.

The main improvement that comes to mind would definitely be the interface, something that would be much more appealing for the user. Notable ideas include:

- Keep the Drum Machine and Graph Interface in one single window.
- Improve User Feedback, for example showing the time it will take to evolve a connection.

- Improve the Graph Interface, a lot more robust and less ”clunky”.
- Insert a Save / Load Graph Option and a Sample Export Option.
- Improve the Filter Interface to something much more intuitive.
- Create a Rhythm Library with pre-created rhythms and user created rhythms, where the user can simply drag these into the drawing board creating nodes.
- Support a much more complex drum machine.
- Graph traveling feedback, the user would be able to see on the graph, the position of what was currently being played on the drum machine, when traveling a path.

The Genetic Algorithm is also something that given the time, we would improve upon. We believe that there is still room for this algorithm to show its true nature and given the time it would be a great asset for the system.

Another improvement we thought about was implementing Heuristic Search Methods for Path Traveling between two nodes, such as Best First. The heuristics could simply be the objective function of our optimization process.

We also thought that the initialization time on the TSP 2-Opt travel would certainly need improvement, as generating a fully connected graph can be a tedious experience for the user.

Appendix A

The Horowitz Prototype

This system was created as part of our prototyping phase and consisted of an analogous recreation of the Horowitz's Generating Rhythms with Genetic Algorithms[Hor95] paper.

A.1 Tools

To create the Prototype, NetLogo[Wil][TW04] was used for its easy accessibility, fast prototyping and sound extension. As a basis for the drum machine, the Netlogo model library presented some pretty good startups, and in this particular case the drum machine model was a great framework to build upon.

A.1.1 NetLogo

Netlogo is a multi-agent programmable environment, with a simple programming language and interface, which brings an easy to use and fast creation mantra towards its users. It's particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of "agents" all operating independently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals.

How Netlogo works is relatively simple, in the main interface the user is presented with a black window, this window is called the patch map. This patch map is where the agents (known as turtles) will appear. The user then has the ability to manipulate the interface such as add buttons, sliders, etc. which will influence the backend programming (much like Visual Basic). Once an interface is designed the user then programs the various functionalities in the procedure's tab. The programming language is an extension of Logo that is capable of supporting agents.

An example of the advantage of such a system is the ability to quickly create an interface and define a population.

A.1.2 The Drum Machine Model

As a basis for this prototype we used an existent drum machine model found in the NetLogo Library, which really saved us precious time. The model simulates a drum machine, where the user can create drum patterns in one 4/4 measure.

The patch map (previously mentioned) which in the case of the drum machine, is neatly separated and organized by note type, in this case each square represents an eight note which means in a 4/4 measure its 8/4 of a regular note (in a 4/4 measure a note is 4 beats) and percussion instrument, which are a combined number of 47 instruments. Time is on the y axis (vertical lines) while instruments are on the x axis (horizontal lines). The user than applies the notes (a random colored square) atop this grid choosing exactly what and when a percussion instrument should play in the measure.

The drum machine will loop infinitely, playing the drum pattern using a white vertical line that represents the current time in the measure. Each time the line intersects with a note it plays the instrument, and once it reaches the end of the measure it restarts from the beginning.

The user has the ability to turn up or down the tempo of the music (making the current time marker move faster or slower), turn the velocity up or down (the emphasis which the instruments are played), rewind to the beginning, save the drum patterns, load drum patterns.

A.2 The Horowitz Drum Machine

This section will detail the specifics of the Horowitz prototype, mainly the implementation of the genetic algorithm.

A.2.1 The Beat Map

Colored squares represent drum hits. The world is a ”map” (Figure A.1) of the composition in which time moves from left to right. Each row of patches is a particular drum, and each column represents a moment in time.

A.2.2 Defining the Chromosome

An individual’s chromosome, a list of notes of the ”beat map” (Figure A.2), are represented by coordinates that map details such as, which drummer (instrument) plays the note and at what time he play’s it. Un-played notes are not represented in the chromosome, so notes that aren’t within an individual’s chromosome are, by default, not played by the drum machine. The population is of course none other than the collective of chromosomes of the current generation.

