University of Aarhus
Department of Computer Science
Ny Munkegade
8000 Århus C
Denmark

May 2003

Automatic Discovery of Parallelism and Hierarchy in Music

Master's Thesis

Søren Tjagvad Madsen
Martin Elmer Jørgensen

Abstract

There are three main ideas in this thesis.

Graph representation for non-monophonic music

We have attempted to create a datastructure for the symbolic representation of music. Based on the two most fundamental temporal relations found in music scores, precedence and simultaneity, we have developed a *graph representation of music*. A music graph may represent both monophonic and non-monophonic music. The music graph is a powerful and flexible abstraction that can be extended considerably to represent almost any symbol found in notated music scores. It is our attempt to break out of the sequential/parallel dichotomy used in so many representations in the literature, while still basing the representation on precedence and simultaneity. We argue that a graph representation is a more elegant representation, because it doesn't necessitate repeated restructuring of the data structure switching between the two fundamental modes (sequential or parallel structures) during the analysis and segmentation of music.

Search for musical parallelism

To search for similarities in music graphs, we have implemented the *SimFinder system*. The SimFinder uses a genetic algorithm to search for similarities in a music graph according to different *similarity measures*. Whereas much of the literature considers 'only' monophonic music, the SimFinder may search a music graph for either monophonic or non-monophonic similarities. We have experimented with several similarity measures for both monophonic and non-monophonic similarity. The SimFinder is a modular framework that allows new similarity measures to be defined and easily substituted for the ones we have defined. Similarity measures are built using a *multiple viewpoint system* that allows us to analyse music from a multitude of perspectives at once. By designing and mixing the right viewpoints we are able to specify what we are searching for. The SimFinder is designed so that many different kinds of viewpoints may be combined and tested experimentally.

Segmentation using a graph grammar

We describe a selection of past uses of formal grammars for the description of music. Just how powerful a grammar must be to be able to adequately describe music is a subject of debate, but it seems that something more than a context-free grammar is needed. We argue the importance of hierarchy in music and present a segmentation algorithm called the SimSegmenter for non-monophonic music, which is based on the SimFinder. The SimSegmenter builds a *graph grammar* to hierarchically describe a piece of music represented in a music graph. We present some test runs on fugue and chorale subjects by J.S.Bach.

Keywords: automated music analysis, non-monophonic music, multiple viewpoint systems, hierarchy and formal grammars, musical similarity and parallelism, graph representation of music, symbolic representations of music, graph grammar.

Contents

1 Introduction

Most music-loving people know the feeling that "I've heard this passage before, but where?" The situation illustrates our ability to recognise similarities between music passages or pieces, that may be structurally very different – an ability that seems so natural and easy that it is all the more frustrating to realise how hard it is to simulate human analysis of music using a computer program. But computer scientists love tough problems, and we in particular are also very fond of music.

An automated analysis of music faces several non-trivial problems. Music is multi-faceted and multi-layered. A thorough analysis of a piece must take into account metrical hierarchies, grouping and reductional hierarchies, tension-relaxation relations, phrasing and harmonic hierarchies, and musical parallelism. These interact in complex ways, and it is a challenge to isolate a single sufficiently independent area to concentrate a study on. The interconnectedness also has a price in computational complexity that must be coped with in an analysis program.

Most music is non-monophonic. Many studies in computational musicology have concentrated on monophonic music because it is easier to describe sequentially in formal structures. Once we attempt to extend an analysis to non-monophonic music, the computer representation of the music is prone to the "sequential-parallel dichotomy", which constricts subunits in hierarchical structures to be sequences or simultaneities. This is not ideal. There is a challenge to find a representation that is flexible and strong enough to be suitable for the analysis of non-monophonic music.

Musical knowledge is often tacit knowledge. If humans can be said to process musical input, many of us learn to do so simply by repeated exposure to music, either played live or recorded. Many constraints and conventions in Western tonal music can be learnt in this way without having been taught or formulated. There is thus a formalisation challenge, when we strive to achieve computability.

These are three major challenges to an automated analysis.

In the 1970s and 1980s, computational musicology took what we call a 'linguistic turn'. The inspiration came from generative grammars, a tool developed in linguistics. The most famous book on music and generative grammars is Lerdahl and Jackendoff's *A Generative Theory of Tonal Music* (GTTM). Innovative at the time, this work remains impressive for its sharpness and tight integration of musical knowledge into formal descriptions of music. However, there are three points where the GTTM could be improved. First, it relies on but does not define musical parallelism. Secondly, it does not give an explicit account of non-monophonic music, and thirdly, the GTTM is not as formalised as to be a computational theory of music analysis. Computability was not the goal of the GTTM, but it is to us.

We have no ambitions of extending the GTTM. Rather, we would like to pick out these three points as interesting problems that need a solution and see how far we can get. Our subject is automated musical analysis on the symbolic level. We thus do not touch upon audio processing. Like Lerdahl and Jackendoff, we have chosen to remain inside the idiom of Western tonal music.

The ideas are the following: a graph representation for the analysis of non-monophonic music, a multiple viewpoint system to search for musical parallelism

in the music graph, and a segmentation algorithm to analyse music in the graph. The segmentation algorithm uses the multiple viewpoint system to locate musically parallel subgraphs in the graph; it then substitutes the subgraphs with simpler elements and proceeds to do this in several layers, giving a hierarchical structure of nested abstractions that are essentially production rules in a graph grammar. The thesis is thus an exploration of how we could build a structural analysis of music using a method that not just incorporates but is *based on* parallelism. These are our attempts to cope with the problems of parallelism and non-monophonic music. As for computability, we have implemented the ideas in Java and tested our ideas on music of J. S. Bach.

Chapter 2 presents the idea of hierarchy in music and a number of uses of grammars for the description of music. Chapter 3 presents some considerations of the computer representation of music and of the concept of musical parallelism. Chapter 4 presents our SimFinder system that we use to search for similarities, and chapter 5 presents our segmentation algorithm, the SimSegmenter, as well as the evaluation of some test runs on a fugue and a chorale subject by J.S.Bach. We think there are a number of interesting possible extensions to the SimFinder system, which we present in the last chapter, chapter 6.

2 Is music grammatical?

We would like to give an impression of some of the ways in which it makes sense to describe music grammatically.

2.1 Musical structure

As listeners we tend to hear music not as one note at a time, but instead we recognise the melodies or sounds that they form. In fact the most prominent task of a conductor or musician is to phrase the music into natural and logically bounded entities, exposing natural sections in the music and thus making the music more understandable to the listener. Of course this process is influenced by human judgement, and therefore there is not one sole way to hear or play a piece of music. But guided by the performers, listeners seem to have a considerable amount of consensus about what belongs together in deciding the extension of the units. In general, it seems that the low level structural units are most easy to agree on (notes, chords, motifs), and the same is true about the high level units (movements and stanzas). The intermediate units are a more prone to differences in subjective judgement.

In *The New Grove Dictionary of Music and Musicians*, Whittall states as a fact that "an organizing impulse is at the heart of any compositional enterprise, from the most modest to the most ambitious." [Whi01] By structured is meant that music is decomposable into smaller units and subunits (structures) – it has a well defined *form*. The organisation is then the division into sections and the relation of those sections to each other. The discussion of musical form is of great interest in music pedagogy: one can learn how musical structures can be put together and thereby gain a deeper understanding of some piece. That structure is inherent in music is witnessed by common musical terminology; there is an entire vocabulary describing musical subunits like notes, chords, motives, themes, phrases, measures, parts, stanzas, and movements etc.

Seen from a historical view, the structured form of music has always been a matter of importance. Some types of organising principles have been more used than others. In popular music, for example, a principle of alternating between verses and chorus has been used for hundreds of years (as for example in Danish folksongs and in contemporary pop music). Composers of art music have developed more elaborate *formtypes* like sonata and minuet. The formtypes are generalisations of the form of a number of actual pieces. The pieces can be logically segmented in their own right, and thereby revealing a common organising principle – the formtype. As time has passed and the tonal language in the 20th century has become more and more complicated, form has been given greater importance. On a lower structural level, one can speak of pieces structured by the composition principles used when composing the piece. The serial music of A. Webern and A. Schönberg is an example of this. Also in baroque music we find structurally important consistent ideas constructing the music. For example, some music, like a fugue, is based on the principle of imitation; most of the musical material can be generated from alterations of a handful of phrases in it. In contemporary music there are also examples of music generated from a limited set of ideas, as for example in the tintinnabuli style of Arvo Pärt or minimalistic music from Steve Reich. Of course many sorts of formshaping ideas have been tested as a basis for shaping music. Even

the golden section principle has been pointed out as an organising principle on many levels of the music, for example in Ernö Lendvais studies of the music of Béla Bartók.

But what does it take for a structure to be noticeable by a listener? There are of course more than one feature involved. One important aspect is parallelism. Parallelism covers repetition of phrases in different variants: transposition, elaboration and simplification. What parallel passages have in common is that they sound more or less similar in spite of the differences in the note material. We have been concentrating a lot on parallelisms, and will return to the topic in section 3.2.3. Another structuring circumstance is the fact that passages can sound initiating or ending, in a tension-relaxation relation, and never in the opposite direction (i.e., ending with tension). Again listeners seem to agree to a considerable extent on what sounds opening and what sounds closing, or on what sounds tensing and what sounds relaxing. To give an example, most people will agree that the first measure in figure 1 sounds as an opening, and that the following measure sounds closing. The phrase cannot end after the first measure, and the second measure would act as a very surprising beginning.

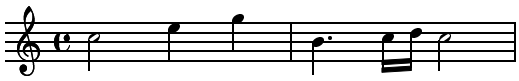


Figure 1: The phrase consists of an opening and an ending.

Another example is that almost every melody in any songbook will end on the root note of the tonic. It seems that anything else does not meet our expectations of a definitive closing. Furthermore passages that are good continuations – which holds the tension between the tension points – do occur. Lerdahl and Jackendoff [LJ83] give further four main structuring features: grouping of phrases, the metrics of the piece, location of more import events and a formalisation of the tension-relaxation based idea. We shall describe Lerdahl and Jackendoff's theory in a little more detail in section 2.4.7.

But one thing is form, and another content. To have an understanding of the organising structure of music does not mean that we are able to simply go ahead, fill out an empty form and write a great piece of music. A well tested structure helps the process along the way, but a whole is that which has a beginning, a middle and an end, and the connection between the elements must not be arbitrary or too obscure. So it is difficult to separate form and content, since they are two supporting sides of the musical piece.

2.2 Hierarchy

Grammars are hierarchical structures. In general, hierarchy is a very important ways of structuring knowledge, and there are hierarchical structures in music.

2.2.1 Hierarchy as an organising principle

In 1962, Herbert A. Simon proposed the idea of a general systems theory, abstracting properties common to complex systems known from physical, biologi-

cal or social systems [Sim62]. The central theme was that "complexity frequently takes the form of hierarchy, and that hierarchic systems have some common properties that are independent of their specific properties." [Sim62, p.468] If this is true, some of Simon's remarks on systems described in physics, biology or in the social and behavioural sciences might be applicable to hierarchies in music too.

A hierarchic system, in Simon's words, is "a system that is composed of interrelated subsystems, each of the latter being, in turn, hierarchic in structure until we reach some lowest level of elementary subsystem." [Sim62, p.468] Such relations between subsystems are known from physics: atoms, taken as elementary particles, may be combined to form more complex systems like molecules, but the atom in itself may also be viewed as a complex system of even smaller parts. In biology, the subsystems taken to be elementary may be organs, tissue, cells, proteins or amino acids, depending on the purpose at hand. Descriptions of social interaction employ terms as 'person', 'colleague', 'boss', 'family', 'friends', which suggest a limited number of relations connected to the individual; not all people communicate with all people, and this grouping of individuals often goes hand in hand with hierarchic descriptions, e.g. of formal organisations. The employees are subordinate to their immediate boss or leader, who in turn may be subordinated a higher scale leader. Symbolic systems exhibit hierarchic structure too, such as the division of this thesis into chapters, sections, paragraphs, sentences etc., and music as a symbolic system also exhibits hierarchy in terms of movements, parts, themes, and phrases.

The use of hierarchy as an organising principle could perhaps be advantageous, giving the upper hand in an evolutionary perspective. Simon illustrates the idea with the parable of the two watchmakers, each assembling watches from one thousand small elementary pieces. Every time a telephone call forces them to put aside the watch being assembled, it falls apart, and the work must start over. But one watchmaker devises a scheme in which bigger stable subparts may be assembled from ten pieces, and even bigger parts of the final watch can be built from ten of these stable subparts. A telephone call then will wreck only the assembly of a ten-piece subpart, not of the entire thousand-piece watch, making him more productive and prosperous in the long run. The central concept is that of "stable intermediate forms", which is what allows us to see a complex subsystem as a unit among other units, or a component of a greater complex system. The power of hierarchy as a perceptual organising principle is perhaps more convincingly illustrated when applied to information processing: a completely unredundant complex structure is its own simplest description. But many structures have some amount of redundancy that may be identified and abstracted away as component structures, thus giving a simpler (hierarchic) description of the complex structure through an appropriate recoding. It is simpler in the sense that we need less information (e.g. measured in number of characters or bits) to describe the same structure. Thus hierarchy can be argued to have strength as a general perceptual strategy, allowing us to understand, describe and memorise more complex systems, and their parts, than would otherwise have been possible. Whether the success of hierarchical description arises because the world possesses so many inherently hierarchical structures awaiting our description or because our perceptual apparatus is by default hierarchically inclined is left to philosophical debate.

The fundamental whole-part decomposition ability is used e.g. in mathe-

matical problem-solving such as proving theorems. We have a desired goal (the theorem) and start out constructing a proof from axioms and known theorems. The search is like finding one's way through a labyrinth, iterating through a series of transformations of the axioms. Some transformations we recognise as useful, leading us closer to the actual theorem we want to prove. This fact makes such transformations, e.g. a lemma, instances of "stable intermediate forms" that the further proof may build upon without starting all over, if the chosen pathway in the proof should nonetheless turn out to be a dead end. We just revert to some previous stable intermediate form and try again. The recognition of useful, or stable, intermediate forms – what Simon calls 'selectivity' – requires some form of information feedback to guide the search. He identifies two such kinds of feedback: first, the *selective trial and error* method employed in finding the mathematical proof requires us to be able to select stable intermediate lemmas. This is analogous to organic evolution, in which "various complexes come into being [...] and those that are stable provide new building blocks for further construction," [Sim62, p.473] thus being selected and tried through further evolution. A second source of selectivity, *previous experience* allows us to quickly identify some intermediate solutions as useful and discard others. A biological analogy for experience could be *reproduction*, allowing earlier "solutions" to the search for most fit individuals to be passed on and modified through inheritance of genetic material.

Whether convinced or not about the pervasiveness and importance of hierarchy as a general perceptual strategy, it seems likely that some amount of hierarchical structure may be found in music. Remember that hierarchical structure need not be (and often is not) based on relations of power, subordinance, or importance, as e.g. in the boss-employees structure, or military hierarchy and ranking. Hierarchy as discussed above may just as well describe relations of containment – of spatial or temporal arrangement, in which some whole is built up of parts. It is this whole-part relation that interests us in connection with music. Pick any pop song from any radio station; most people will not have difficulties identifying some part as "a verse" and some other part as "the chorus", or "the refrain". Already a partitioning has begun, and partitioning the verse further into chords and melodic phrases does not make these less important than, say, the entire refrain. Chords and phrases are simply smaller constituent parts, but without them, the song would not be what it is. We return to the idea of evolving hierarchy in section 5.6.

2.2.2 Multiple hierarchies

Dannenberg comments in his *Brief Survey* [Dan93] that a single hierarchy system is inadequate to represent music scores because score notated music most often contains multiple interweaved hierarchies. "Notes can have several beams, and beams are often broken to help subdivide rhythms, so a multilevel beam hierarchy arises naturally. Phrase markings, ties, and slurs form another multilevel hierarchy that is completely separate from beaming." [Dan93, p.21] Dannenberg lists several other possible hierarchies. Voices or instruments can be considered in a hierarchy, where e.g. the solo violin is a member of the 1st violins, who are part of the violins, who are part of the strings, who are part of the orchestra. Segmentation can be done in hierarchically ordered units such as movements, sections, and measures. Phrases may cut across sections and develop their own

hierarchy; and chords could e.g. be structured hierarchically after harmonic importance, or distance to the tonic. Some of these different structural hierarchies may converge on some occasions but most likely will also diverge on many others. Thus describing an entire piece through a single hierarchy is inadequate.

2.3 Music and language

The comparison of music and language has generated a considerable amount of interdisciplinary debate involving fields like music theory, philosophy, linguistics, semiotics, cognitive psychology, artificial intelligence and computer science. There are heated exchanges of opinion on the status of music as a conveyor of information and the possibility of semantically loaded music. According to Lerdahl and Jackendoff[LJ83], a literal coherence between music and linguistics is doubtful. Language has a semantic content; a sentence *carries a meaning*, which we are most often able to agree upon, but there is no such agreement in the understanding of a musical phrase. Linguistic methods can be adapted, however, to describe music. A generative grammar can be used to describe the structure of infinitely many instances of musical pieces. We will not go much further into the existing debate on musical meaning, only to hint at a possible interpretation of 'meaning in music' in relation to grammatical descriptions. This little idea fits into our use of the multiple viewpoint system to build hierarchical structures.

The linguistic turn What we have studied most in this project is what we call the *linguistic turn* in computer music research. There seems to have been a collective euphoria over the strength of much of the pioneering work in linguistics done in the 1950s and 1960s by researchers such as Chomsky. Generative grammars are a very powerful abstraction, and by the mid 1970s, the enthusiasm washed over the community of researchers in cognitive and computational musicology. We imagine that at the time every computational musicologist had a generative grammar in his back pocket. The most renowned publication remains Lerdahl and Jackendoff's *A Generative Theory of Tonal Music* (GTTM) from 1983. It seems that by the mid 1980s, there was general realisation that music as a problem domain is different from language. In any case, simply throwing a set of linguistic tools at the musical domain is not enough, as is also underlined by the many attempts to incorporate musical knowledge into grammars and to find grammar variants well suited to the description of music. To judge by the number of publications, by 1985 the linguistic turn had ebbed away. To our knowledge, only very recently has there been renewed interest and major publications on the topic of formal grammars and the computer analysis of music. E.g. David Temperley's 2001 book *The Cognition of Basic Musical Structures* [Tem01] extends and refines many concepts from Lerdahl and Jackendoff's GTTM. In section 2.4, we present some of the exponents of the linguistic turn.

2.3.1 Automatic music

In *Grammaires, automates et musique* [Che02], Marc Chemillier remarks that the technological development in recent years has provoked a two-fold change in our relation to music: in the creative process, and in our consumption of

music. With the advent of the *sampling* technique that is the founding principle of loop-based music, e.g. in the genres of hip-hop and electronic music, the activity of composing new music is gradually displaced from the traditional compositional activities (such as composing motives and themes and combining them with harmonies, bass figures, percussion rhythms, etc.) towards the activity of sequencing and mixing pre-existing loops or "samples". This recycling and repeated sequencing of pre-recorded material can be effected manually but in fact lends itself well to a macro-like automatisation. Using grammars to describe and structure music is a natural prolongation of this automatisation. Particularly, composition using a combination of different representational entities (audio fragments, midi sequences, sounds generated by software synthesizers that are controlled by midi signals or otherwise, etc.) could perhaps be integrated in an elegant way using grammars.

Although the record industry for twenty years has slept through what seems like the most promising development to their business, Internet distribution of music recordings is now (and has been for a number of years) technologically feasible and potentially very lucrative. Albeit in softer words, this is the second change pointed out by Chemillier [Che02], also entailed by technological development. Distributing music over the Internet cancels the material necessity of a single, unique master record that is mass-reproducible at low cost in vinyl or compact disc. In fact, on the net, there might be as many versions of the musical work as there are downloads of it, if the consumer is allowed to specify certain parameters for the desired piece of music. Systems for interactive television are being built at this moment, so the idea of interactive music systems is not at all revolutionary. What might upset some, though, is that such a change affects the very idea of a musical work of art; the *œuvre* becomes not a unique *piece* of music, but a *programme* able to generate pieces of music within a certain frame. A pioneer of algorithmic composition, Pierre Barbaud introduced this notion of a musical work as a programme in the 1960s and also worked with formalisms like finite automata to generate music (see section 2.4.2). We know from the theory of computation that certain classes of automata are equivalent to certain classes of grammars, and thus the "œuvre as a generative programme or automaton" available on the net for user customised music download suddenly sparks new relevance into the investigation of grammatical descriptions of music.

Another very promising application of automatic music descriptions is the generation of context-dependent music for computer games. The interactive nature of gaming puts new requirements on the generative mechanism, as does the overall setting of the game: music is supposed to be a mood-supporting or mood-inducing effect that works in conjunction with the rest of the audible, visual and motor-sensitive input supplied to the user by the game to create the desired illusion.

The section title *Automatic music* has a double edge: it can refer to music created through automated composition, where little or no human intervention is needed once the generating programme or automaton has been built. Or, it can refer to sets of musical structures as described by automata, or equivalently, generative grammars. Inspired by Chemillier's idea, we imagine a scenario in the not too distant future where e.g. electronic music is sold not in mp3 files but in extended grammar files specifying legal structures and some probabilistic control mechanisms that combine basic drumloops, instrument sounds, effects, etc. needed to generate and synthesise a piece, or in fact, many pieces. In this

scenario, we download not the newest song by our favourite band but the newest song space. The song space is a file played not by WinAmp but by WinGenerator, an interpreter for the downloaded grammar files. The sound produced by the interpreter has some particular feeling to it, all the while it actually produces a different composition each time it is invoked. For copyright reasons and the secrecy of private work methods, such a scheme will probably be more successful if the finished grammar files are compiled or encrypted to disallow reverse engineering that would discover the grammar and associated building blocks of a particular song space. On the practical side, there is an implementation challenge in ensuring that the generative mechanism that interprets the grammar doesn't require excessive amounts of computing power.

Perhaps such creation and distribution would be more prone to commercial success with electronic music than with e.g. classical music. As mentioned in the introduction, this thesis is more specifically concerned with the abstract, structural properties of classical Western tonal music, as found in a traditional musical score, than with electronic music or any audio representations of music. We do consider in thought, however, a way to adapt the SimFinder system's search for parallelism and hierarchical structure to sequencer/sampler-based compositions (section 6.1.9).

2.3.2 Formal languages and grammars

The central question of interest to us concerns *formal* languages and musical structure: can musical structure be described by formal languages, and if yes, what kind of language is necessary? This is not easily answerable, if an answer exists at all. Let's start with a brief recapitulation on formal languages, their descriptive power, and then proceed to an account of some of the historical approaches to the use of formal grammars to describe music.

The theory of computation has taught us that both automata and grammars may be used to describe formal languages. A formal language is a set of strings of symbols, and an automaton is able to recognise strings that are in the language it describes, and reject strings that are not. Similarly, grammars can be said to generate their language, in the sense that all strings, and only strings, in the language of a given grammar may be derived, or generated, by applying the production rules of the grammar.

The *Chomsky hierarchy* divides the formal grammars into classes; Type 3 is the most simple class, namely the *regular grammars*, whose languages alternatively may be described by finite automata. Type 2 denotes the *context-free grammars* (CFGs), which are equivalent to push-down automata, Type 1 denotes *context-sensitive grammars*, the equivalent of linear bound automata, and Type 0 denotes the so-called *free grammars*, equivalent to Turing machines. Type 3 and 2 grammars are computationally tractable, but type 1 and 0 grammars are more difficult to work with. There are many modifications to CFGs that attempt to augment their expressiveness while still keeping the context-freeness; we shall see some examples in the next section (2.4). John Martin's *Introduction to languages and the theory of computation* [Mar97] is a good textbook on languages, grammars, and automata.

2.4 Historical uses of grammars for music

Perhaps we have not yet spelled out completely what it means to describe music using a formal grammar. Recall that a formal grammar through its production rules generates a set of strings of symbols. This set is referred to as the *language* described by the grammar. Music can be given a very detailed symbolic representation, most often in the Common Music Notation[1]. A sequence of notes and other musical symbols, then, is a string of symbols that lies in some set, or language, of symbol strings. This language can be described, or generated, by a generative grammar, but we don't know what rank in the Chomsky hierarchy the grammar must have. This depends on the complexity of the string.

If we should succeed in finding a suitable grammar for, say, a Mozart piano sonata, the generative grammar will also be able to generate many other pieces. One of the most enchanting ideas from the literature of musical grammars is that of finding a grammar to describe a set of pieces (e.g. *all* Mozart's piano sonatas), or the entire œuvre of a composer, or a musical style as such. The idea is that there are some common stylistical traits to such a set of musical pieces that may perhaps be modelled in a generative grammar. Finding a grammar that generates an entire musical style could be done in two ways: the top-down, or knowledge engineering approach, in which a set of production rules are manually determined; the other, bottom-up, or empirical induction approach would require an analysis of at least a representative subset of the style and a grammar induction method to combine the analytical results into a coherent set of production rules generating all the pieces in the style. Some attempts have been carried through with success, e.g. Steedman's grammar for jazz chord sequences (see section 2.4.8) below. Steedman makes extensive use of knowledge engineering, embedding musical knowledge in the productions of his grammar.

However, clearly not all music is well described as strings of symbols. Non-monophonic[2] music poses problems, because if we stick to grammars that generate strings, we would have to describe a non-monophonic piece in terms of strings. If each part in a four-part piece is monophonic in itself, then each part can be described as a string of symbols, but there must be some interaction or correlation among the four parts that makes a representation consisting of four separate grammars inappropriate[3]. Otherwise any parts could be combined with any parts to produce good music, which is clearly not the case. Also, we cannot expect the piece to be well described as a sequence of sets of simultaneous notes, like a chord sequence. The problem arises e.g. with polyphonic music, where parts operate in a melodically and rhythmically independent way.

Not surprisingly, most approaches have focused on monophonic music. Once non-monophonic music is taken up, the sequential/parallel dichotomy inherent also in the two approaches loosely described above, rears its head. Our graph representation is an attempt to break out of this representational dualism. What we need then, is a graph grammar to describe a (set of) music graph(s). The sequential/parallel dichotomy is described in section 3.1.3. The music graph is described in section 4.2.

[1] See section 3.1.1.

[2] For an explanation of the terms *monophonic*, *non-monophonic* and *polyphonic* music, see the beginning of section 3.

[3] And there are no exceptions for *string quartets*!

2.4.1 Schenker

Heinrich Schenker, a much referred-to German music analyst and thinker, is regarded by many as the founding father of a more formalised movement in music theory, aiming to describe musical structure through hierarchical structures. Schenker's analysis methods, however, are not as formal as to be readily implemented as an automatic analysis program.

Schenker has proposed a semantics of the music described from a tonal point of view. A musical piece is a journey in the tonal space from one key and through others back to its origin. His analytic methods are based on describing hierarchies of stable tonal centres. On each level some most stable events are taking control or dominating what else is going on. We would like to cite David Cope: "Schenker (1968-1935) demonstrated that music could be understood on three principal levels: foreground, middleground, and background. His analytical approach involved the revealing of each level as one approached a work's *Ursatz* (kernel or fundamental structure). This included a stripping away of elaborations, modifications, or transformations of the foreground through a unique notation that could demonstrate that non-consecutive elements can resolve or extend foreground elements."[Cop91, p. 37]

The notion of Ursatz is furthermore supposed to be equal for all pieces or movements. The Ursatz consist of two things. The *Urlinie* (fundamental line) which consists of a diatonic stepwise descent to the root from the 3rd, 5th or the octave. The other fundamental thing is the the bass line, which encapsulates the harmonic motion of the piece as a tonic followed by the dominant and then returning to the tonic.

2.4.2 Barbaud (1960s)

Barbaud has built several music composition programs based on finite automata. The programs comprise a Markov chain whose output is transformed by a rational transducer into a piece of music. A Markov chain is similar to a finite automaton, but additionally has a probability associated with each transition between the states of the automaton. Barbaud's Markov chain produces the harmonic framework for the piece (like a figured bass notation). The probabilities in the Markov chain are set by Barbaud himself, who points out that his automata are only examples and encourages his readers to construct their own automata. A rational transducer is a finite automaton that accepts strings in a regular language and outputs strings in another regular language; each transition 'reads' a symbol from the input string and 'writes' another symbol as its output, thus transforming the input. The role of Barbaud's transducer is to add melodic motives to the harmonic framework and thereby complete the composition. Motives are chosen randomly from a table of known motives and transposed diatonically (i.e. within the appropriate scale) to fit the harmonies dictated by the Markov chain. The rhythm of the chosen motif may also be altered to fit the underlying bass line. Chemillier [Che02] gives an example of a two-voice counterpoint piece generated by one of Barbaud's programs[4].

[4]Several audio and midi examples of computer generated music are available from Chemillier's web page at http://users.info.unicaen.fr/~marc/publi/grammaires/grammars.html

2.4.3 Simon and Sumner (1968)

In 1968, Herbert Simon and Richard Sumner propose a "language of musical pattern" [SS68]. The hypothesis is that "musical patterns, even when quite complex and sophisticated, involve only repeated use of the few simple components [...]: 'alphabet', 'same', 'next', and rules of combination." [SS68, p.222]. The article describes a formal representation of musical patterns and then outlines a method to induce patterns from a score. Musical patterns have several traits:

1. they involve *periodicity*, or repetition, and often with *variation*),
2. patterns make use of *alphabets*, i.e. sets of symbols over which their description is constructed.
3. patterns may be *compound*, made up of subpatterns that in themselves are arrangements of symbols,
4. patterns generally possess *phrase structure* marked in the score by various punctuation marks (e.g. bar lines, fermata, etc. These are used in the pattern induction program),
5. and patterns may be *multidimensional*, involving information in dimensions such as pitch, rhythm, harmony, dynamics.

An alphabet could be e.g. the 'diatonic alphabet' of all notes in the C major scale. A pattern p could then specify that its ith symbol x_{pi} was, say, the second symbol in the diatonic alphabet (a D, if the diatonic alphabet is in C major). It could also specify that its ith symbol was `NEXT(DIAT,`x_{jk}`)`, meaning that x_{pi} was the next symbol in the diatonic alphabet (`DIAT`) counting from the kth symbol in pattern j. This could be used to describe a transposition. Another relation on symbols is `SAME` which tells us that two symbols or patterns are exactly alike.

The pattern induction method consists in combining cues taken from rhythmic and harmonic elements in a score. E.g. the first eight measures of a Beethoven *Dance*[5] (see figure 2) are segmented into groups based on patterns of note lengths:

$$(1+1+4)^2(1+1+1+1+1+1+1)(1+1+1+1+1+1+1)(1+1+4)^2(1+1+1+1+1+1+1)(2+4)$$

where each number 1, 2 or 4 represents the length of a given note in the sequence (where 1 is an eighth) and '+' signifies immediate precedence, or concatenation in string terminology. This is rewritten as:

$$(1+1+4)^2((1)^6)^2(1+1+4)^2(1)^6(2+4)$$

and used to show that the rhythmic pattern of the chosen eight bars follow an ABAC form, where B=$((1)^6)^2$ and C=$(1)^6(2+4)$. Simon and Sumner do not mention it, but the expression arrived at is a regular expression, that could alternatively be described by a regular grammar. One wonders if they thought all rhythmic structure in music can be described in this way.

The outlined algorithm builds on the idea of sequence extrapolation, known from psychological tests where humans are asked to give the next symbols in a sequence of letters or numbers. But the authors concede: "Some aspects of the performance of the pattern inducting program for music have been hand

[5] The article does not tell us where this dance is published.

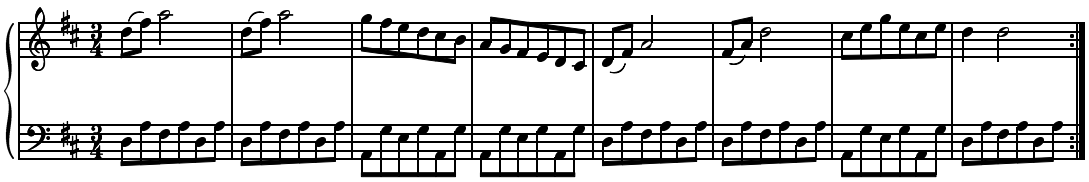


Figure 2: The start of the Beethoven *dance* used as an example by Simon and Sumner. (They do not specify *which* dance.)

simulated, but construction of the actual computer program has not yet begun". We find the further description too vague to tell how successful the approach would be. On the other hand, if the program was implemented at all, and had been a huge success, why aren't everyone in the field referring to it by now? The article seems a little marked by the optimistic pioneering spirit of the 1960's early thoughts on computational musicology and on formal representations of music usable in computer programs, but it does give us some cues to and ideas of the characteristic properties of patterns we can expect to find in music (see the list above). The subject of how to locate patterns based on musical characteristics is discussed further in sections 3.2.3 and 4.

2.4.4 Buxton, Reeves, Baecker, and Mezei (1978)

In the late 1970's, hierarchical structure in music was considered in the Structured Sound Synthesis Project (SSSP) at the University of Toronto, Canada. SSSP developed a composition tool to work with synthesized sounds and therefore focused on the needs of the human composer when interacting with the machine. In [BRBM78], four basic composition tasks are listed:

1. The definition of a palette of timbres using different sound synthesis techniques,
2. Score definition, i.e. composing the entire pitch-time structure of the piece,
3. Orchestration, i.e. combining elements of the composed structure with different sounds from the timbre palette, and
4. Performance of the material, from the smallest constituent part to the entire piece.

The described sound synthesis techniques used to define timbres are different ways of controlling a synthesizer; one would expect that newer versions (ie. newer than 1978) of the SSSP also include the ability to load and manipulate recorded sounds in a sampler-like way. An important point in the SSSP is not to impose any a priori ordering on these four tasks, to allow the composer to jump freely back and forth between them.

What interests us most about the SSSP is the data structure used in the score definition task. There are different opinions of what a score is. At one end of the spectrum is the view that a score is a set of notes occurring at different

times. In hierarchical terms, this gives a very flat tree structure. At the other extreme, a score can be seen as a single entity[6]. But both views treat music "chunk-by-chunk", either at the note level or at the score level. What remains to be defined is the intermediate structural levels of a composition.

A score is defined as "an ordered sequence of musical events"[BRBM78, p.11]. Here, we interpret the word 'sequence' *not* to mean that musical events are ordered in a strict *temporal* sequence[7], which would prohibit simultaneous notes and thus chords; rather, we take it to mean 'an ordered collection' of musical events, and further: "By 'musical event' we mean simply an event which occurs during the course of a composition which has a start-time and an end. Thus, the entire composition constitutes a musical event (the highest level), as does a single note (the lowest level). Similarly, chords, motives, movements, etc., are all musical events. [...] any musical event (e.g., a motif) can be made up of composite musical events (e.g., chords and notes)" [BRBM78, p.11].

Buxton et al. actually give a BNF grammar for the most basic elements of the hierarchy ('Mevent' is shorthand for 'musical event'):

$Composition ::= Mevent$;
$Mevent ::= Mevent^* \mid Score \mid note$;
$Score ::= Mevent$;
$note ::= terminal$ (i.e., some musical note);

Each composition then will be a derivation tree of the structural grammar, where intermediate levels such as chords and motives are represented by sub-scores, which will probably contain their own sub-collection of musical events, using a $Mevent \rightarrow Score \rightarrow Mevent \rightarrow Mevent^*$ derivation.

Here, one of the properties of hierarchical structure that we described in section 2.2.1 is exploited by Buxton et al. A recurring motif need only be represented once in a *master-copy*, and all other occurrences of it can be represented as *instances*, of the master-copy. Each instance is provided with a reference to its master-copy and a list of transformations applied to it, so that variations are allowed among instances. The instances of Buxton et al. are *shallow copies* of the master-copy. This means that if the master-copy is changed, all instances are automatically changed too. We shall discuss the difference between shallow and deep copies further in section 5.4.3.

2.4.5 Roads (1979), Moorer (1972), Winograd (1968)

In 1979, Curtis Roads published an excellent article named *Grammars as Representations for Music* [Roa79] which has since become a point of reference for many publications on computer-based grammatical descriptions of music. Roads first considers different types of grammars and then gives a survey of grammars applied to music studies.

Roads distinguishes between *iconic* and *symbolic* representations. Iconic representations bear some topological similarity with the represented object (the denotata); e.g. a digital recording in an audio waveform. Symbolic representations are constituted of signs that have no contiguity with their object – the

[6] The authors credit Xenakis for this view, in his book *Formalized Music*, Bloomington: Indiana University Press, 1971.

[7] Such as the sequential subgraphs which we describe in section 4.2.

link between signifier and denotata is purely conventional. Formal languages and grammars are examples of symbolic representations.

Roads describes the classical *transformation grammar* advanced by Chomsky to describe natural language. It comprises three levels:

1. A phrase-structure grammar, describing abstract kernel sentences in terms of nouns, verbs, etc.

2. Transformation rules. When applied to the kernel sentences, these rules produce sentences in English (or some other natural language).

3. Morphophonetic rules. These rules transform the English sentences produced by the transformation rules into streams of phonemes.

The transformation rules involve complex issues with e.g. active-passive verb relations, and auxiliary verbs. Roads conjects that there is no clear musical analogy to such relations, so a grammatical description of music should skip the transformation rules and concentrate on building a phrase-structure grammar. Also, a set of morphophonetic rules interpreting abstract tokens into a lexicon of sounds could be useful as a sort of *orchestration* layer in grammatical description of music.

After a presentation of the Chomsky hierarchy, Roads gives a short description of *regulated grammars*, which are different techniques to augment the expressive power of context-free grammars while not entirely turning them into context-sensitive grammars. The article then describes a series of grammar-related musicological endeavours, ranging from the early reflections of French musicologists Ruwet and Nattiez to Roads' own work on "composing grammars". There is no point in reproducing the entire survey here, but the ideas of Moorer and Winograd have caught our attention.