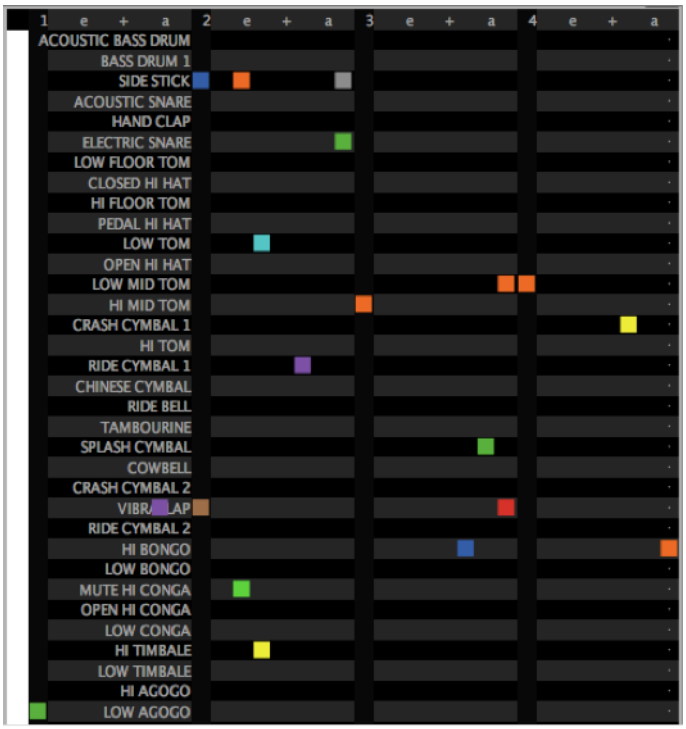

Figure A.1: The Beat-Map

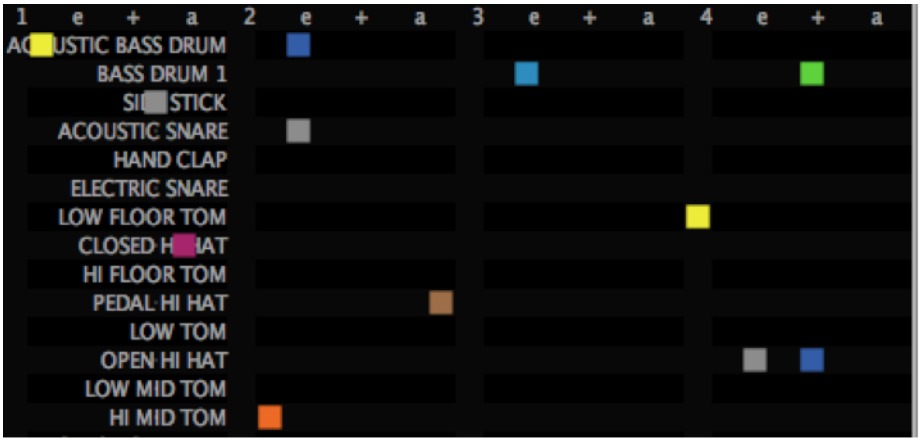

Figure A.2: Representation of One Chromosome

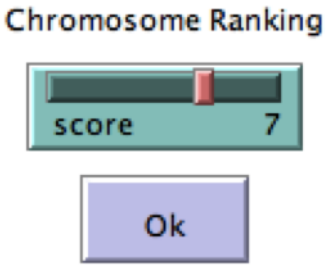

Figure A.3: Fitness Selection Screen

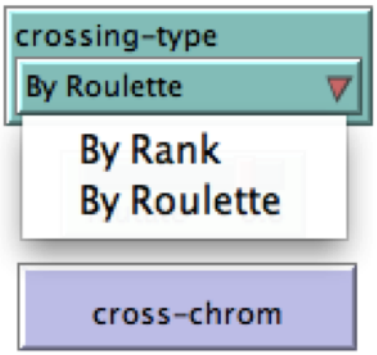

Figure A.4: The Selection Method Option

A.2.3 Applying Fitness Values

Fitness is obtained directly from the user (Figure A.3), who evaluates each beat and ranks them from a scale of 1 to 10. This method of fitness selection is relatively subjective and goes towards the likes and dislikes of each individual user, who might appreciate a certain rhythm more than another.