Moorer (1972) Consider the following grammar example:

Piece ::= Sonata | Rondo | Fugue
Sonata ::= Exposition Development Recapitulation
Exposition ::= MelodyInCmajor MelodyInFmajor MelodyInCmajor

Now if we were to continue the context-free grammatical definition, how could we specify that the recapitulation should make use of the same melodies as were used in the exposition? In an article on computer composition [Moo72], James Anderson Moorer uses the example to point out that there are problems with using context-free grammars to describe music. In accordance with the general sonata form, the recapitulation must also involve variations of *the same* melody as in the exposition. This can be specified in a context-sensitive grammar, not in a context-free grammar.

Moorer discusses the use of Markov probabilistic methods for the composition of melodies. When probabilistic automata are inferred from an existing corpus, higher order methods produce exact repetitions of the input melodies, while lower order methods produce note combinations that are almost meaningless. The intermediate order methods produce a few new and interesting results, but mostly "degenerate" melodies consisting of ascending or descending scales. He concludes: "What is needed here is an intermediate-order grammar with a

selection mechanism that filters out things that can be represented too simply." [Moo72, p.107]

Moorer's own experiment features a composition algorithm whose many parameters should be tuned by the user until the algorithm outputs music that he likes. Composition is done in a top-down manner. First the overall form of the piece is structured in terms of *minor groups* and *major groups*. The major groups structure sets of minor groups, while the minor groups each have rhythmic and chordal structure. When this overall structure and the rhythmic patterns of each minor group are assigned, the chords are chosen. First a chord hypothesis is generated from first-order probabilities, then it is accepted or rejected by heuristics introduced by Moorer. The heuristics are meant to act as the above-mentioned selection mechanism to "filter out long sequences of nontonic chords or otherwise dull sequences." [Moo72, p.111] E.g. ABBA forms, where A is not the tonic, are discarded. The third and hardest step is to construct a melody over the chordal, rhythmic and grouping skeleton. Notes are chosen on the basis of minor groups; at each point, either a new figure is invented or a previous figure is copied or copied-and-altered. New figures are invented using probabilistic models of note-to-note intervals and of when notes should be constrained to the chord and when not. The chord constraint probabilities vary for stressed and unstressed notes. The central balancing act of such an algorithmic composition seems to be to "preserve repetition and periodicity but to deny boredom." [Moo72, p.111] Finding such a balance for Moorer's three composition levels (form and rhythm, chords, and melody) is by far the hardest for the melodic line.

Moorer shows five monophonic melodies composed by his algorithm. They clearly show some inner structure in the sense that parts of the melody are repeated, or repeated in an altered form. But Moorer concedes that they are "alien sounding, indicating that perhaps not all 'legal' melodies are interesting melodies." [Moo72, p.112]

Winograd (1968) Terry Winograd has used a "systemic grammar" for semantically-directed parsing of music pieces [Win68]. The idea is that the analysis is primarily a syntactic parsing of the music, but it is augmented with semantic routines that guide the parsing process. Winograd's notion of semantics is based on the harmonic functions found in music. It produces a 'meaningfulness' rating of different parses to eliminate derivations that are syntactically valid but musically meaningless. "Some progressions such as V→I or II→V, have a clear function, while others such as VII→II or III→I may occur, but do so at the cost of confusion of the tonal structure, and can thus be considered less likely, or less 'meaningful'. "[Win68, p.24] Winograd gives an example of a simple grammar for harmonic phrases (see figure 3) and then proceeds to argue that such a grammar is inappropriate on it's own; we need additional rules or transformations to produce a concrete harmonic sequence of chords and chord inversions from the abstract roman numeral harmonic functions. Winograd's semantically-directed parsing using a systemic grammar is an attempt to remedy this.

Problems with context-freeness What Moorer and Winograd's work have in common is that they use context-free grammars that are augmented in some

harmonic phrase $\rightarrow$ opening cadence
cadence $\rightarrow$ plagal
cadence $\rightarrow$ authentic
authentic $\rightarrow$ dominant (linear) tonic
dominant $\rightarrow$ V
dominant $\rightarrow$ VII
$V \rightarrow V_{triad}$
$V \rightarrow V_{seventh}$
$V_{seventh} \rightarrow V_7$
$V_{seventh} \rightarrow V_{\frac{6}{5}}$

Figure 3: Winograd's example grammar for harmonic phrases.

way to more appropriately describe music. Moorer's problem of specifying that the same melody should occur in different variations in the exposition and the recapitulation of the sonata is equivalent to Winograd's problem with producing the right chords from the roman numerals. There is a relationship between these non-terminals (be they melodic variations or harmonic functions) that is difficult to capture in a context-free grammar. The two authors therefore introduce additional mechanisms to make their formalism more powerful. This is some of the ammunition that Roads uses when he concludes: "in nearly every study discussed in this paper the production rule has been shown to be insufficient by itself as a representation for music, particularly in a context-free form."[Roa79, p.53] On the other hand context-sensitive grammars are "unattractive as work tools"[Roa79, p.54], because they are both difficult to specify by hand, and there are technical problems with context-sensitive parsing. An extension of ordinary context-free production rules is needed to work with grammar representations of music.

Another problem with context is that most music is non-monophonic: "In formal grammar theory, 'context' is a sequential notion, while in music, context is both parallel and sequential. [...] A good model for 'context' will most likely involve a rather complicated data structure. All of these notions of extended musical context lead, in a computer implementation, to an extended grammar representation, e.g., production rules augmented by embedded procedures. In any case, the notion of a grammar as simply a list of production rules is inadequate for music." [Roa79, p.54] If the reader will pardon the long citation, we think this is a very good reason to adopt a graph grammar approach to the hierarchical description of music. Graph grammars are more powerful abstractions than string grammars, and using a graph, we are able to represent both parallel and sequential context, as well as mixtures of them.

2.4.6 Holtzman (1980)

Holtzman describes a Generative Grammar Definition Language (GGDL) for music [Hol80] and gives an example of its application on a Schönberg piece. The GGDL is implemented as a computer program and is intended to be used either by musicologists as a tool to test their research method, or by composers

to investigate their compositional method or simply, to compose. The GGDL lets you specify a generative grammar, which it then puts into action and generates music with. The question of grammar inference from a corpus of known pieces is thus not touched upon. Holtzman describes it the following way: "A traditional musical score is a collection of utterances in a musical language. The language theoretically could be formally described by a set of rules. Typically, the grammar by which a musical structure is derived, the process of composition, is not easily, and certainly not completely, detectable – it is oblique but may be inferred." [Hol80, p.5]. The GGDL is a tool for human, not automated computer inference of musical grammars.

In GGDL, the user specifies a type 0 grammar, i.e. a free grammar, composed of the "phrase-structure rules" and of "transformational rules". One of the examples is a small grammar for a sonata form:

Sonata $\rightarrow$ A B A
B $\rightarrow$ development
A $\rightarrow$ Theme_1(key)Theme_2(key)
Theme_1(key) $\rightarrow$ Theme_1(tonic)
Theme_2(key)_development $\rightarrow$ Theme_2(dominant)
development_Theme_2(key) $\rightarrow$ development_Theme_2(tonic)
development $\rightarrow$ modulation of themes

which could produce the result:
Theme_1(tonic) Theme_2(dominant) modulation of themes Theme_1(tonic) Theme_2(tonic)

The underscore character is Holtzman's notation for "any string". A right-hand side underscore should refer to the same string as the corresponding left-hand side underscore.

The phrase-structure rules are the usual grammatical production rules possibly coupled with a set of *rewrite control rules*. Instead of letting the machine choose a random production when several are possible (such as choosing either A or B in "$X \rightarrow A|B$"), we can specify rules to control which production is chosen. *Blocked generation* specifies that when one production has been used, it may not be used again before all other productions with the same left-hand side have been used. In *finite-state generation*, a probabilistic finite-state automaton is supplied with a production rule to control how probable each production in the rule is at any given moment. *Multiple invocation* specifies that a production rule once invoked should give a number of productions as a result. This can be used in conjunction with a blocked generation rule to give random combinations of a set of notes, as used in serial music. Furthermore, *metaproductions* can be given, that act not upon non-terminals in the derivation tree, but upon the set of production rules before they are even applied. This is useful for describing structures with undefined objects, which are later filled in with some given structures. The transformational rules act upon sequences of symbols. One can either reverse, invert, transpose, or merge the sequence(s). Reversing a sequence is an operation which produces the music 'backwards'; inversion is an inversion of the inter-symbol interval found in a given ordered alphabet, and a transpose is a transpose of a number of symbols in the ordered alphabet. Merging two sequences consists in picking the next symbol alternatingly from one and the other sequence. The examples given by Holtzman use abstract symbol

sequences, and the given generative mechanisms are sufficiently general as to be used for both tonal and non-tonal music.

Holtzman uses two concepts from linguistics, namely those of *syntagmatic* and *paradigmatic* relationships. 'Syntagm' means 'combination of signs', and in this context, 'paradigm' means 'an example of a conjugation or declension showing a word in all its inflectional forms'[8]. Consider the derivation tree of some string of symbols from a grammar. Syntagmatic relations occur in a horizontal direction, they are "surface-relations of the tokens at any one level in a structure – they are the result of the combinations of tokens on the same level." [Hol80, p.4] Paradigmatic relations occur, according to Holtzman, in a vertical direction "when tokens may be substituted for one another and still perform the same function in the system as a whole."[Hol80, p.4] A linguistic example of a syntagmatic relation between words is the difference between the two sentences:

He whispered softly, and *He shouted softly*

It is quite plausible that 'softly' come after 'whispered' in a sentence like this, but 'shouted softly' is an unexpected combination of words. Paradigmatic relations in linguistics are based on notions like *similarity*, *inclusion* or logical (not temporal) *entailment* and examples include relations like *synonymy* (identical meaning relations), *antonymy* (opposite meaning relations), *hyponymy* (specific-general relations), and *meronymy* (parts-of relations). In a formal grammar, think of the production rule shown in figure 4. This production specifies that A may be followed by B followed by C – syntagmatic relations *on this level* in the derivation, no matter whether A, B and C are terminals or non-terminals. It also suggests the paradigmatic relation that, in some cases, ABC is synonymous with CD, in that an X may be replaced by either one or the other. Also there is a hyponymy relation between X and the results it may produce, in that X is more general and the results are more specific.

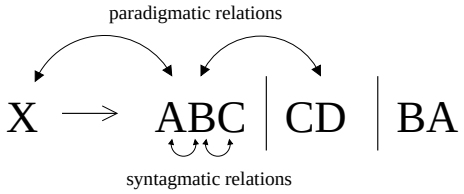


Figure 4: An illustration of *syntagmatic* and *paradigmatic* relations in a formal grammar production rule.

Are there syntagmatic and paradigmatic relations in music, and how could they be interpreted? Holtzman gives us a hint: "To the extent that paradigmatic relations define the function of objects in a structure, they may be considered the basis of the semantic interpretation of a structure. [...] A theme of melody or other musical macro-unit is a grouping of lower level units, and is therefore a vertical or semantic interpretation of the lower level unit's function. Likewise, a juxtapositioning of themes (or series, etc.) is a semantic interpretation of lower

[8]Merriam-Webster Dictionary, http://www.webster.com

level units." [Hol80, p.4] In other words, we may construct a musical semantic interpretation of paradigmatic (similarity or inclusion) and syntagmatic (juxtaposition) relations. Musically speaking, saying that X is a musical phrase also tells us something about how X can be used in combination with other building blocks on a higher level. We don't combine phrases with each other in the same way as we combine notes with each other. And saying that X can in turn produce either ABC, CD, or BA tells us that these are different 'inflections' of the same X, different variations of the same phrase. We shall return to this when speaking of musical parallelism and variation (section 3.2.2).

With all its rewrite control rules, metaproductions, and transformational rules, the GGDL is a very powerful and complex description language. So much so, that perhaps one looses the elegance and original simplicity of the idea of generative grammars. If the hierarchical description of a piece is longer than the non-hierarchical description, is it still worth the effort? The complexity of a GGDL grammar can probably be dealt with computationally because it is only used for generation from a human-defined grammar. Automatically inducing a meaningful GGDL grammar of some piece would seem to be a very tough challenge.

2.4.7 Lerdahl and Jackendoff (1983)

Since 1983, Fred Lerdahl and Ray Jackendoff have become the most prominent point of reference in the area of music and grammars, towering over everything else in terms of the number of references made to it. "A Generative Theory of Tonal Music"[LJ83] (GTTM) is the much discussed work that earned them this fame. The authors have a background in recent music theory and contemporary linguistics respectively. In the GTTM, they propose mechanisms for constructing grammars suitable for *Western tonal music* which is roughly speaking the major-minor based tonal music evolved in western Europe during the 18th and 19th centuries.

Lerdahl and Jackendoff make it clear that their theory is based on the mental processes of the listener. The fundamental assumption is that the listener is not just hearing one note at a time, but on the contrary perceives the music as organised patterns. The GTTM claims that the listener, while listening, generates a mental construct, and the goal of GTTM is to explain how this entity is constructed. The listening experience may differ from person to person, but some elemental constructs are controlling the experience. They introduce the term "experienced listener" – a person who is experienced in the musical idiom and able to grasp and analyse major pieces. The experienced listener is an idealisation of how one understands music. The rules of the grammar are meant to generalise the organisation that the experienced listener constructs while he is listening. The ultimate goal of the GTTM is to understand musical cognition, and thus describe a theory for formalising the listener's intuition.

Lerdahl and Jackendoff are very explicit or algorithmic in their proposed way of musical understanding. This has resulted in numerous rules. The overall ideas of the procedure is as follows. The music is first analysed in four separate ways according to: *grouping structure*, *metrical structure*, *time-span reduction*, and *prolongational reduction*. For each analysis, *well-formedness rules* and *preference-rules* are given. The application of well-formedness rules suggests different possible ways of dividing the music into smaller segments, which be-

long together according to the analyses. The boundaries of these segments may conflict, so the preference-rules are applied to solve the conflicts. Preference rules are criteria for evaluating and choosing between the possible analyses of the piece. In the end one ends up with a *preferred analysis* of the piece. Each of the 4 analyses suggests a hierarchical structure that partitions the piece in nested groupings.

The **grouping structure** analysis segments a homophonic piece into nested groups of varying sizes. The boundaries of the groups correspond to the phrasing of the music on the different nesting levels. Generally speaking, the phrasing boundaries correspond to points where it would be natural for a performing musician to draw a breath. We will return to this topic in section 4.3.1 on page 69 since we have been experimenting with some of the proposed rules.

The **metrical structure** analysis identifies the positions of strong and weak beats at the level of a quarter note, half note, a measure, and so on during a homophonic piece. The *tactus* of a piece is the overall "feeling" of the music – the most prominent metrical layer, that the conductor will point out, or that you eventually will end up tapping when following the music. This layer sets the tempo in which the listener will follow and experience the music.

The intuition behind **time-span reduction** is that we are able to hear "through" ornamentations of a melody. We are able to remove the "noise", and only hear the simple melody or chord progression behind. In this process we reduce the music into these most important structural events on different levels as for example the most important note in a motif, or the most important motif in a phrase. This is done according to a combination of melodic and harmonic factors, which forms a pitch hierarchy. An entire piece of western tonal music can eventually be reduced to a key note or a tonic triad. The time-span reduction is performed based on the result of the grouping structure and metrical structure analysis in a bottom-up manner, in the sense that parts come together to form a whole.

The word *reduction* comes from the Schenkerian vocabulary, where it is also used as an abstraction under which one can see the most important events on each hierarchical level in the analysis. By combining notes in logical entities it is possible to make a comparison of the entities. In this way we are able to make a comparison on a higher hierarchical level.

Prolongational reduction is a notion about tension progress. Lerdahl and Jackendoff describe tension and relaxation as transitions between stable and less stable states of the music. The fundamental idea is that any note in the music either is within a movement from relaxation to tension or within an opposite movement from tension to relaxation. The tension-relaxation movement occurs on many levels in a hierarchical structure. A relaxation could for example be a movement from a less consonant chord to a more consonant. On a higher level the harmonic movements in the music are of course important. L&J consider the rising fifth (tonic-dominant or I-V) as a tension (away from the key), and the opposite as a relaxation (back to the key). In a more elaborate cadence (IV-V-I), the subdominant (and all variants of it) is always in a relaxing movement towards the dominant which then again gives rise to a relaxation towards the tonic. Cadences are of special importance in this field. L&J use the prolongational reduction rules to build tension-relaxation trees in a top down manner. The prolongational reduction connects all parts of the music in a meaning relation to each other.

Should we criticise this very innovative and hugely successful work, we would pick out three points.

1. All in all the GTTM looks very much as a pseudo algorithm for making a preferred analysis, and this is true to some extent. The "algorithm", however, is not completely defined in every detail. To achieve computability, the preference rules could be weighted and thresholded. But this, L&J argue, is *artificial* since such a scheme produces only positive/negative judgements, and not ambiguous or vague ones. It is also somewhat *arbitrary*, because threshold values can be set to, say, 68, or 72; which is best? "A simple numerical solution of this sort provides an illusion of precision that is simply absent from the data." [LJ83, p.54] Also, the interaction of local and global rules on the musical structure is not sufficiently specified to allow a simple quantification mechanism to work.

   As computer scientists, one of our main goals is computability, which we have in the SimFinder system. Achieving it in the SimFinder is a little easier than to begin weighting rules in the entire GTTM machinery, because the SimFinder does not incorporate as many different and interacting layers of hierarchy as the GTTM does.

2. Another issue is the loosely defined concept of parallelism. The Metrical Preference Rule 1 states that if two passages can be construed as parallel, they will preferably receive the same metrical structure [LJ83, p.75]. But L&J omit a closer specification of what allows two passages to be construed as parallel. In section 3.2 we discuss the importance of parallelism to the structure of music. The SimFinder is a proposal for a search method that identifies parallel structures in music.

3. One major drawback of the GTTM is that the rules only apply to music with a single melody (including accompaniment). This melody determines the grouping structure, which asserts that the music only has one important melodic line. This is often not the case in baroque music, and the GTTM is not capable of analysing such polyphonic music. The SimFinder operates on a music graph that represents non-monophonic music and thus carries through non-monophonic analyses, although we admit they are more difficult to control than the monophonic analyses.

Several attempts have been made to explore the value of the rules in reality. Hirata presents a music knowledge representation framework, which relies on user input as time-span trees [KH02]. Temperley has contributed an extension and refinement of the overall theory [Tem01], and Miller et al. have experimented with the grouping and metrical structuring rules.[MSJ92]

2.4.8 Steedman (1984)

Mark Steedman provides us with some substance to the ideas of syntagmatic and paradigmatic relations, that Holtzman presented us to (section 2.4.6). In [Ste84], he presents a generative grammar for jazz chord sequences. The grammar is simple, just seven production rules, and it accounts for a multitude of variations over the standard "12 bar blues". The beginning production

$$\text{Rule0: S12} \rightarrow \text{I I7 IV I V7 I}$$

describes the fundamental form of a 12 bar blues, each roman numeral representing two bars (I is the tonic, V is the dominant and so forth). This is what all 12 bar blues chord sequences can be reduced to. The idea now is that these fundamental chords may be further subdivided and altered without changing the 12 bar blues 'feel', because "certain sets of chord sequences are considered by musicians to be closely related and to be harmonically equivalent in a sense which is close to the linguistic idea of 'paraphrase'." [Ste84, p.53] When paraphrasing, we obtain a synonymy relation between the original and the paraphrase; but this is exactly what a paradigmatic relation is. Specifying e.g. Steedman's rule number 2:

$$\text{Rule 2: } \mathrm{x}(7) \rightarrow \mathrm{x}(7)\ \mathrm{Sd}_x$$

tells us that without changing the blues feeling, any chord x can be replaced with two chords of each half the duration of x [9], where the first is the same chord x, and the second is the subdominant of x. Further, if x is a seventh chord, then the produced half-length x must also be a seventh. We have the paradigmatic relation that an "x" is synonymous with "x Sd_x" in this paradigm. The semantic paradigm is determined by the entities we are concerned with: chords; "the domain that corresponds to semantics – that is, the 'meaning' of chord sequences – is the domain of *harmony*, in which, for example, a given pair of chords played in succession may convey the meaning of a 'cadence'." Therefore substituting a cadence with something else should preserve the overall feeling of closure or relaxation that is the content, or musical point, of a cadence.

To give another example, in Rule 2, it is the second part-element that is changed to the subdominant, not the first. Consider a metric analysis of a 12 bar blues. It is subdivided in three times four bars. The first four bars can be recursively divided in two, giving metric subdivisions in which the first of two elements is always more metrically stressed than the second element. Thus the harmony of the second element is subordinate to that of the first and can be changed (e.g. changed to the subdominant in Rule 2) without affecting the overall harmonic expression of the two elements as a whole.

Rules 1 and 2 are phrase-structure rules. The grammar also contains *substitution rules*, which specify in which context a symbol x may be substituted by another, y. These are syntagmatic relations, describing which elements may be combined, and in which order. Cadences are examples of combinations of elements that we expect to hear in a 12 bar blues, as in fact in most other Western tonal music. If we should tentatively describe some of the meaning of a cadence, it has a tension-relaxation movement, which communicates a strong sense of closure, or coming-to-an-end, to us. This coming-to-an-end also tends to signal a return to the tonic, which can be used to good effect, e.g. when modulating to another key, to make the listener feel at home in the new key. The sequential arrangement V-I that we call a cadence thus makes more sense than e.g. a I-♯IV transition, just as it makes more sense to say "he whispered softly" than "he shouted softly". Steedman explains in some detail the music-semantic reasons behind different substitution rules, e.g. the use of leading notes to create expectation. It is an important point that when familiar with the idiom of

[9] Steedman's convention is that the length of the left-hand side should be equal to the length of the right-hand side of every production, so when subdividing x as we do in Rule 2, each subdivision receives half the length of x.

Western tonal music, we *do* actually hear these connections, which according Steedman, justify the grammatical structure described. He encourages readers to play the examples on an instrument to "hear" that the described structures are as they are.

Steedman draws a distinction between *valid* and *good* chord sequences. The grammar only defines which sequences are valid, and admittedly generates both sequences that are good and sequences "of a rather fringe variety" [Ste84, p.66] The point is that the grammar generates only sequences that are still *meaningful* in terms of the syntagmatic and paradigmatic relations inherent in the "12 bar blues" style. This does not prevent them from being far-fetched; the grammar would need additional rules or alterations to sort that out.

2.4.9 Conklin (1995)

Darrell Conklin has among other things worked with *musical viewpoints* and entropy-based music prediction for a number of years. In *Multiple Viewpoint Systems for Music Prediction* [CW95], Conklin and Witten conject that "highly predictive theories will also be good generative theories" [CW95, p.54]. In other words, if, given some melodic context, we are able to predict how the sequence of notes will continue, we will also be able to generate a good melody from any other given context. The question whether the presented learning of melodic continuation *does* generalise nicely to melodic invention in general, is not completely answered in the article, as the authors admit. Nevertheless, two major ideas are presented: to use an entropy-based predictive model as a generative mechanism for music, and to use a multiple viewpoint system to analyse the musical material from a multitude of perspectives simultaneously.

Conklin and Witten distinguish two kinds of approach to the construction of a generative theory for a musical language: the *knowledge engineering* approach, and the *empirical induction* approach. By knowledge engineering is meant the definition of rules and constraints, that are explicitly coded in some logic or grammar. Empirical induction, on the other hand, attempts to develop a theory through analysis of existing compositions. Conklin and Witten remark that "The knowledge engineering approach was discarded after careful consideration. There are too many exceptions to any logical system of musical description, and it will be difficult to ensure the completeness of an intuited theory." [CW95, p.52] They then proceed to describe how a predictive theory of Bach chorale melodies can be *induced* using *context models* – a subset of Markov models.

A context model keeps account of a set of sequences defined over some symbolic event space. This event space could be e.g. an alfabet, the set of pitch classes, the set of tuples of (pitch value, note length), you name it. Each symbol sequence that the context model is presented with (and thus learns) is associated with an occurrence frequency. This allows us to predict continuations of sequences: given a context sequence $c = [c_1, c_2, \ldots c_n]$, for all events e in the event space we may now compute the probability that e will occur immediately after c. Conklin and Witten use a mixture of short and long term context models to predict musical symbols. The long term model is trained on a corpus of Bach chorales to model the melodic structure inherent in Bach chorales in general. The short term context model is transitory and local to a particular sequence that we are predicting when applying the overall model to some sequence. The short term model is being trained on the sequence we are predicting

while contributing to the prediction. It is discarded when we begin predicting a new sequence. It thus makes the combination of long and short term models adaptive, i.e. able to respond both to the influence of an entire Bach corpus and to earlier developments in the particular sequence at hand.

The idea underlying viewpoints is that several points of view may be of importance to an analysis of music. To be able to make good sequence predictions, it may not be enough to look at the pitch of successive notes, but also the length of notes; fermata; and relations between pitch and length of successive notes. Also such relations between a note and the first note in its bar, or the first note in the piece; and notes' relation to the key signature. Conklin and Witten also present some *threaded* viewpoints, which include information on a variable number of events, depending on the concrete context. In section 4.3.1, we take a more detailed look at viewpoints, both as defined by Conklin and Witten and as these are in comparison with the viewpoints we have used in our SimFinder project.

The *type* τ of a viewpoint depends on what information the viewpoint lets us see in a sequence. Type τ could e.g. be the melodic interval between notes, or the scale degree of notes, and $[\tau]$ is "the set of all syntactically valid elements of type τ" [CW95, p.58]. A viewpoint is formally defined as comprising two things. First, the partial function $\Psi_\tau : \xi \rightharpoonup [\tau]$ that is used to transform a sequence $s \in \xi^*$ (a concatenation of symbols from the event space ξ) into this viewpoint's view on s. But a viewpoint also comprises a context model on strings in $[\tau]^*$, thus allowing a viewpoint to predict a symbol in $[\tau]$ when given a context symbol sequence in $[\tau]^*$. But viewpoints may also be *linked*, constituting together a new viewpoint. This can be described in terms of their types: the combined $\tau = \tau_1 \otimes \tau_2$ is itself a type. Forming a new viewpoint in this way makes elements of $[\tau]$ tuples of elements in the respective $[\tau_1]$ and $[\tau_2]$. Using a linked viewpoint of type $pitch \otimes noteLength$ to predict a symbol thus gives us a tuple prediction result $\alpha = (pitch_\alpha, length_\alpha)$ based on a sequence $[(pitch_1, length_1); (pitch_2, length_2); \ldots; (pitch_n, length_n)]$. This sequence specifies that the first note has both $pitch_1$ *AND* $length_1$, the second note has both $pitch_2$ *AND* $length_2$, etc. The set of all primitive and linked types is partially ordered and can be represented in a lattice showing the decreasing order of generality when linking more and more primitive types, see figure 5. $\emptyset$ is the most general type, specifying nothing at all. When more types are linked, they describe something more specific. The most specific type in figure 5 is $\tau_1 \otimes \tau_2 \otimes \tau_3$. If two sequences s_1 and s_2 are equal under a viewpoint v_{12} whose type $\tau_1 \otimes \tau_2$ is more specific than the type τ_2 of another viewpoint v_2, then s_1 and s_2 will also be equal under v_2. We discuss the general to specific ordering of viewpoint types in relation to our own viewpoints in section 4.3.1.

A multiple viewpoint system then consists in analysing the music using a set of primitive or combined viewpoints, e.g. $\{\tau_1, \tau_2 \otimes \tau_3\}$. Here, the symbol predicted by the conglomerate viewpoint system must be determined on the basis of the individual predictions of each viewpoint. But each viewpoint prediction has an associated entropy, which can be seen as a measure of how certain that prediction is. Viewpoints whose prediction is more uncertain are then assigned less weight than the more certain predictions in the computation of the overall outcome.

When viewed from a grammar perspective, context models are probabilistic finite-state automata and as such a mechanism comparable in expressiveness to

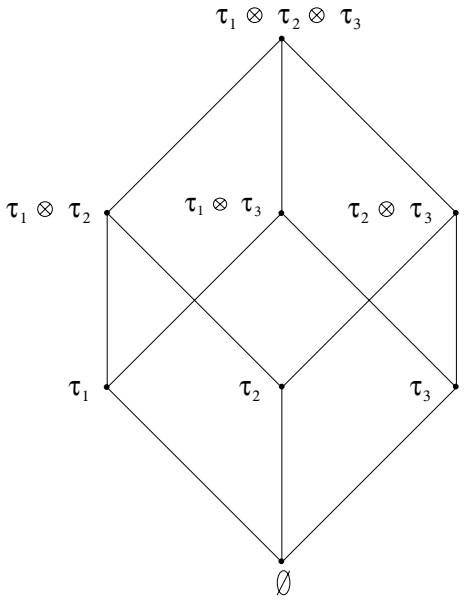


Figure 5: The type lattice for three primitive types τ_1, τ_2, and τ_3, and the possible linked types arising from them (after [CW95, p.59]).

a regular grammar. However, the exact weighting schemes used in combining the short and long term models, and in combining the different viewpoints into an overall prediction, may improve the strength of this description method.

Knowledge engineering vs. empirical induction We would like to comment on the Conklin and Witten remark cited above, that "the knowledge engineering approach was discarded after careful consideration." We think that the knowledge engineering approach cannot be completely given up in the area of music. An empirical theory induction that is blind to domain specific knowledge is bound to be less successful than one that is aware of rules and constraints inherent in the domain. The subject of analysing and composing music *does* seem hardly penetrable by intuited logical systems, because music listeners and composers have lots of tacit knowledge, i.e. knowledge inherent in our musical ability, but which is to a large extent unformulated and informal; and rules exist, in music as much as elsewhere in the dirty real world, to be broken. It is hard to achieve completeness not only of an intuited theory of music, but also of an *empirically induced* theory. There is a large amount of work left to be done in the way of representing music formally, before machine learning techniques may satisfactorily solve music analysis and composition problems. This representation problem is an area where knowledge engineering drawing on the work of music theorists seems to be the most promising path ahead.

This is exactly why multiple viewpoint systems is a good idea. As a matter of including domain specific knowledge – in this case knowledge of where musical structure is most likely to be found – we don't see any great difference between specifying by hand a set of constraints and rules whereby music should

be analysed (as e.g. Lerdahl and Jackendoff do [LJ83]) and specifying a set of viewpoints over which patterns or probability distributions are to be found.

The central problem in both is to find a suitable representation. So representing music in all its multi-faceted and multi-layered richness in a formal, flexible and expressive way is also a knowledge engineering problem, and one to which multiple viewpoint systems seem a good attempt at a solution. They allow us to analyse structure in a multitude of more or less musically relevant dimensions simultaneously, and also, in Conklin and Witten's experiment, to examine possible cross-correlation between dimensions. We believe knowledge engineering to be of the outmost importance in any automated analysis of music, and Conklin and Witten's use of multiple viewpoints is one of the most flexible implemented examples hereof.

2.4.10 Cope (2001)

David Cope, an american composer, has made a major effort in algorithmic composition and analysis of Western tonal music.

For many years, Cope has been developing a system called "Experiments in Musical Intelligence" (EMI or 'Emmy'). The goal is to make EMI copy the style of any composer, and it has come quite a long way. The system is able to generate musical pieces in the style of Bach, Mozart, Chopin etc. depending on its input. The system is corpus driven, and is able to learn from the input pieces. But it is not only the actual musical surface from which it learns. EMI is equipped with a huge library of knowledge of Western tonal music. It uses this knowledge to make an analysis of the input pieces. From this analysis, it gains insight about the structure of the music, tonal function analysis, how it is phrased, rhythm and meter correlations with tonal functions – all sorts of information a musicologist would search for as well. In addition, it stores information about the many prominent fragments (significant patterns) used in the pieces. These are the building blocks for the new piece.

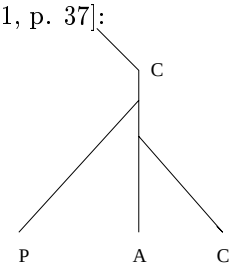
Plainly put, the system works like this: chop up and reassemble. This is of course done in a very controlled way. There are a lot of mechanisms controlling the coherence of the music. This is done according to two criteria. In a comment in Cope's book "Virtual Music" on the EMI system, Douglas Hofstadter describes the criteria: "Make the *local flow-pattern* of each of the voices similar to that in the source piece. Make the *global positioning* of fragments similar to that in the source pieces."[Cop01, p.44] The first task is related to the syntax and the form of the composition and the last is related to the content of the composition.

The *local flow-pattern* determines the way each voice should be put together from the selected small fragments available (what he calls voice-hooking). Furthermore it determines how the fragments should unfold in time as for example arpeggiation or alberti bass. This process is called texture-matching.

The *global positioning* is the task of making the small fragments connect logically to each other and to form still larger phrases also logically or semantically connected. The idea is that to insert a fragment in a piece, it can only be done if the insertion location is similar to the place where the fragment occurred in the input piece. To keep track of this feasibility, Cope uses a hierarchical system. When analysing the input piece, a hierarchy is made which determines every fragment's position in the local context (notes, measures) to

medium-range context (phrases) to large-scale (periods), and to global context (sections). The fragments are categorised according to tension-relaxation abilities. A fragment can, on each level, be considered as belonging to one of five meaning-related categories (SPEAC labels): *statement* (S), *preparation* (P), *extension* (E), *antecedent* (A) or *consequent* (C). The labelling is done bottom-up so the labels of larger sections depend on the lower. The connection between the SPEAC abstractions is given like this in [Cop91, p. 37]:

Current	Can be followed by
S	P, E, A
P	S, A, C
E	S, P, A, C
A	E, C
C	S, P, E, A



For example, a small hierarchy could be that a consequent C can consist of a statement S followed by an antecedent A followed by a consequent C. (See the tree above). Every part of the hierarchy knows the labels of its larger scale parts in which it belongs. In order to compose, the system can swap pieces that fit in the same place – have the same string of SPEAC labels determining the global context. If this is not possible, he sacrifices the requirements of the topmost level and instead tries to find a fragment with a matching string that is one letter shorter. This favours local coherence. Cope thus uses the hierarchy of existing pieces to copy as a structure for a composition. He uses GTTM-like prolongational reduction preference rules to determine the labels. The ordering in the tension-relaxation scheme seems to be quite a clever way of dividing all parts of a composition into stable subunits.

Besides this overall structural form-copying, Cope extracts some significant patterns which he calls *signatures* from the input pieces – the patterns that occur most repeatedly, also counting elaborated matches. Furthermore, he stores the conditions under which two similar passages are found. For instance if a melodic phrase is repeated a fifth higher, the program records the distance between the entries of the melody and the interval. The program can then use the idea applied to a totally different phrase – the idea is called 'template plagiarism'.

David Cope's use of grammars is given in the SPEAC labelling of his hierarchical analysis of the input pieces. His program is able to perform a kind of Schenkerian reduction of the input pieces. This analysis is the grammar which determines all tension relations.

2.4.11 Graph grammars

As described in section 2.4.5, context free grammars may not be powerful enough to describe recurrent structures in music. It seems that some form of enhancement of the basic context free production rules is needed. Roads proposes regulated grammars, Moorer introduces additional heuristics to guide the top-down composition algorithm, Winograd uses his systemic grammar, Holtzman uses re-write control rules and metaproductions, and Steedman's blues chord grammar is context sensitive. Another example of a formalism that is stronger than the context free grammar is: graph grammars.

A graph grammar is a generalisation of the 'text string' grammar. If we take a graph to be a sequence of vertices connected to each other in a string-like fashion, a graph grammar is equivalent to the normal string grammars that we use to describe formal languages. But graphs may have any topology, e.g. they could possibly contain cycles, and they can be directed or non-directed. A graph grammar contains a set of graph re-write rules that transform the graph. Each re-write rule applies to some specified subgraph σ_1 (the left-hand side of the rule); if that subgraph exists in the graph at hand, it can be substituted by the subgraph σ_2 in the right-hand side of the rule. Either such a rule specifies how the newly inserted subgraph should be connected to the surrounding graph, or some general scheme is used. E.g. if the inserted subgraph has three vertices, we could substitute each removed edge to σ_1 with a set of edges going from the same vertex to all vertices in σ_2. Graph re-write rules take many forms, e.g. vertex replacement rules or edge replacement rules.

A graphic is a graph whose vertices are attributed. What we speak of as our 'music graph' thus is a graphic: each vertex contains note related attributes such as pitch, length, etc. A formal definition of graphics and graphics grammars can be found in [HM87]. The article also gives examples of the use of graphics grammars for the description of wallpaper patterns, biological organisms, and semantic nets. Although we do use the note attributes of vertices in our graphic, we shall keep speaking of it simply as a 'graph', and the grammar produced by the segmentation algorithm (see section 5) as a 'graph grammar'.

Fahmy and Blostein (1998) Fahmy and Blostein have applied graph rewriting rules to the problem of sheet-music recognition [FB98]. In relation to our symbolic music analysis, image analysis is slightly off-topic, but we think the article gives a relevant example of the use of graph rewriting in the musical field.

An important problem in optical symbol recognition is ambiguity. A given object in an analysed score may be interpreted in several different ways; e.g. a vertical line could be either a note stem or a bar line. The best interpretation of the line depends on interpretations of the surroundings. If there is a note head very close to the line, it is more likely to be a note stem than a bar line. An optical symbol recognition method outputs a set of *primitives* – the objects it has located – each labelled with a set of possible interpretations, e.g. {note stem, bar line}. If each label is also associated with a likelihood, we can use a *stochastic relaxation* algorithm to modify the likelihood vectors according to the surroundings, in order to approach a unified, disambiguated interpretation of the entire score. Fahmy and Blostein's approach, on the other hand, is an example of *discrete relaxation*, in which successive steps of the algorithm simply rule out interpretation possibilities of the primitives, e.g. by deciding that the vertical line cannot be a bar line, and deleting the 'bar line' label from the set of possible interpretations of the vertical line's primitive.