These fitness values are then taken into consideration in the selection method, so that a higher ranked individual is more likely to procreate than a lesser-fit individual. The advantage of this system is that the music will evolve more likely towards the users musical taste, but at the same time not limit the evolution to a few individuals.

A.2.4 The Selection Method

As for the selection method of the chromosomes, there are two available choices, the Roulette Selection or the Rank Selection (Figure A.4).

A.2.5 Elitism Modifier

The Elitism modifier also plays a role in the selection process, as it forces the genetic algorithm process to keep the best-ranked N number of chromosomes at each generation[Mit98].

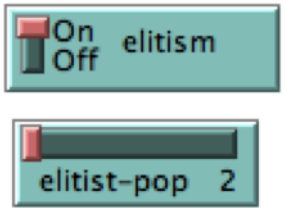

Figure A.5: The Elitism Modifier

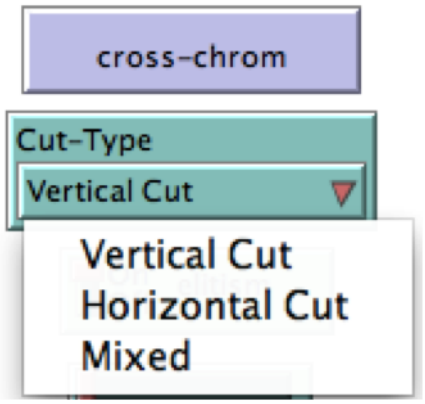

Figure A.6: The Cut Selection Options

It is to note that these individuals can be lost if they are not selected in the next generation or if they are destroyed by mutation or crossover (Figure A.5).

A.2.6 Crossover Method

The first consideration to be taken when applying the crossover, is defining a cutting point, so that it is possible to cross the two different chromosomes evenly so that a new generation may arise. As previously mentioned a chromosome in the application is a collection of musical notes that are represented within the ”beat map”, each note details the drummer and time of what and when a sound should be played (Figure A.6). To avoid streamlining and hardcode the generative process, the ability was given to the user on how the chromosomes should be split.

The Vertical Cut

The vertical cut is when two chromosomes are split vertically (by looking at the beat map), or more precisely by the time in which a note is played. Using this process the played instruments aren’t changed, or more appropriately the new generations will share the same instruments from the previous generation, though the time in which these instruments are played are most likely to be changed due the crossing of the chromosomes (Figure A.7).