The initial possible interpretations of primitives are represented as vertices in a graph, where edges represent relations between primitives. E.g. a vertical line cannot be both a note stem and a bar line at the same time, so the two vertices 'note stem' and 'bar line' would occur in the graph with an 'exclusive' edge between. This is a relation of mutual exclusion that exhibits the ambiguity of interpretation of the vertical line while constraining the overall interpretation to these two possibilities. The algorithm presented by Fahmy and Blostein employs

graph rewriting rules to successively transform the graph of primitive interpretations to a less ambiguous graph. The domain knowledge used to rewrite the graph is based purely on hard constraints derived from music notational conventions, not on soft constraints resulting from the interpretation of e.g. the relative spacing of primitives, or higher level musical knowledge of harmony, melody etc. The graph rewrite rules state that in the presence of a given subgraph σ_1, we may replace it by another subgraph σ_2. What is embodied in this set of rules, then, is a mechanism to reason about the semantic properties of music primitives and find a best interpretation of them, given knowledge of the common music notation.

The reasoning proceeds in successive stages where the rules applicable on later stages depend on the resolved (or relaxed) ambiguities of earlier stages. E.g. when we reason about the number and total length of notes in a measure, which is constrained by the meter, we need to know the note lengths of the different notes. The note lengths, then, must be decided in a stage prior to the measure division of the score. According to the authors, this gives a hierarchical model of the input score. The constraint relaxation method is a way to locate 'stable intermediate forms' in the sense of Simon (see section 2.2.1), from the combination of which more complex forms are built.

2.5 Summary

We have described a number of ways in which it makes sense to describe music grammatically.

Music is structured. Parallelism and tension-relaxation structures are important examples of ways to organise a piece of music, so that it is decomposable into elements. The sheer whole-part decomposability of music calls for hierarchical description. Hierarchy is found in many non-musical structures and may even be advantageous as a general perceptual strategy, giving the upper hand in an evolutionary perspective. In section 2.4.6, we asked if a hierarchical description of a piece is still worth the effort if it is in fact longer than the non-hierarchical description. We think that the answer is yes; it is still useful for the purpose of understanding a small, very intricate structure in terms of a longer description composed of simpler parts. Having simpler parts is essential; hierarchy is dependent on the existence of stable intermediate forms from which larger hierarchical units can be built. As an inherently hierarchical description method, grammars are suitable to describe music, though it is a non-trivial task to describe all aspects of music in a grammar. Multiple hierarchical structures are present in a single piece.

The 1970s and 1980s saw a linguistic turn in computational musicology. Formal grammars or their equivalent automata can generate/describe formal languages. But there is much domain-specific knowledge in music which we need to take into account when describing music grammatically. Grammars can be used as composition tools, as e.g. in the SSSP of Buxton et al., or in Holtzman's GGDL. They may also be used as analysis tools, to discover structural relations or to describe stylistic traits of a genre or a composer. Steedman's jazz chord grammar is an example, and the GTTM is also analytically inclined. As an analytical system, the GTTM could be improved in three respects. Our project is of course completely incomparable in scope with the major theory of the GTTM; we include no analysis in the way of metrical structure, time-span

reduction, or prolongational reduction; only grouping structure is touched upon, and rather superficially. But the SimFinder proposes ideas for the solution of the three main objections we find to the GTTM: the SimFinder achieves computability by weighting of rules and measures; through these measures and a multiple viewpoint system, it gives a limited account of parallelism in music; and it tackles the problem of treating non-monophonic music. In section 4 we shall see how well these problems are solved.

We have seen regular languages used by Simon & Sumner, and probabilistic finite automata used by Barbaud, and by Conklin & Witten; we have seen examples of context-free grammars (Buxton et al.) and arguments by Roads and Moorer that context-freeness is not enough for the description of music. Winograd, and Holtzman use augmentations of context-free grammars, and Steedman uses a context-sensitive grammar. It seems likely that musical structure depends a lot on context. A graph grammar is a stronger formalism than a context-free string grammar, and using a graph representation allows us to analyse non-monophonic music without necessarily succumbing to the sequential/parallel dichotomy. This is explained in the next chapter.

As an automation and personalisation tool for music production in the Internet age, grammars permit an extension of the concept of musical œuvre. There are also uses for automated music in interactive computer games. We have no illusion however that using musical grammars generatively to automatically produce music will produce artistic innovation. What interests us in art, after all, is that works of art express and embody something human. Moorer draws our attention to the fact that computer composition is a balance between two opposites: to preserve repetition and periodicity while denying boredom. The difference is particularly striking when you have seen the output of low, intermediate, and high order Markov methods. In an article on composition using genetic algorithms [TW99], Todd and Werner describe the same problem as a *structure-novelty* tradeoff. A computer composition should follow a set of music-semantic or structural rules that allow us to recognise the composed piece as music, but only to a certain extent. If all rules are followed too slavishly, the produced piece will be a boring pastiche of existing music. The composition algorithm should therefore introduce novelty and unexpected turns in its creation by violating rules, but again only to a certain extent. Too much novelty and lack of structure becomes unintelligible, but it also matters *in which way* rules are broken. In the next chapter, we will look closer at the notion of structure in music, but only from an analytical point of view. We do not consider automated composition further.

Concerning the question of top-down knowledge engineering approach vs. the bottom-up empirical induction approach to the construction of generative theories, we have argued that knowledge engineering is by no means superfluous. Music has multi-faceted and multi-layered structures, and multiple viewpoint systems are an example of a way to imbue a learning system with knowledge of the musical domain. The widely acclaimed GTTM incorporates large amounts of musical knowledge in the preference rules. The success of Steedman's grammar may be connected with the fact that there are clear music-semantical reasons for the production rules.

As a means to creating structure, grammars are not unproblematic: there is a difference between valid and good chord sequences generated by Steedman's grammar. Winograd also distinguishes between syntactically valid and

meaningful structures. Restraining a grammar to generating valid, good and meaningful music is tough, if at all possible. The linguistic notions of syntagmatic and paradigmatic relationships may perhaps help us in this endeavour. Steedman describes some of the *syntagmatic* relations built into his grammar; e.g. cadences and leading notes are important means to convey musicalness or to make the chord sequences musically meaningful. *Paradigmatic* relations of similarity, inclusion, or entailment are inherent in the hierarchical structure of a grammar and work together with syntagmatic relations to make a structure musical. If there is something like 'musical meaning', perhaps the inclusion of well chosen domain-specific musical knowledge in the construction of the grammar, by way of musically well founded syntagmatic and paradigmatic relations, is a means to achieve such a 'musical meaning'. So that also in this respect, it *makes sense* to describe music using a grammar.

3 Structure in music

The most used music representations vary in abstractness. Digital audio recordings we consider to be a very concrete representation, because it is very explicit about what we are actually going to hear. In the other extreme, the common music notation (CMN) is a very abstract description of music, because it leaves much room for interpretation of a written score. Interpreting a score so much that an expressive performance of it can be computer generated is a field of study in itself in computational musicology. But our subject lies further down the abstractness scale; describing music hierarchically using a generative grammar gives an even more abstract description than common music notation. Ideally, it describes not only musical structures but also the ways in which these may vary and be recombined while still being valid, interpretable musical structures.

Figure 6: Varying degrees of abstractness in music representations.

Our project is concerned with structure in symbolic representations of music. The purpose of this chapter is two-fold: first, we look at the CMN as a level of musical description and discuss requirements for the computer representation that we use in the project. Secondly, we consider different types of parallelism in music as we may find it at the level of the CMN, and how these can serve as a basis for building a hierarchical description of music.

Terminology Let's first clear a path through some of the terminological bush. We use the term *monophonic music* to refer to music composed of a single melodic line. Monophonic music does not contain any simultaneous notes – only one note can be sounding at a time. Music for only one part is often by nature monophonic, since some instruments cannot play more than one note at a time. This is true for a wind instrument, or a singing voice, but a keyboard instrument part is often not monophonic. The term monophonic is also used on music with more instruments playing in unison as for example a violin group or a choir part. Gregorian chant is by nature monophonic. Music which *does* contain simultaneous notes we denote *non-monophonic music* (see figure 7).

In the music literature, there is no distinction of importance between monophonic music and one-part music, but we need to draw it here since it is relevant to the computer representation.[10] A non-monophonic piece may consist of solely monophonic parts as for example in a wind instrument quartet.

The term *non-monophonic music* may seem a little clumsy, but we have chosen it so as not to interfere with the traditional meaning of *polyphonic music*,

[10] Hopefully, this will become more clear in section 4.3 on sequential (monophonic) similarities and section 4.4 on non-sequential (non-monophonic) similarities in music.

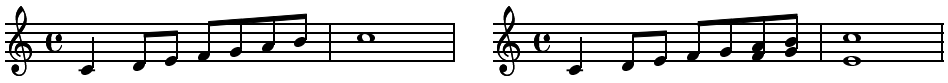


Figure 7: A monophonic and a non-monophonic piece of music

which means counterpoint based music like a fugue. Polyphonic music has parts which are melodically and rhythmically independent of each other and is thus a subset of non-monophonic music. Neither will we need to interfere with the meaning of *homophonic music*. This is another subset of non-monophonic music, in which multiple parts move in the same rhythm (as opposed to the rhythmically independent parts of polyphonic music).

Polyphonic and homophonic music are terms that say something about two extremes in musical composition techniques, whereas monophonic and non-monophonic are more technical terms which refer to the complexity of the score.

3.1 Representing music

As we have argued in the discussion of the importance of knowledge engineering versus empirical induction, the computer representation of music is of great importance. It determines how much information the music can contain, and how we are able to work with it computationally – what is natural to do in one representation can be extremely difficult in another representation, or even impossible if it does not contain enough information. If we choose a serialised form as e.g. in most file formats, this raises the question about which dimension (pitch or time) is the most important, and therefore should be given easiest access. Sometimes we want to regard the music vertically, as homophonic music (look at the chords), and other times to examine the horizontal development, i.e. the melodic lines. In a good representation it should be possible to do both.

3.1.1 Common Music Notation

The common music notation (CMN) is a result of many years of attempts to write down music as precisely as possible. It is a graphical representation, well suited to visual interpretation. CMN has proven itself very useful in notating western tonal music, since it is based on the diatonic idiom, but also nontonal music is easily readable in this format. In fact CMN is the most used format for storing pieces of western tonal music.

Curtis Roads notes that most score notations are mixtures of iconic and symbolic representations: "Graphic scores and tablature tend to be more iconic, while traditional stave notation contains more formal elements such as note heads and stems, dynamic terms [...]" [Roa79, p.48][11]. This means that some of the information (the formal, symbolic elements) contained in a CMN score are easily translated into a purely symbolic representation, while other characteristics of a score (the iconic features, e.g. 2D-placement) resist being used in symbolic ways.

The visual aspect of a score is an example of iconic features that ease the human interpretation of it: as a graphical representation of music, the CMN is two-

[11]See also our presentation of Roads' article in section 2.4.5

dimensional. The horizontal dimension represents a strict time line, and pitch is represented vertically. The position of the noteheads in combination with the key signature gives the exact pitch information. Higher sounding pitches are notated above the lower sounding pitches. From the stems of the notes and the filling of the note heads (and the dots of prolongation and ties between the notes), one can read the duration of the notes. The horizontal placement of the notes (and rests) determines the order in which they are to be played. Simultaneous sounding notes are placed above each other (this is what makes the second piece of figure 7 non-monophonic). Time is partitioned into bars of a certain duration as for example 4 quarter notes. The bars are often important to the metrics of the music.

The human interpreter thus uses a mixture of symbolic and iconic information to basically read "which pitches should sound when", but often also with a lot of performance guidelines like phrasing (slurs), dynamics, articulation and accents is specified. Some of these are added by the publisher and not the composer, so we should not assign this too much importance. It is also possible to specify textual messages (footnotes), as well as lyrics, so almost any kind of information can be added. All this information may be of importance in creating a perceivable parallelism, so in an ideal analysis, we should take it all into account.

The CMN is less suitable for specifying timbral characteristics, which is a huge issue in modern popular music. Neither non-western music as Indian raga's or music with more or less than 12 partitions of the octave is expressed well in CMN.

Diatonic information One thing that deserves a special explanation is the diatonic information inherent in the CMN score. A diatonic scale is a division of the 12 semitones of an octave into seven steps: five double-semitone steps and two single-semitone steps. A piano is constructed in such a way that scales played on the white keys are diatonic. Notes can be placed *on* the lines or *between* the lines. There is then either a single or a double semitone step between notes on two adjacent lines, just like between the white notes on the piano. The diatonic scales comprise the major scale (C to C on the white keys) and the minor scale (A to A on the white keys). The scales can be transposed to other roots, in which case the scale qualities are obtained by changing the key signature as well. The major and the minor scales are the two most prominent scales in the Western tonal music and hence the term 'major-minor' tonal music, but other scales are used.

One can also temporarily alter the steps with one or more sharps or flats, but the scale step (degree) remains the same. Although there are more than one way of notating a desired pitch, diatonic music is always written in relation to the fundamental scale steps. This is important because it allows us to construe two scales as equal even if one of them has alterations. This might not at first sight seem as a good idea, but it is a fundamental thing to be able to do. For example, a theme in major is not equal in pitch to the same theme in minor, but they're representations of the same theme nonetheless. We can discover this by looking at the diatonic scale steps they origin from (see section 3.2 on similarities).

In western tonal music we often encounter *diatonic transpositions*. For ex-

ample one musical idea can be played in the main key and then diatonically transposed one step and the again diatonically transposed one more step as for example: *C-D-E-C*, *D-E-F-D*, *E-F-G-E*. The listener hears this as a repetition of the idea, even though the note-to-note 'jumps' in the theme have different sizes counted in semitone steps. This is what musicologists traditionally denote a *sequence*. Since only notes of the fundamental diatonic scale are used, a sequence stays in the same key. In a *strict transposition* such as *C-D-E-C*, *D-E-F♯-D*, *E-F♯-G♯-E*, where every note is transposed an equal number of semitones, we also hear a repetition of the same theme, but the key changes along the way, giving us the impression that a modulation has taken place.

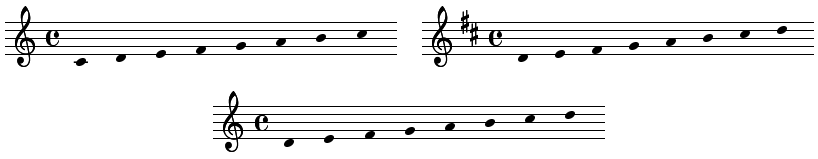


Figure 8: The C major scale, the scale transposed two semitones up to D major (note the change in key signature), and the scale diatonically transposed one scale step up (ending in D dorian scale).

Representation of diatonic information requires us to represent the overall key signature (fixed accidentals) and for each note we must know which line it is written on and any accidentals present (its spelling). A representation with just one integer for each pitch (e.g. by numbering the keys of a piano, or giving the midi pitch) is not enough, since we cannot determine if the original note was for example a *C♯* or a *D♭*. This is called the *pitch spelling problem*. We don't know of a 100% reliable solution of the pitch spelling problem, and we have chosen not to try to solve it. Instead we have taken two approaches: read music from a richer file format (MuseData) which does contain all the relevant information, or alternatively, ignore the pitch spelling information when it is not available in the source file (midi files). We present the file formats in section 3.1.2.

Let's consider an example of the importance of diatonic information. Figure 9 shows four bars (bars 37-40) from the recapitulation in the Mozart piano sonata K.V.545.

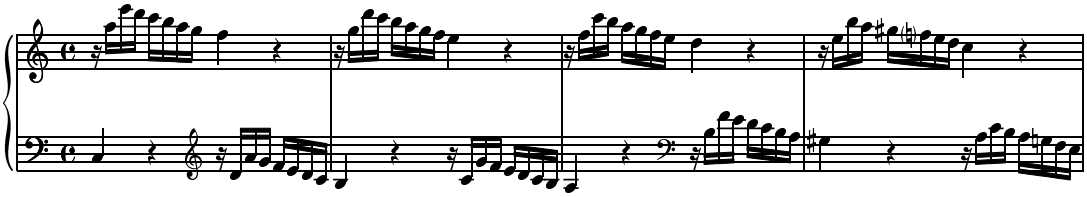


Figure 9: Diatonic sequence by descending fifths. The voice leading of each descending fifth mimics the voice leading of a dominant-tonic progression. (From Mozart Piano Sonata K. V. 545).

What we see is a sequence – one musical idea repeated over and over. The similarity seems clear to the naked eye. The 8-note melodic line is presented 8

times (the last in a slightly changed form). The melodic figure is constituted by a fifth jump upwards followed by stepwise descent. Every time the figure starts on different scale steps – the scale step one fifth below the preceding. This gives a harmonic relation between each occurrence. The harmonic background is controlled by this idea of a descending fifth. The chord sequence is then: [Am-Dm,G-C,F-Bm$^{\flat 5}$,E-Am]. [12] So we return in the same key as we started. We see that all triads that we can make with roots of an A minor scale (and consisting of notes solely of that scale) are presented (except one): [Am,Bm$^{\flat 5}$,C,Dm,E,F,G]. The exception is that the chord on the fifth scale degree (the dominant) is using an alteration of its third (the E would otherwise have been an Em). This is a fundamental thing in western tonal music. The reason is that we keep the cadence abilities intact in the E-Am relation. The altered third in the E major chord ($G\sharp$) is called the leading note, and leads one semitone up to the root in Am.

From a diatonic view, the melodic sequence looks the same (one fifth up and then descending stepwise), but when we examine the semitone intervals, there are some differences. The rising fifth is a perfect fifth (7 semitones) except when starting on B where it is a diminished fifth (6 semitones) and hence the '$\flat 5$' predicate on the B minor chord (Bm$^{\flat 5}$). An augmented fourth also is a jump of 6 semi tones. This is one of the pitfalls in the pitch spelling problem, but in this case the interval of 6 semitones is a fifth.

The descending stepwise (seconds) take also different forms. We would like to attract attention to the third one in each occurrence. It is the step from the third scale degree to the second in each chord (in Am it is *C-B*, in Dm *F-E*, G: *B-A*, C: *E-D*, F: *A-G*, Bm$^{\flat 5}$: *D-C*, E: $G\sharp$-*F* and then the Am again). In the minor chords, that interval is a small second (one semitone) and in the major chords a large second (two semitones). But in the E major, which is not a real E major but an E minor based on a minor scale with a raised third, the interval is an augmented second (three semitones). Three semitones is the same as a minor third, but this is *not* what is going on here. The interval occurs as a cause of the altered third in the dominant as explained above.

In order to be able to recognise the melodic pattern going through this sequence of descending fifths (in scale step), we have to be aware of the diatonic augmentations. In spite of the different versions of the same intervals we do hear them as versions of the same idea.