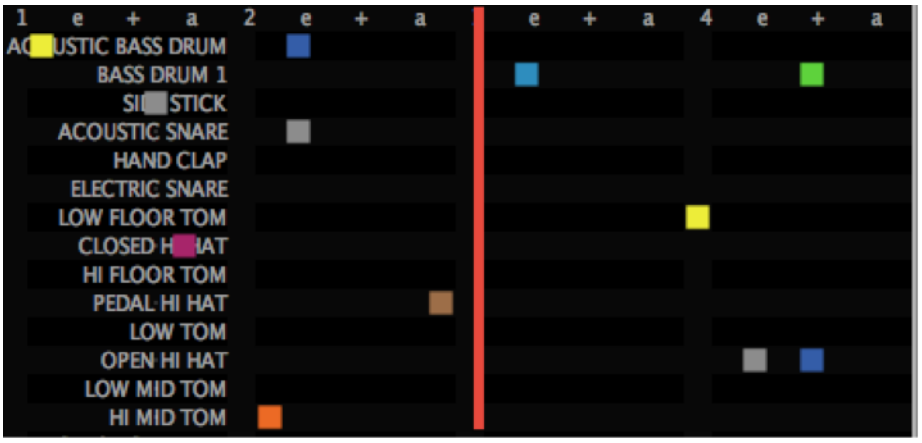

Figure A.7: The Vertical Cut

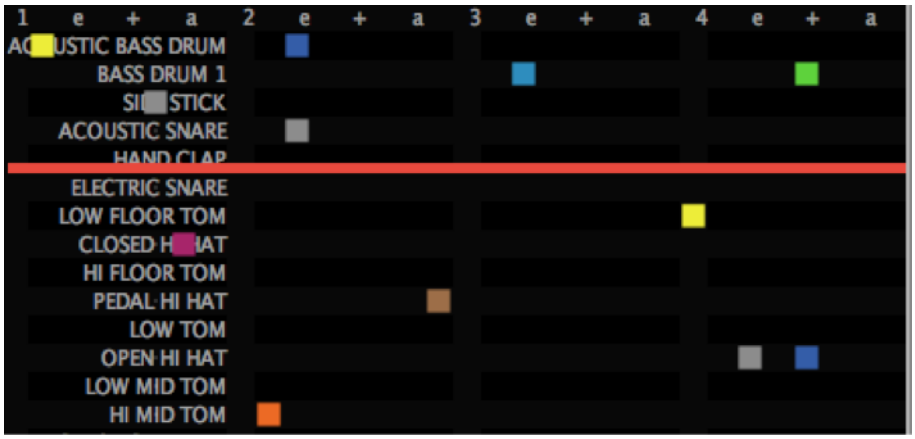

Figure A.8: The Horizontal Cut

The Horizontal Cut

The horizontal cut is the opposite of the vertical cut, while in the vertical cut the chromosomes are split by their location in time, in the horizontal the chromosomes are split by the instruments they play (or more specifically the location of the instruments on the beat map). In practical terms, what this means is that the time in which the notes are played will most likely be shared between the next generations and the previous generations but who plays those notes are more likely change (Figure A.8).

The Mixed Cut

When the horizontal or vertical cutting types are chosen, it will apply the same chosen crossover method on the entire population. By selecting the mixed type cut, the application will randomly choose between the horizontal or vertical method for each crossover within the population.

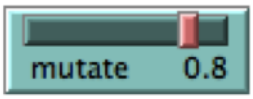

Figure A.9: The Mutation Chance

A.2.7 Mutation

Every new generation chromosome has a chance to mutate. This percentage is defined by the user, and can go from 0% to 100% mutation per new generation. When a mutation occurs, a chromosome might gain or lose one random note or replace a note with another one (Figure A.9).

A.2.8 The User Interface

It was necessary to create an interface that gave the user the ability to control certain aspects of the drum machine. Even though the application is all about generative music the user still controls and influences some key aspects of the final music (Figure A.10).

The user has the ability to:

- Set the maximum number of chromosomes to the initial population;
- Define the number of notes of the generation 0 chromosomes;
- Cycle through each chromosome from a given population;
- Rank each chromosome of a given population;
- Define the crossover type;
- Set elitism selection on/off and number of elitist chromosomes to save for next generation;
- View the number of the current generation and the current chromosome loaded in the drum machine;
- View plotting information about the average fitness and the chromosome with the highest fitness from previous generations;
- The ability to change the tempo and emphasis (velocity) of the instruments.

After the initial prototyping and testing, it was pretty obvious that the interface was a little too cluttered and deserves a refining and fine tune in the next iteration of the application.