Drawing on CMN There is a lot of terminology coupled with the CMN. We believe that an ideal computer representation of music has features similar to the common music notation. It is desirable to work with music as if it was notated in CMN, so that we are able to use the same expressions and talk about music as notated in CMN. We consider CMN to be the most detailed representation for our purpose, since we concentrate on the symbolic representation of music and not audio recordings. After all, the format was, and still is, the composers' way of presenting music to us. One advantage we would like to keep from CMN is the readability of the two-dimensional score; it gives us equal visual access to both pitch and time information. This may play a role

[12] Please notice that the terms scale and chord or harmony are two sides of the same thing: defining the local tonality. A chord (for example Am) states that the underlying notes are within the A minor scale.

in the way we humans analyse and think about music. We will argue later that our graph representation provides equal access to both horizontal and vertical views, without restraining the access to horizontal and vertical views. Also, the representation of our music graph in the graphical user interface (GUI) can to some extent be "read" in a way graphically similar to a common music notation score.

3.1.2 File formats

Let's take a look at the data we are going to represent. "Beyond MIDI" [SF97] is an excellent book describing the jungle of music file formats. We have been working with two different file formats from which we are able to read a musical piece. The formats are Midi and MuseData.

Midi is a *protocol* by which computers, sequencers, sound modules, keyboards and other midi-enabled instruments of the computer age may communicate in real-time on 16 virtual channels, each representing an instrument. The midi *file format* reflects this in that all information is represented in events: midi events, system exclusive events, and meta-events. Midi events control notes' starting and stopping on the 16 channels, just as they would be controlled if receiving the events as real-time messages from another midi-enabled instrument. Only, in a midi file, there is no real-time timing to determine the onset of an event, so each event is stamped with a delta-time telling how many midi 'ticks' have passed since the last event. A midi tick corresponds to ten milliseconds. System exclusive events can be used to extend the midi protocol, a feature mostly used by hardware manufacturers. Meta events are used for track-names, lyrics, cue-points and other optional information that is not a part of the midi protocol but still handy in a file format. Key and time signature, as well as slurs, accents and other additional information can be encoded in meta-events, but there is no guarantee that a given midi file contains such information. When loading a midi file, we assume that no other information is present than the note onsets, durations and which part they belong to. Midi files are good for their extensive availability, but guarantee nothing but an encoding of the most basic pitch and length information of notes. The time resolution is the midi tick, which allows far too much expressive timing for our purposes, where a representation as close as possible to CMN is wished for. It is thus necessary to quantise midi files when they are loaded. We use the jMusic java package to transform midi files into a score-like notation (see appendix A.2 on implementation).

MuseData on the other hand takes as starting point the printed score. The compelling thing about MuseData is that it provides almost all thinkable information from the score. This includes diatonic, or pitch spelling, information. Notes are represented as spelling, octave, accidentals and duration. All expressive information (accents and relative loudness) is also encoded, and graphical information as clefs[13], bar lines (in different types) and even stem directions can be read from a MuseData file. The database contains about 900 works – mainly baroque music (Bach, Händel and Telemann) but also some Mozart, Haydn and Beethoven. There are two encoding formats, where only *stage 2* fullfills the high requirements of the encoding standard. We use stage 2 files, and this reduces the number of files offered. The files are free for download from the homepage

[13] The clefs have no impact on the pitch since the pitch is specified as described.

(`www.musedata.org`).

3.1.3 Temporal relations

Like theatre and cinema, music is a form of art that unfolds in time. It seems impossible to imagine a piece of music with no temporal extension; interleaved and intercausal structures in pitch, harmony, timbre and dynamics evolve over time and make music such a potentially complex overall structure.

In his book *Representing Musical Time* [Mar00], Alan Marsden discusses the representation of time in logic, and specifically time in music representations. A distinguishing trait of time, as opposed to other domains such as space, is that "time (almost always) is taken to be one-dimensional. There is only 'before and after' or 'past and future'. Thus, when using a logic in which times or objects-in-time are individuals, there is a fundamental binary relationship of *precedence* between these objects usually expressed with the symbol $<$, thus '$a < b$' means that a comes before b." ([Mar00], p.16) There are different kinds of temporal logic that may be divided into *in-time temporal logic* – a modal logic, where truth and falsity of a statement S is relative to the point in time when S is stated, and *out-of-time* temporal logic – a first or second order logic that describes precedence relations as seen from a non-involved, absolute point of view. But whether an in-time or out-of-time logic is used, the central relation used between events in time is *precedence* or its negation, *lack of precedence*, i.e. simultaneity. If we consider time *periods* instead of simply *events* in time, there are 7 interesting basic relations, see the table in figure 10. In addition to simultaneity and precedence (cases 1 and 2 in the table), we can distinguish 'meeting' (or 'immediate precedence'), overlapping, and the three containment relations 'starts', 'during' and 'finishes'. This necessitates additional bookkeeping, compared to an event representation with the two basic relations among events, precedence and simultaneity; luckily, the event representation is suitable for score notated music, in which time is also discretised.

1	xxx yyy	x=y
2	xxx yyy	x<y
3	xxxyyy	x meets y
4	xxx yyy	x overlaps y
5	xx yyyy	x starts y
6	xx yyyy	x during y
7	xx yyyy	x finishes y

Figure 10: The basic relations in a *period* representation of time. The table is an adaptation of [Mar00, Table 1, p.59].

Structurally, this gives rise to some natural building blocks for representations of music (and other time-based phenomena). Stringing together a number of precedence relations $a < b$, $b < c$,... between events or structures a, b, c,... gives rise to *sequence* structures such as $[a, b, c, \ldots]$; and simultaneity relations $A = B$, $B = C$,... between events or structures $A, B, C \ldots$ give rise to *parallel* structures such as $\langle A|B|C \ldots \rangle$. This is illustrated in figure 11, where the entire structure given could be described as $[[a, b, c], \langle A|B|C|D \rangle]$, square brackets '[ ]' denoting sequences and angled brackets '⟨ ⟩' denoting parallel structures.

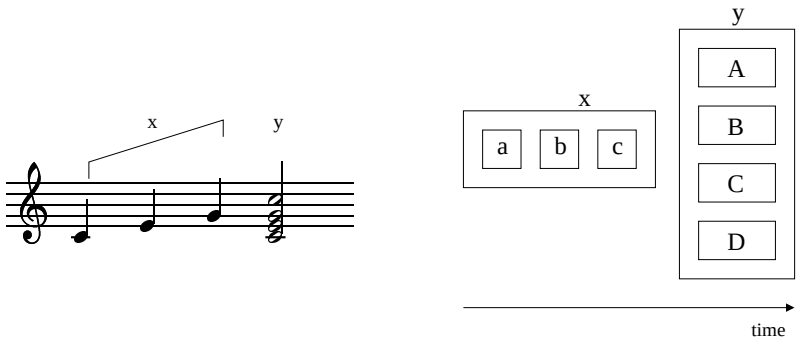


Figure 11: A sequence structure x of events or structures $a, b, c, \ldots$, connected by precedence relations, and a parallel structure y of events or structures A, B, C, D connected by simultaneity relations; x and y are also in a precedence relation $x < y$, so the entire figure could be described as $[[a, b, c], \langle A|B|C|D \rangle]$.

This fundamental distinction between 'sequence and parallel', or 'concatenation and combination', occurs in several different places in the literature.

Mira Balaban and music structures In [Bal92], Balaban sets out to define the concept "Music-piece" in terms of a hierarchical structuring of music objects and an explicit representation of time. The building blocks of the hierarchical structure are "Music Structures" that may contain elementary music structures or composite music structures. An elementary music structure is a pair (p, d) where p is a sound object that characterises the sound (you could think of it as the pitch of a note), and d is the duration of p, e.g. [A#, 4]. A composite music structure is built by a number of horizontal or vertical concatenations of elementary or composite music structures. Balaban's "horizontal concatenation" is what we referred to above as a sequence structure, whose elements additionally follow *immediately* upon each other; and the "vertical concatenation" is a parallel structure whose elements start simultaneously. The composite music structure may be placed at any starting point in time and thus relativise the times given for its component music structures.

Balaban counts the context-freeness of the music structures as a strength: "Due to the context independent nature of music structures, they can be used to describe pieces that can be repeated at different time points in a larger piece. The essence of the musical concatenation constructor is the repeated relativization of different time lines; the ability to describe structured pieces [...] derives from this essence." ([Bal92], p.121). But, as we have seen in section 2.4.5, it

seems that structural descriptions that are stronger than context-free descriptions are more appropriate for music.

The 'sound object' abstraction in the definition of elementary music structures is a powerful one. It allows Balaban's music structures to be applied to analyses of structures in tonal music, where the sound object p describes the pitch and perhaps dynamics or other properties of the sound event, such as *timbre*. From this point of view, the sound object might as well be a digitised audio recording ("a sample"), as often used nowadays in loop-based electronic music. In this way, the abstract concept of music structures can be directly applicable to the analysis and composition of sequencer/sampler-based music, although the comparison of different sounds (e.g. recorded drum sounds) is bound to be more difficult than the comparison of pitch classes. In section 6.1.9, we discuss how such an audio extension could be made to the SimFinder system.

Bernard Bel and polymetric expressions The division into sequential and parallel structures is also found in the work of Bernard Bel, who has worked on formal representations of music and most notably, the Bol processor grammars [BK92]. In *Symbolic and Sonic Representations of Sound-Object Structures* [Bel92], Bel develops the notion of *polymetric expressions*, that structure events in time. Events are located in an "event universe, a finite set of object structures with three time relations [...]. These relations are precedence, simultaneity and sequentiality. The latter relation also implies precedence; this implication is part of consistency conditions binding the three relations." [Bel92, p.73] Polymetric expressions therefore can be nested combinations of "sequences" and of "superimpositions" of objects in time. This corresponds to the sequential and parallel structures referred to above. Bel argues that it is a strength of polymetric expressions that the complete rhythmic structure need not be represented, e.g. in a superimposition of two sequences $\langle [a, b, c, d] — [e, f, g] \rangle$, the length of individual events $a, b, c, \ldots$ are not represented explicitly. Thus we cannot immediately tell what temporal relationships hold between b and f, d and g, etc. and a method is needed to infer missing temporal relations in the structure.

Bel chooses to have three basic temporal relations among events, but he also notes that sequentiality implies precedence, so in fact one could reduce the basic relations to the fundamental *simultaneity* and *sequentiality*.

Darrell Conklin and horizontal/vertical viewpoints Conklin has extended his multiple viewpoint system to apply to non-monophonic music. His data structure is yet another example of the sequential-parallel dichotomy of musical objects. Music objects M are defined as

$$M ::= Note | Seq(M) | Sim(M)$$

Conklin mentions Hudak, Balaban, and the OpenMusic environment[14] as other examples that use "a similar ontology with simple elements, sequences, and

[14]The cited articles are:

- P.Hudak, T.Makucevich, S.Gadde, and B.Whong: *Haskore music notation – an algebra of music*, Journal of Functional Programming, 6(3): 465-483, May 1996.
- M.Balaban: *The music structures approach in knowledge representation for music processing.* Computer Music Journal, 20(2): 96-111, 1996.
- C.Agon, G.Assayag, O Delerue, and C.Rueda: *Objects, time and constraints in OpenMusic.* In Proceedings of the International Computer Music Conference, Ann Arbor,

superpositions as objects. It appears that the various instances of this ontology differ mainly in the temporal overlapping restrictions imposed on objects, whether recursive embedding of objects is permitted, and whether rests are primitive objects". [Con02, p.33] The introduction of vertical viewpoints is used mainly to find significant recurrent patterns of successive vertical structures. The representation stays locked, so to speak, in either viewing a music piece as fundamentally a superposition of sequences, or a sequence of superimpositions. This is perhaps not optimal when analysing music. Our music graph is an attempt to break out of the sequential-parallel dichotomy.

The sequential/parallel dichotomy The sequential/parallel dichotomy results from splitting the representation into strictly vertical (parallel, or simultaneous) or horizontal (sequential, or related by precedence) groupings that may in turn be structured in parallel or in sequence with other elements or composite structures.

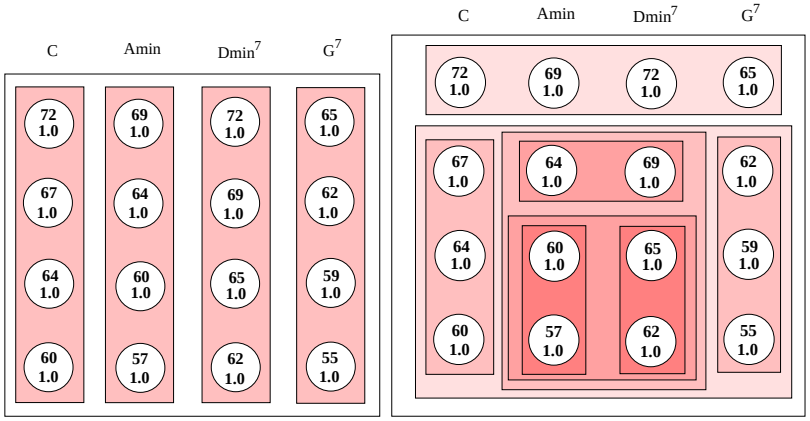


Figure 12: Two different sequential/parallel representations of the same four chords. Each circle represents a note with midi pitch value and note length.

Such parallel/sequential representations may represent non-monophonic music, in virtue of the parallel structures. But the same piece may be represented in many different ways. Figure 12 shows how the same four chords may be partitioned in two different ways using sequential and parallel structures. Each circle represents a note; the upper number is the midi pitch of the note (where 60="middle C", 64=E, 65=F, etc.) and the lower number is the length of the note (1.0 being a quarter note, 0.5 is an eighth note, 4.0 is a whole note, etc.). Implicit in both examples is a nested, or hierarchical, structure, which is somewhat more complicated in the right example.
For this project, we have tried to choose a representation that does not necessarily have an implicit hierarchical structure that may bias the analysis. We also want our representation to express the basic temporal relations of simultaneity and precedence while allowing groupings of notes that are not constricted to

Michigan, 1998, International Computer Music Association.

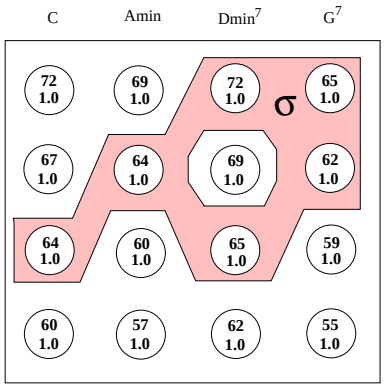


Figure 13: A subset σ of notes in the four-chord progression. σ does not have an inner nested, or hierarchical, structure to the notes it contains.

monophonic sequences or homophonic chords. E.g. we would like to be able to consider the subset σ of notes in the chord sequence in figure 13 without *necessarily* imposing a nesting of sequence and parallel structures (as shown in figure 14). The two examples of possible parallel/sequential representations of

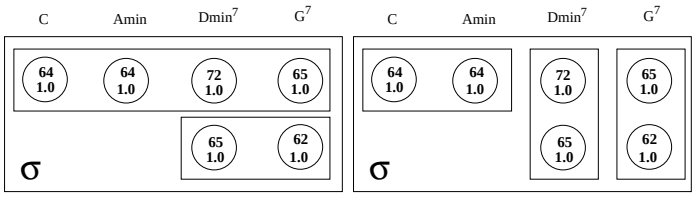


Figure 14: Examples of two possible representations of σ using parallel and sequential structures. These necessarily impose a hierarchical structure.

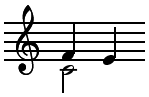


Figure 15: This musical fragment is perceived as a melodically elaborated C major harmony.

σ shown in figure 14 are not unique; there could be others.

To give one more example showing how the nature of music is not captured by this parallel/sequential dichotomy, but rather as a unification of the two, we would like to comment on the tiny musical fragment shown in figure 15. The fragment can be heard as an elaboration of a C major chord where the third has been suspended with the fourth and then resolved to the third on the weaker

beat following. In this fragment, the C and the E constituted the C major chord, so when trying to guess the underlying chord of the notes, we do not find the important notes starting at the same time, but we would have to guess what should be considered sounding together (and thus be represented together). Viewed from the melodic angle, we would miss the important harmonic features in the correlation of the two melodies. Instead we propose to group all three notes together in one structure, not preferring one or the other encoding.

To avoid biasing an analysis that uses nested parallel/sequential structures, it is sufficient to develop comparison methods that decide the equivalence of two representations, and transformations that convert one representation to another more suitable representation. An example of such representation juggling is given by Conklin in his article on vertical patterns and viewpoints: "as a prelude to finding vertical patterns, it is necessary to restructure or partition the basic encoding of a piece from a simultaneity of *Seq* objects to a sequence of *Sim* objects." [Con02, p.34]. He continues: "The music object data type is well-suited to this task because it allows the musical surface to be structured in many alternative ways." [Con02, p.34] As far as we understand, it not only *allows*, it also *necessitates* that the representation be restructured now and again to fit the purpose at hand. We have chosen instead to not necessarily impose inner vertical or horizontal structure upon subsets of notes. This can be done by representing the temporal relations of simultaneity and precedence between pairs of notes as arrows, or directed edges. The notes then become vertices in a graph, and specifying a subgraph can be done by pointing to a subset of vertices and edges. The simultaneity and precedence relations are still present as edges in the subgraph, but there is not necessarily an inner hierarchy specifying which notes are tied more closely together in a *Seq* or *Sim* structure than others. In this sense, it is an open representation. In the last section (5), we introduce a mechanism to allow but not necessarily impose grouping along any edges in the graph.

It could be objected that representing music without any inner structure is no big deal: a simple list of all notes with their onset time and other properties could do that trick. This is true, but it means that the relations between notes have to be computed every time we want to use the representation for analysis purposes. E.g. if we would like to find repetitions of a sequence of consecutive notes in a non-monophonic piece, for each note considered, we must perform a search for following notes to locate possible successors. This is computationally expensive. The graph representation has an explicit representation of temporal relations between notes, so that e.g. successors of each note can be looked up in constant time. The cost of such a data structure is additional maintenance whenever the graph is changed.

3.1.4 Brinkman's data structure

In his book *Pascal Programming for Music Research*, Alexander Brinkman describes a data structure that bears some resemblance to our own music graph (see [Bri90, pp.759-809]). It is designed to represent scores and allow flexible analysis of scores. The data structure is based on multiple doubly linked (and inter-linked) lists, constituting in effect a graph. The basis is the *spine* list whose elements are points in time at which some of the events in the score occur. The spine is like a time line. For each part in the score, there is a doubly linked

list of the notes in it. These are the *horisontal links* connecting notes in a sequential fashion. The horisontal links correspond approximately to the `FOLLOW` edges in our graph. Our `FOLLOW` edges are directed, but in the implementation of the music graph, a vertex knows both its incoming and outgoing edges, so in practice we have the same structure as Brinkman's doubly linked lists. More importantly, rests are not represented explicitly in Brinkman's linked lists, so the horisontal links are not relations of immediate precedence, as our `FOLLOW` edges, and Brinkman has no precedence links between parts. The only inter-part links are related to simultaneity. There are bi-directional *start-time links* connecting all notes beginning at the same time to each other and to their corresponding time point element in the spine. These correspond exactly to the `SIMULTANEOUS` edges in our graph. Brinkman also inserts *stop-time links* that connect notes ending at the same time. E.g. a whole note (whose length is 1.0 in Brinkman's representation) beginning at time 2.0, and a quarter note (length = 0.25) beginning at time 2.75 will have the same stop time at 3.0, so these two elements are connected by a bi-directional stop-time link. In section 4.2, we discuss the introduction of `SIMULTANEOUS_END` edges in our graph, which would be the exact equivalent of Brinkman's stop-time links.