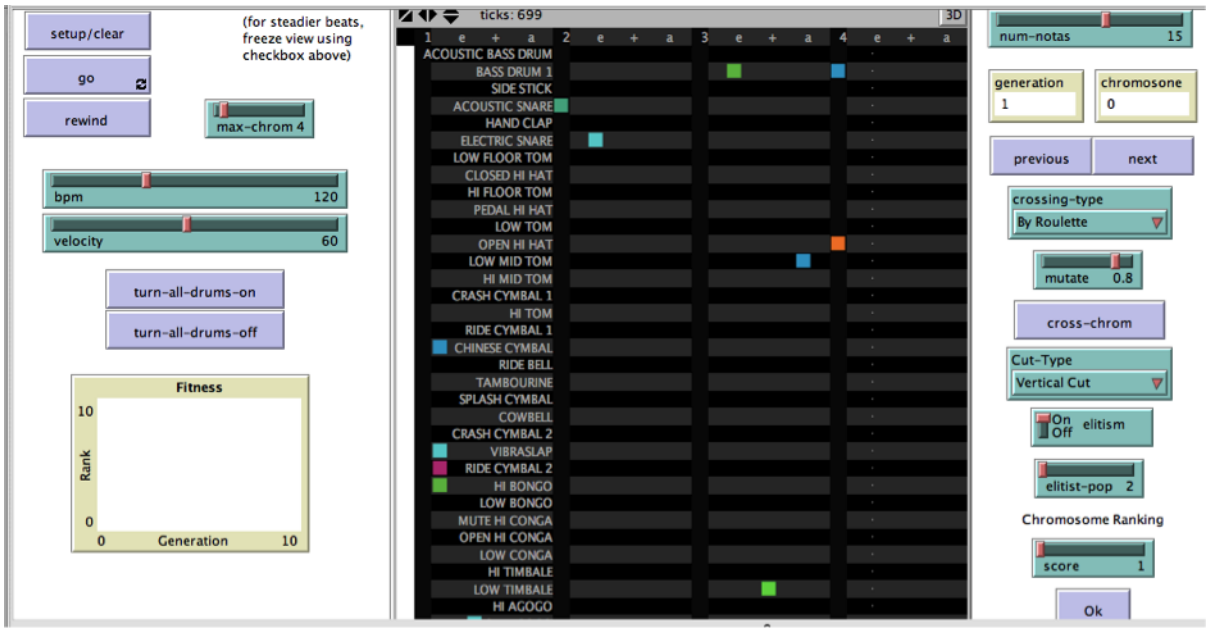

Figure A.10: The Drum Machine Interface

A.3 Using the Horowitz Drum Machine

The interface of the drum machine is likely to be overwhelming at first glance (Figure A.10) but is quite easily used. To startup the application, use ”setup/clear” button, this will jumpstart the application with the default setting of notes per chromosome and max population. If the user wishes to change these settings, one can do so by only changing the number desired of notes in the ”num-notes” slider and the number desired of population size in the ”max-chrom” slider. This will create the generation 0 chromosomes.

To listen to each beat, the user must click on the ”go” button. This will start playing the beat; a user can cycle through the different beats (chromosomes) of the population by using the ”next” and ”previous” buttons found in the upper right corner of the interface. The monitors will always show the current generation and the current chromosome the user is listening to. Applying fitness to a particular chromosome the user needs to use the slider score in the lower right corner of the interface and set it to the rank he feels suits best the current chromosome. Clicking on ”Ok” will lock the score to that particular chromosome.

Once the user is satisfied with the scores given to each chromosome (it should be noted that the default score for each chromosome is 1), it’s time to create a new generation. Before anything a user must define which selection method is used (Roulette or Rank), which cut type will be applied in the crossover (Horizontal, Vertical or Mixed), if elitism is applied and if yes, how many will be kept from the last generation (note a new generation will always have at least 2 new chromosomes made from crossover), and finally the mutation percentage that is applied to the ”new born” chromosomes. Once

everything is defined, the user only needs to press the ”cross-chrom” button and a new generation population will be born.

Other options are available to the user like the ability to change the tempo of the beats, the emphasis of the notes (dubbed velocity), turning drummers on and off and the visualization of fitness information in the plotting window.

Bibliography

[BE05] D. Burraston and E. Edmonds. Cellular automata in generative electronic music and sonic art: a historical and technical review. *Digital Creativity*, 16(3):165, 2005.

[BELM04] D. Burraston, E. Edmonds, D. Livingstone, and E.R. Miranda. Cellular automata in midi based computer music. In *Proceedings of the International Computer Music Conference*, pages 71–78. Citeseer, 2004.

[BG05] O. Braysy and M. Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

[Bil94] J. Biles. Genjam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, pages 131–131. Citeseer, 1994.

[Bil05] Al Biles. Evolutionary music. *Artificial Life*, pages 1–22, 2005.

[Bla03] TM Blackwell. Swarm music: improvised music with multi-swarms. *Artificial Intelligence and the Simulation of Behaviour, University of Wales*, 2003.

[Bro05] A. Brown. Exploring rhythmic automata. *Applications of Evolutionary Computing*, pages 551–556, 2005.

[BS02] J. Bachrach and M.I.T.A.C. Seminar. Beatrix: The amorphous drum ensemble v3. 2002.

[BV99] A.R. Burton and T.R. Vladimirova. Generation of musical sequences with genetic techniques. *Computer Music Journal*, 23(4):59–73, 1999.

[Cro58] GA Croes. A method for solving traveling-salesman problems. *Operations Research*, pages 791–812, 1958.

[Dar58] C. Darwin. *The origin of species*. Number 811. Hayes Barton Press, 1958.

[Daw76] R Dawkins. *The selfish gene*. Oxford University Press, 1976.