This describes the temporal structure in Brinkman's linked list representation. An interesting feature of the linked list is the inclusion of special links that connect additional score information to the lists. E.g. the key and time signature can be represented as separate elements linked to the first element in the spine (the time line), or bar line elements can be periodically linked to the appropriate elements in the spine, thus showing when measures start and end.

The strength of such a graph representation, in Brinkman's words, is that it allows analysis programs to "move about in the score in any manner desired, looking back or ahead at will, combining horizontal and vertical motion in any manner required. This makes it possible for us to evaluate context to a degree that is difficult to achieve when dealing with one-dimensional representations such as strings." [Bri90, p.760] A graph representation therefore solves the problem of representing both sequential and parallel context at once. Our graph structure is described in detail in section 4.2.

3.2 Musical parallelism

Western tonal music is larded with repetitions and variations of small fragments. This provides a natural way for the listener to structure music into motives, phrases, passages, sections, and themes and to ultimately "understand" the piece in terms of these smaller components. Imagine a piece without any repetitions (or parallelisms); the listener must perceive new information all the time, and he may well get more and more confused, since there is no way to relate different sections of the piece, or indeed divide the piece into sections.

We are concerned with the musical surface in a symbolic representation of music. In that area, parallelism may be found at different layers in the music: the note level, motivic level, thematic level, or looking at the grand form of a piece. But far from all parallelisms in the musical surface are relevant. Parallel patterns can occur coincidentally, and they may be imperceptible; we do not wish to find those. An example of such a pattern could be one that consists of notes in different contexts (different voices or different themes) and separated far from each other in time. These are very unlikely to be perceivable and it

is doubtful that the composer is aware that they occur. An important, but difficult task is to sort the perceptible patterns from the imperceptible ones.

We have chosen not to find all patterns and *then* determine their importance, but rather to begin by narrowing the search to a set of patterns that we believe are more likely to be perceptible. This means that we concentrate solely on parallelisms whose constituent musical objects (notes) follow immediately upon each other. No musical objects in a pattern can be temporally disconnected from the rest of the pattern. This restriction increases the chances that found patterns are perceptible. But on the other hand, it also restricts us from finding certain much more advanced transformations of musical patterns, that human listeners are able to perceive. This would require us to be able to do some kind of reduction (like e.g. Lerdahl and Jackendoffs time-span reduction, or prolongational reduction) of the musical material, which we have not been experimenting with. Reduction is quite a hard problem as we will see in the next section.

3.2.1 Theme with variations

An essential thing to be able to do is to recognise a variation of a theme. The classical form type "Theme with variations" attracts our attention. This section will present a little appetiser which shows some different kinds of parallelisms that *are* relevant. Have a look at figure 16.
The figure shows the first 8 measures of a Mozart theme, and 2 out of 12 variations on this theme. The composition solely consists of these variations. The theme presents the music in its least elaborated form. In each of the variations new different musical ideas have been added to the main skeleton of the music. For example in the first variation we find the right hand melody ornamented with sixteenths (the melody can be found in the sixteenths). The originating note is never played as the first note, but often the second or fourth, and always attacked from the scale step below or above. The left hand is almost playing the theme presentation, but with some rhythmic variation. In both voices chromatic alterations have been made, giving "leading note" affinities.

The eights variation is a C minor version of the theme (the theme is in C major). Furthermore an imitation idea has been imposed: the first two bars of right hand melody are reproduced in the next two bars in the left hand (transposed a fifth). In the end, a chain of dissonances and resolutions appears. Again the notes from the melody in the original version can be found in the variation, but this time with the flat alterations which is the difference between C major and C minor.

Common to all the three fragments shown is the underlying chord progression which the notes insinuate:

Theme	[C,C,F,C,G^7,C-Am,F-G,C]
Var I	[C,C,F,C,G^7,C-Am,F-G,C]
Var VIII	[Cm,Cm,Fm,Cm,G^7,Cm-Ab,Fm-G,Cm]

The similarity with variation VIII becomes a little clearer when we look at the triad roots, relative to the tonic, C. In roman numerals, the C-relative triads of all three variations look like this: [I,I,IV,I,V^7,I-VI,IV,V,I]. In this latter representation we cannot see the difference between major and minor.

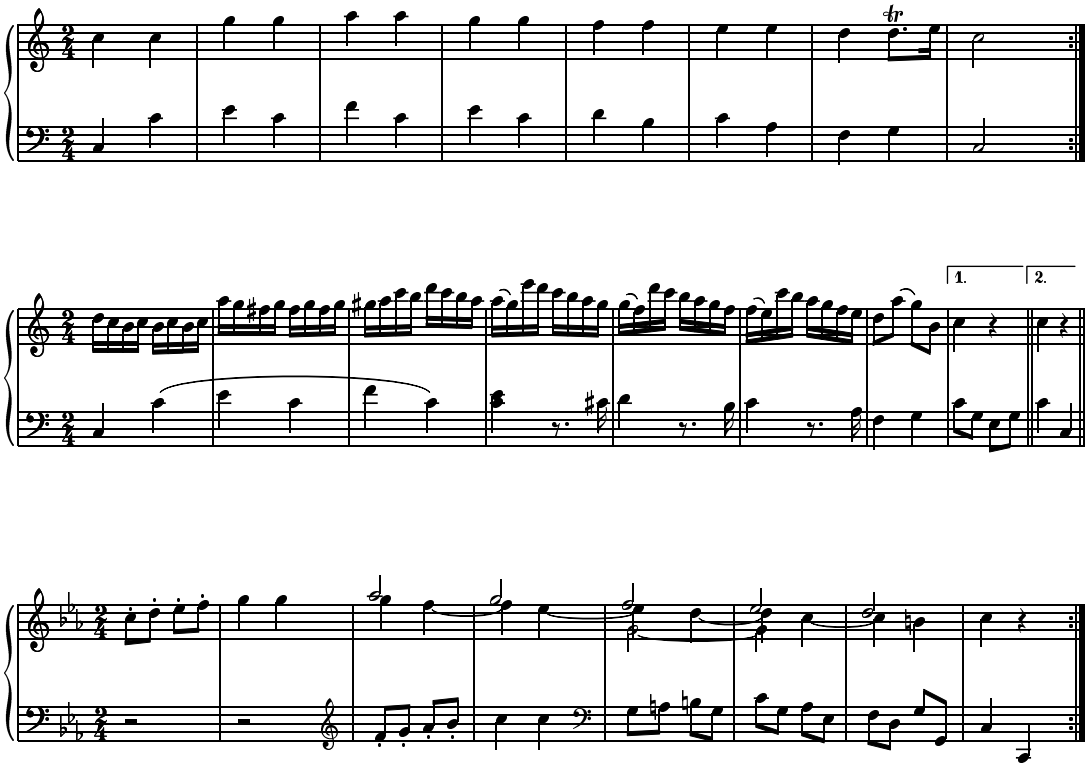


Figure 16: Theme and variations I and VIII of the first 8 measures from Mozart K. V. 265

The other ten variations of K. V. 265 (not shown here) exploit other ideas such as more ornamentation of both voices, rhythmic variations (tripletting of quarter notes), one variation in 3/4 meter instead of 2/4, and harmonic development.

The main thing here computationally is to be able to find out what the commonalities of two given pieces are, and what the differences are. In many cases, a lot of new notes are added around the simplest version of the theme. The task is then to remove the noise and find the musical abstraction under which the remaining can be construed to be similar. When listening, we tend to be able to hear these variation types clearly, and can easily categorise what is going on, whereas the computation of them is more demanding. Our minds tend to be drawn by the similarities instead of the noise. The opposite seems to be the case when computing!

Chord recognition is an example of a problem that the automated discovery of parallelism faces. Determining the underlying chord progression of the theme in figure 16 is not trivial because Mozart did not state all notes of the triads we would like to find. For example the notes of the first measure do not specify all

notes of a C major triad (but only C's), but they create an illusion of C major which most people would agree on when they hear the continuation. The F major triad also lacks an A, and the G^7 chord is lacking the G (an incomplete dominant). Chord recognition is a subproblem that can be isolated and studied. We have not implemented chord recognition and will not go deeper into it.

Hopefully, this section has illustrated a few of the difficulties in algorithmically constructing sensible reductions of a musical surface. We will now look more closely at different types of parallelism.

3.2.2 Parallelism types

The illusion of parallelism can be created through repetition, simple transformation, elaboration and simplification. The artistic effects can be varied infinitely. For example, a repeated note can be varied according to many parameters, and still be construed as a repetition of one note: duration, timbre, loudness, octave. A phrase (consisting of notes) can be transformed into other notes so that it only resembles the original according to for example pitch contour, pitch interval steps (diatonically or exact pitch interval) or rhythmic structure. We would like to extract the defining features or *transformations* which create this illusion.

The term 'transformation' is used by [WHC91] when a musical object can be inferred from another musical object in a perceptible way, and it is possible to specify the transforming function. One musical fragment can be transformed to another musical fragment, and the *transformation function* determines the type of their relation, describing in what way they can be said to be different inflections of the same musical idea. In other words, the transformation function specifies the paradigmatic relationship between the two fragments[15]: a relation of similarity, inclusion or entailment.

We use the term *transformation* to denote repetition, simple transformation, elaboration or simplification.

Repetition and simple transformation By *repetition* we mean strict repetition of a phrase. All notes in the two versions of the phrase must be pairwise equal in a particular order. This is what demands the least effort to find. By *simple transformation*, we mean transformations which do not remove or add notes. We will list some of the simple transformation functions. These functions are quite common means of creating parallelisms. Let us call the two musical objects σ_1 and σ_2. Then σ_1 can be transformed into σ_2 by the following means:

- Exact repetition. No transformation at all. The notes of σ_1 and σ_2 are the same, so this is in fact an identity function.
- Transposition. All notes of σ_2 have been displaced by the same number of semitones in the same direction.
- Octave transposition. This is a special case of transposition where each note in σ_2 is displaced 12 semi tones (one or more times) in the same direction. This is also perceptible as a special case of exact repetition, since we tend to perceive octave transposition as almost an exact repetition.

[15] See section 2.4.6 on paradigmatic and syntagmatic relations.

- Diatonic transposition. All notes in σ_2 have been displaced by the same diatonic interval, for example a fifth, or a second.
- Change in modality. This is in fact the same as a change of key. The note spellings of σ_2 are the same, but the key signature is changed. For example the change from C major (no accidentals) to C minor (three flats).
- Change in durations of the notes. The notes of σ_2 have the pitches, but different duration values. The duration values can be changed according to a fixed scale (scaled durations), or a repeated pattern of duration values (a new rhythm idea), or the changes can be arbitrary.
- Same scale, but octave displaced some times. The melodic qualities of σ_1 remains intact in σ_2, even though some jumps are not the same.
- Harmonic imitation. σ_2 is a harmonic imitation of σ_1. This is a practical problem which arises when making imitations. The intervals of σ_2 are forced to be slightly changed to stay in key. The melodic contour remains the same. The connection between *dux* and *comes* in a fugue is of this type (see section 5.5.2 on our analysis of a fugue for a further explanation).
- Change in meter. For example: σ_1 is in 4/4 and σ_2 in 3/4. Each measure is then changed to fit the new meter (so the number of measures is the same). This is a special case of change in durations.
- Inversion. Every interval of σ_1 is transformed into the negated interval in σ_2.
- Retrograde. σ_2 is σ_1 played backwards.

To locate repetitions and simple transformations in a piece, we must both find the locations of the two variations and find the simple transformation function that relates them. The problem is: given two musical structures, can one of them be transformed into the other? This is not made easier by the fact that more than one simple transformation can be applied to a musical structure. For example, a variation which is both transposed and rhythmically altered will be equal to the original under a combined "diatonic transposition *and* change in duration" transformation. In a multiple viewpoint system as ours, it is natural to describe these different simple transformations using viewpoints and their related view comparators. The task then is to find the viewpoint under which two passages are equal.

The ways in which viewpoints are combined varies between multiple viewpoint systems. In Conklin's system, conjunctions of viewpoint like the "diatonic transposition and change in duration" example above can be defined and thus form a lattice, as described in section 2.4.9. We have resorted to numerical combinations of viewpoint evaluations in similarity measures. We describe our own implementation in section 4.3.1.

Elaboration and simplification Simple Transformation functions are relatively easy to describe. Another thing is elaboration and simplification. Elaboration and simplification are complements of each other and can be described as transformation functions that are allowed to add or remove notes during the transformation.

The notion of elaboration is very important in music. For example in baroque music, decoration of long notes is almost a necessity. This gives an awful lot of possibilities. *The New Grove Dictionary of Music and Musicians* [Sis01] proposes some characteristic ways to do these elaborations, but we are not able to describe elaborations as precisely as the simple transformation functions mentioned above. The dictionary proposes:

> **Ostinato variations**. Variations built upon a short pattern of notes, usually in the bass register, which functions as an ostinato.[...]
>
> **Constant-melody**. A melody, usually widely known, appears intact or with only slight embellishments in every variation, moving from voice to voice in the texture.[...]
>
> **Constant-harmony variations**. The harmonic progression takes precedence in retentive power over the melody.[...]
>
> **Melodic-outline variations**. The theme's melody, or at least the outline of its main notes, is recognizable despite figuration, simplification (unfigured variation) or rhythmic recasting.[...]
>
> **Formal-outline variations**. Aspects of the theme's form and phrase structure are the only features to remain constant in this predominantly 19th-century type. Phrase lengths may expand or contract within the general outline, with harmonies usually referring to the theme at the beginning and end of a variation.[...]
>
> **Characteristic variations**. Individual numbers take on the character of different dance pieces, national styles or programmatic associations.[...]
>
> **Fantasy variations**. In this 19th- and 20th-century type, occasionally used as a title, the variations allude to or develop elements of the theme, especially its melodic motifs, often departing from any clear structural similarity with it.[...]
>
> **Serial variations**. Modification of a serial theme (a 12-note row or some slightly longer or shorter configuration) in which figuration and accompaniment are derived from the row. The structure of the theme usually remains constant.[Sis01]

The descriptions here are rather more vague than those for repetition and simple transformation. Elaboration and simplification can concern basslines, melody lines, harmony, stylistic knowledge, motivic development and serial ideals. As an algorithmic problem this is somewhat harder than finding a simple transformation function. The problem is to find out if a subset of the notes of fragment 1 are equal to the notes or intervals of fragment 2 under any of the described simple transformations. If it does, we have to find out if the resemblance is perceptible. This is not guaranteed by the similarity of notes or intervals. Not just any pattern is perceptible. Determining which patterns are perceptible is another field of research; we leave this discussion to the study of psycho-acoustics.

To compare one fragment, which is a chord, with another fragment, which is a melody, could possibly be done in many imaginative ways. A musically relevant way is to construe the melody as notes contributing to one chord. The task is then to compare two chords. It might not make any sense to compare a ten note melody seen as a ten-note chord with a three note triad. It is sometimes necessary to reduce some notes away (for example the doublets or 'less important notes'). Again the notion of reduction (to be able to rule out the less important notes) before comparison proves to be useful, but it is a difficult

problem to solve algorithmically. A simple approximative method to reduce more extended musical fragments into comparable structures is to pick out only notes on metrically strong beats. They are often important in determining larger scale parallelisms.

In general, to be able to compare two fragments, they must be seen from the same point of view, using the same abstraction over the data. We have defined a number of abstractions in the form of viewpoints, but only repetition and simple transformations can be found using these. We have no viewpoints that make decisions on the importance of individual parts of the compared fragments, i.e. reduce them (remove notes) in order to find elaborations or simplifications. We thus concentrate on repetition and simple transformation.

Cambouropoulos (2000) There are, however, examples of methods that find elaborations. Cambouropoulos [Cam00] works on string encodings of music. A melody is then a string pattern. He proposes a way of solving the elaboration problem – the filling and thinning of patterns – in monophonic music. Given two melodies to examine, he keeps track of the net effect of the jumps (adds the differences in semitones) that appear from one note and to the rest. When each melody eventually has made the same jump, he groups the notes together as one jump and advances to find the next jump that they might have in common. The starting point is to take one interval (two notes) in the first melody and to compare it to as many notes in the other melody it might take to reach the same net interval. The difficult part is for each interval to decide which melody has the filling notes and which one that doesn't. The algorithm might do well on some cases, but is apt to find insignificant patterns.

Cambouropoulos uses a strategy in finding patterns in music which is based on finding all exactly repeated patterns of all lengths. When one tries to do the same with approximate matchings even more candidates can be found. The patterns are candidates of being significant or perceptible.

He can then determine a prominence value to each of the found patterns according to the following factors: prefer longer patterns, prefer most frequent patterns and avoid overlap. The task is to find a suitable way to balance the factors into a suitable significance criterion.

Smaill, Wiggins, and Harris (1993) We would like to mention the approach to similarity search presented by Smaill et al. [SWH93] because it resembles our own so much. The article presents a segmentation algorithm that uses an abstract data type. The abstract data type is intended to be independent of musical style, or even of the tonal system used. It is also intended to function as an abstract interface between the particular music representation that the source material is encoded in, and the analysis program working on it, thus facilitating the exchange of one representation for another, or of the programming language for the analysis program. The data type allows for groupings of notes or of other groups in so-called *constituents*.

The segmentation algorithm, like ours, works on similarities. It is due to Nattiez, who adapted it from Ruwet [16]. The Smaill et al. version uses four

[16]The authors refer to two books by the french musicologists: J.-J. Nattiez: *Fondements d'une semiologie de la musique*, Union Generale d'Editions, Paris 1975, and N.Ruwet: *Langage, musique, poesie*, Editions du Seuil, Paris 1972.

different similarity measures:

1. Identity - the two passages are completely identical
2. Longer identity - apart from the length of the first note, the two passages are completely identical
3. Transpose - a constant number of semitones
4. Loose transpose - a transpose where note durations may differ.

First the piece is searched for completely identical phrases, i.e. using the strongest similarity measure, Identity. If such a repeat is found, it is called a *motif*, and all other occurrences of the motif are searched for again using the Identity measure. These other occurrences are called *derivations* of the motif. The motif and all its derivations are removed from the search space, which is now searched again for derivations of the motif, but according to the other three, weaker, similarity measures. Found derivations are labelled according to the measure used and then removed from the search space. This procedure repeats until no two similar phrases can be found any more. The program outputs a list of the segments found in the piece.

The similarity measures *Identity, Longer identity, Transpose*, and *Loose transpose* are all examples of repetition and simple transformation. As described in the next sections, our segmentation method also uses exact repetition and simple transformation to search for a motif (the SimFinder, see section 4), and then finds all occurrences of it (the SimSegmenter, see section 5). In section 5.4.3, we shall point out the detailed differences between the above described algorithm and the one we use.