[Eig10] A. Eigenfeldt. Coming together: composition by negotiation. In *Proceedings of the international conference on Multimedia*, pages 1433–1436. ACM, 2010.

[EP09] A. Eigenfeldt and P. Pasquier. A realtime generative music system using autonomous melody, harmony, and rhythm agents. In *12th Generative Art Conference GA2009*, 2009.

[GM07] M. Geis and M. Middendorf. An ant colony optimizer for melody creation with baroque harmony. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 461–468. IEEE, 2007.

[GMJ05] M. Gimenes, E.R. Miranda, and C. Johnson. A memetic approach to the evolution of rhythms in a society of software agents. In *Proceedings of the 10th Brazilian Symposium on Computer Music (SBCM)*, 2005.

[GMS04] C. Guéret, N. Monmarché, and M. Slimane. Ants can play music. *Ant Colony, Optimization and Swarm Intelligence*, pages 310–317, 2004.

[Hor95] D. Horowitz. Generating rhythms with genetic algorithms. In *Proceedings Of The National Conference On Artificial Intelligence*, pages 1459–1459. JOHN WILEY & SONS LTD, 1995.

[Hyd] Hydrogen. Hydrogen - advanced drum machine for gnu/linux. Website `http://www.hydrogen-music.org/hcms/node/395`. Visited in May 2011.

[JP98] B. Johanson and R. Poli. Gp-music: An interactive genetic programming system for music generation with automated fitness raters. *Genetic programming*, pages 181–186, 1998.

[Kau] P. Kauppila. Monkey machine - the online drum machine. Website `http://rinki.net/pekka/monkey/`. Visited in May 2011.

[Koz96] J.R. Koza. *On the programming of computers by means of natural selection*, volume 1. MIT press, 1996.

[Lad] C.K. Ladzekpo. Foundation course in african dance drumming. In Web Course `http://home.comcast.net/~dzinyaladzekpo/Foundation.html`. Visited in June 2011.

[Lim] Natural Studio Limited. Natural studio. Website `http://www.naturalstudio.co.uk/`. Visited in May 2011.

[LS11] J. Lehman and K.O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

[Mef] Klaus et al. Meffert. Jgap - java genetic algorithms and genetic programming package. Website `http://jgap.sf.net`. Visited in May 2011.

[MF04] Z. Michalewicz and D.B. Fogel. *How to solve it: modern heuristics*. Springer-Verlag New York Inc, 2004.

[Mir01] E.R. Miranda. evolving cellular automata music: From sound synthesis to composition. In *Proceedings of the Workshop on Artificial Life Models for Musical Applications-ECAL*. Citeseer, 2001.

[Mir03] E.R. Miranda. On the music of emergent behavior: What can evolutionary computation bring to the musician? *Leonardo*, 36(1):55–59, 2003.

[Mit98] M. Mitchell. *An introduction to genetic algorithms*. The MIT press, 1998.

[MOŠŽ07] A. Misevičius, A. Ostreika, A. Šimaitis, and V. Žilevičius. Improving local search for the traveling salesman problem. *Information Technology and Control*, 36(2):187–195, 2007.

[Moz94] M.C. Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. In *Connection Science*. Citeseer, 1994.

[MVW+03] B. Manaris, D. Vaughan, C. Wagner, J. Romero, and R. Davis. Evolutionary music and the zipf-mandelbrot law: Developing fitness functions for pleasant music. *Applications of Evolutionary Computing*, pages 65–72, 2003.

[RN02] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd Ed.* Prentice Hall, Englewood Cliffs, NJ, 2002.

[Smi91] Joshua R. Smith. Designing Biomorphs with an Interactive Genetic Algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 535–538. Morgan Kaufmann Publisher, July 1991.

[TI00] N. Tokui and H. Iba. Music composition with interactive evolutionary computation. In *Proceedings of the 3rd international conference on generative art*, volume 17, pages 215–226, 2000.

[TW04] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *in International Conference on Complex Systems*, pages 16–21, 2004.

[Wil] U. Wilensky. Netlogo. Website `http://ccl.northwestern.edu/netlogo/`, Center for Connected Learning and Computer-Based Modeling, Northwestern University. Visited in October 2010.

[YK07] M.J. Yee-King. The evolving drum machine. In *Music-AL workshop, ECAL conference*. Citeseer, 2007.