3.2.3 Parallelism as a basis for structure

Hopefully, the last section has made it clear that it is not an easy task to identify all musical parallelisms. We choose a subset of the described parallelisms to look for stable intermediate forms[17] to build a hierarchic representation out of. As we have seen in chapter 2, musical structures seem to contain both syntagmatic and paradigmatic relations. The latter are relations of similarity, inclusion or logical entailment between tokens of a structural organisation[18]. But this is exactly what stable intermediate forms are, tokens, or building blocks, of a structural organisation of some subject matter.

That such tokens exist should be no surprise. Similarity or parallelism is quintessential to musical structure and occurs on many levels. Simple recurrence of beats is the basis of rhythm, and repetition of longer patterns of notes or beats in exact or modified copies lets the composer build a larger inner structure to the music in which one segment of the piece may (and in fact is intended to) be construed as a parallel to or a development of another segment present in the piece. Without any inner similarities, music is bound to be unstructured and therefore very difficult to digest.

As mentioned in section 2.4.7, Lerdahl and Jackendoff do not define parallelism, although four of their preference rules (one from each category) rely on

[17] See section 2.2.1 about Herbert Simon's article on complexity and hierarchy.
[18] See section 2.4.6.

its existence. This thesis is an exploration of how we could build a structural analysis of music using a method that not just incorporates but is *based on* parallelism. We construct measures of monophonic and non-monophonic similarities using a multiple viewpoint system. The similarity measures are then used to search a music piece for parallelisms that we may use as building blocks in a hierarchical description of the piece.

Ockelford (1991) In his article on the role of repetition in perceived musical structures, Ockelford describes how parallelism produces an illusion of musical order.

"The degree of ordering we perceive is proportional to the fidelity with which the second passage duplicates the first.[...] With successively freer imitation, the impression of order weakens." [Ock91, p.139] When we hear two passages, we hear the second in relation to the first occurrence. According to some criteria, we mentally connect the two passages, and the importance assigned to this connection is relative to the strength of the criteria – do we have an 'exact repetition' or only 'maybe some resemblance'.

Ockelford cites musicologists[19] who claim that the importance of repetition comes from the nature of the art form. [Ock91, p.139] If we omit all 'extra-musical' information such as lyrics, the music is a 'self-contained' art form. We cannot *directly* associate musical expression as such with the phenomenal world – which is otherwise a central characteristic of the other art forms.[20] Therefore our mind's relentless requirement of association or reference can only be satisfied through self-repetition – iteration is the only outward sign of identity available. Therefore it makes sense to organise music in terms of its resemblance to itself. The musical means by which such inner repetition or similarity can perceptually be created and heard are many, as we described in section 3.2.

3.2.4 Limitations to a parallelism-based analysis

Far from all parallelisms of the above mentioned kinds are perceptually relevant. What is and what is not perceptually relevant is a central issue when finding parallelisms in music, and if we had a simple answer to this problem, we would state it here and go on to apply it. We cannot claim that every exact repetition or transposition found by our SimFinder system is perceptually relevant. There are always small fragments here and there that are in fact equal, even though we do not perceive them because we naturally focus on other similarities.

Neither do we claim all perceptually relevant parallelisms to be encompassed by the list given. One could imagine that two passages be construed by the human ear as similar because they contain some very abstract tonal material but have no other similarity whatsoever, e.g. two clusters of randomly chosen notes. If each of the random clusters are tightly grouped and surrounded by otherwise comprehensible music, one would probably find the two clusters to be similar in the sense that they differ very much from their surroundings. This is an example of a perceptually relevant parallelism that might be captured by an extension of the SimFinder system. The SimFinder system is described in the

[19] E.g. Selincourt, Schenker, Sessions, etc., see the article [Ock91, p.139f] for further details.
[20] See [Slo85, p.57f] for a further discussion of reference and meaning in music.

next sections[21]. But as we have chosen to focus on Western tonal music, the abstractness referred to in this example is scarce in the music we shall use as source material.

One kind of parallelism that we most certainly cannot aspire to locate is similarities based on musical experiences which we interpret with reference to the phenomenal world, such as "A and B are alike because A sounds like the purl of a brook, while B is like the roar of a wild river". The extra-musical pictures, stories or ideas that the listener associates with the music, we cannot hope to approach by any means. These associations are very subjective by nature, and are also based on so many psychological parameters that it is yet impossible to analyse automatically. For the brook/river example, our only hope is that whatever in the musical structure made us associate both A and B with flowing water, does actually resemble each other in other, more structurally concrete ways too, so that we may capture it using combinations of simple transformations.

There are also more practical difficulties in segmenting music according to repetitions. We are not able to segment sections of the largest scale, because they have no repetitions. For example, the Bach chorale 'Jesu, meine Freude' has nine phrases: $ABCABCDEA'$. Since ABC is repeated, we are likely to locate the A's (including the variant A', which has the same melody voice as A), B's and C's. But we have nothing to compare D and E with, so we cannot abstract them into patterns of their own.

Sloboda (1985) Sloboda also accepts that patterning (by which he means consecutive repetitions of one fragment) is integral to music. But he believes that it cannot on its own, give a full account of the structure of a musical composition. Music *does* use patterning to achieve structural goals, "but the structural coherence of a piece of music is not to be found solely within the principles of patterning."[Slo85, p.56] The issue here is the difference between form and content. Sloboda's point is that a repeated pattern can carry any kind of expression. The patterning principle does not have starting points and goals built into them. Simple repetitions of any pattern does not guarantee any sort of evolution in the sense of 'goal-directed motion' in the music. Sloboda gives the example of M.C.Escher's famous drawings, showing patterns of e.g. animals, where the animal is transformed from one side of the drawing to another. The patterning or 'repetition with small variation' mechanism in itself could be continued indefinitely, and rather mechanically, but what gives Escher's work another dimension is the fact that the animal is actually transformed from, say, a swan to a tiger. There is an origin and a goal in the pattern, which is essential to making it a worthwhile piece of art.

Sloboda points out that on a small scale (small pieces of music) it is fairly easy to characterise the techniques which control the inner transformations, but it is extremely difficult to characterise what gives a work an integrated form. Sloboda's point is that we should be very careful when trying to revert the musical syntax into a generative context. The analysis based on similarities should be used solely as a description or explanation of the listening process.

[21] It would amount to building a reasonable measure of 'abstractness' in terms of new viewpoints, view comparators, and similarity measures.

3.3 Summary

CMN, the common music notation, is the most detailed and flexible symbolic representation of music. It has a good visual readability, and it is connected with lots of terminology that is useful for music analysis purposes. As we believe it is important to include musical knowledge in an automated analysis, a tight coupling of the computer representation with specialised terminology as we find it e.g. in the CMN is a good idea. As shown in the Mozart example (see figure 9), it is necessary to be aware of diatonic augmentations to be able to recognise certain patterns as variations of each other. Pitch spelling information is available in MuseData files, but not in midi files.

There are two fundamental temporal relations that predominate in many symbolic musical representations: precedence and simultaneity of musical events. It is common to represent music using nested structures of elementary or composite sequential and parallel structures. This gives rise to the sequential/parallel dichotomy, necessitating a certain amount of representation juggling when analysing music. We have tried to build a graph representation that reduces the amount of on-the-fly restructuring necessary, while still representing the important precedence and simultaneity relations between musical objects explicitly. It solves the problem of representing both sequential and parallel structure at once.

Inner structure in a piece of music lets us understand it more easily. Repetition and variation gives an impression of order and development. Music as an art form can be self-contained without reference to the phenomenal world, so iteration is the only outward sign of identity available. Therefore it makes sense to organise music in terms of its resemblance to itself.

We want to segment music into a hierarchical structure that we are able to describe grammatically. Our goal of segmenting music into a structure of similarities is first of all of analytic character, so we do not discuss the generative capabilities of grammars for composition purposes. To build a grammar, we need to find stable intermediate forms – subunits of which to build larger units. Musical parallelism provides us with a basis for finding such subunits. Under the heading of parallelism, many different kinds of variation is possible. These are the paradigmatic relations among grammatical units that we base our grammar upon.

Variation is the whole point of the classical form type "theme with variations". Variation can be melodic ornamentations, rhythmic variations, harmonic ambiguities, etc. When comparing two musical fragments, we would like to find out if there is a transformation that connects them. Given two musical structures, can they be derived from some common musical idea? If so, under which transformation are the two structures equal? We divide transformations into two categories: *repetition and simple transformations*, and *elaboration and simplification*. The latter transformations may also add or remove notes, and they are considerably more difficult to implement. We concentrate on repetition and simple transformations, which we implement using a multiple viewpoint system. The segmentation algorithm is very similar to the one presented by Smaill et al. (see p.51).

We have outlined in this chapter how we attempt to solve two of the interesting problems left from the GTTM: for the representation and analysis of non-monophonic music we use a graph, and for parallelism we concentrate on repetition and simple transformations, using a multiple viewpoint system. The

next section will describe our solutions to these two problems in more detail and thereby show how we achieve computability in the SimFinder system – the third point of improvement we have mentioned in relation to the GTTM.

4 The SimFinder system

This and the following sections (4.2, 4.3, 4.4, and 5) will present our ideas about searching for structure in music. Unless otherwise specified, the entities described (e.g. viewpoints, view comparators, similarity measures, the genetic algorithm) have been implemented in a java program and run on a number of pieces of music.

4.1 Overview

The java program is called SimilarityFinder, or *SimFinder* for short. In summary, SimFinder builds a graph representation of a piece of music and then uses a genetic algorithm (GA)[22] to search for similarities in this mother graph. The GA is populated by *similarity statements* each expressing that there is a similarity between two specific subgraphs of the mother graph. Such a statement can be more or less correct – either the subgraphs really are similar or they're not. Similarity statements are evaluated using *similarity measures*. A similarity measure is an arithmetic combination of the results of a number of simple comparison methods, called *view comparators*. A view comparator compares the two subgraphs when seen from a specific *viewpoint*. The thought is to have the GA optimise the correctness of similarity statements, or put another way, search for the best matches according to a given similarity measure. Similarity statements, similarity measures, view comparators, and viewpoints will be described more thoroughly in sections 4.3 and 4.4.

4.1.1 Design

An important concern in the design of the SimFinder system was modularity. The central subject of exploration in the SimFinder is the following question: which viewpoints can be combined, and in what way, to produce similarity measures that enable us to locate relevant similarities? At the outset, we had intuitions but no answers to this question. Therefore it was a central design issue to allow for quick and easy invention and substitution of view comparators and similarity measures, so that we may also compare and weight the different measures and viewpoints involved. The SimFinder system is prepared to accept new similarity measures, view comparators, and viewpoints defined as static instantiations of anonymous subclasses of the similarity measure and view comparator classes.

As can be seen in the system overview UML diagram in figure 64, Appendix A.1, the pattern of a class A having two subclasses "Sequential A" and "Non-sequential A" is a recurring one in the design. This dichotomy between *sequential* and *non-sequential* reflects the fact that we have explored two different comparison methods. Searching for similarities between sequential subgraphs inside the mother graph can pinpoint similarities inside a single part or between different parts, or point the way to more complicated (non-monophonic, and therefore non-sequential) similarities in the piece. But in order to really search for non-monophonic similarities, we found the need to develop comparison methods proper to non-sequential subgraphs. Our approach to non-sequential sub-

[22] For an introductory book on genetic algorithms see [ZM02] or [Mit01]. Our genetic algorithm is described in section 4.3.4

graphs is a development of the sequential approach, using the non-sequential versions of similarity statements, similarity measures, view comparators, and viewpoints. But there are different constraints on sequential and non-sequential subgraphs, and the comparison algorithms are much more complicated for the non-sequential subgraphs. The following two sections (4.3 and 4.4) deal with sequential similarities and non-sequential similarities respectively.

A few notes and details about the implementation may be found in Appendix A.2.

4.2 Graph representation

As our main focus is on tonal music, such as it is represented in a score, we can conveniently use a discrete event representation of time. There is no performance related information included in the way of exact timing or dynamics information for each note, so the additional bookkeeping involved in measuring and comparing periods of time[23] is unnecessary.

Representing all precedence relations between all notes in a piece can be overwhelming and is bound to include much redundant information. Given the transitivity of the precedence relation, we may infer that *A precedes C* from the facts that *A precedes B* and *B precedes C*, so it is unnecessary to represent *A precedes C* explicitly in our representation. Notes and rests following each other in a score are related by immediate precedence, so it is natural to choose a relation of immediate precedence to connect notes and rests with in the graph. We call this the `FOLLOW` relation. Most other precedence relations can then be inferred from this information.

4.2.1 Construction of the MusicGraph

As is habitual when speaking of graphs, a MusicGraph $G = (V, E)$ contains a set V of MusicVertices and a set E of MusicEdges. In our graph representation, each note or rest is represented by a MusicVertex. A MusicVertex has attributes such as the pitch of the note it represents and its length (is it a whole note, half note, quarter note, eighth note etc.). There are other important attributes such as key and pitch spelling which are included when available from the source material. Each note has an absolute start time (or onset time). Like in usual scores in common music notation, there is nothing like temporal "gaps" between elements of the graph. Rests between notes are represented explicitly in vertices, just as they are explicitly represented by symbols in a staff notation, and thus a monophonic piece (a melody) can be represented by a string of vertices representing notes, or rests, that follow immediately upon each other.

Vertices can be connected by directed MusicEdges of different types representing different relationships between vertices. `FOLLOW` edges represent the immediate precedence relation and are the most important type of edges. When constructing a graph from a piece of music, `FOLLOW` edges are added to the graph between notes, or rests, that follow immediately upon each other, i.e. notes between which there occurs no other notes (or rests) and that do not sound simultaneously. In other words,

[23] See section 3.1.3 on temporal relations.

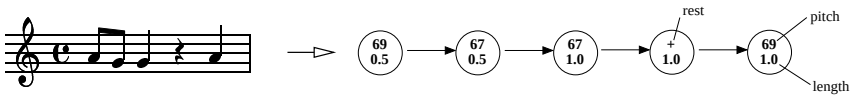


Figure 17: Representing a score in a graph. The upper numbers in the circles are midi pitch numbers ('+' indicates a rest), the lower numbers show the length of each note, where 1.0 is a quarter note, 0.5 is an eighth, 4.0 is a whole note, etc. Arrows indicate `FOLLOW` relations between notes.

Definition 4.1 *Let* $endTime(v) = startTime(v) + length(v)$. *Then* w `FOLLOWs` $v \iff endTime(v) = startTime(w)$

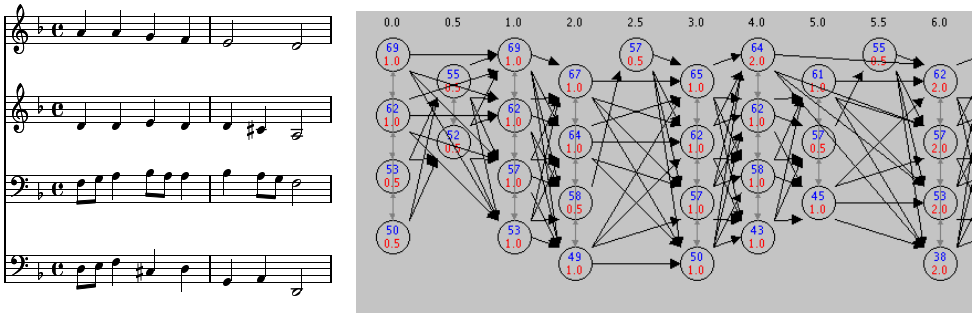


Figure 18: First 2 bars of the Bach chorale Jesu Meine Freude BWV358.

Figure 18 shows part of a graph representing the Bach chorale Jesu Meine Freude (BWV358). Notes from all parts have been added to the same graph and interconnected as if they belonged to the same part. The graph is thus made up of paths that could be melodic lines inside the tonal network of the piece. It is difficult to know if and when it makes sense to consider melodic lines that weave in and out of several different parts. Although the construction of the graph guarantees that the notes of vertices following each other in the graph do actually connect temporally in the score and therefore, to the attentive listener, it should be possible to hear such a melody, often the combination of notes from different parts will give a number of strange jumps in the melody. Particularly when different parts are to be played by different instruments, it will be harder to hear such connections because our natural ability for auditory stream segregation allows (and sometimes forces) us to separate parts because of their timbral differences. In other words, most non-monophonic music is meant to be heard as being separated into parts, although working together. Therefore it is also important to study graphs where the different parts are not connected, i.e. there are no edges between two vertices that belong to different parts. Such a graph we call a *partwise graph*, see figure 19. The graph with connections between parts (as shown in figure 18) is called a *non-partwise graph*.

Most often, each part is monophonic (e.g. the chorale seen in figure 19), but there is nothing in the graph representation that prohibits non-monophonic parts, as may be found e.g. in piano sonatas, where each hand (notated as a part) may play several notes at once.

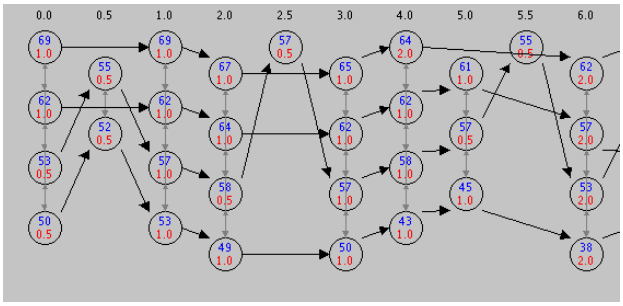


Figure 19: First 2 bars of the Bach chorale Jesu Meine Freude BWV358 represented as a *partwise graph*.

Definition 4.2 *Let $G = (V, E)$ be a MusicGraph. The group of $v \in V$ is the largest possible set $W \subseteq V$ of vertices where $\forall w \in W : startTime(w) = startTime(v)$*

A group is a set of notes beginning on the same beat. In our graphic representations of the graphs, notes in a group are arranged in a vertical line, reflecting their simultaneity. If all notes in all groups had the same length (i.e. a homophonic piece), constructing the graph would amount to finding these simultaneous groups and making a full connection of `FOLLOW` edges between consecutive groups. This is not the case, however, since e.g. the bass could be moving in twice the speed of the other parts, and would have separate one-note groups that only the preceding bass note and the following notes from all voices were connected to (see Figure 20).

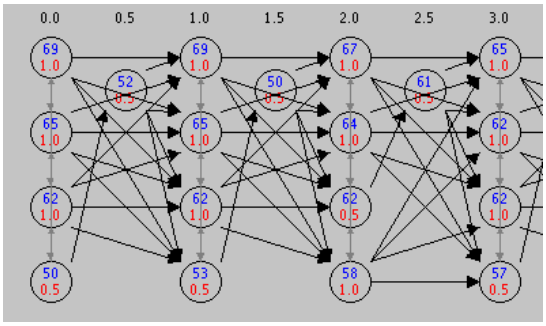


Figure 20: Bass part moving in twice the speed of the other parts. Of the group beginning at time 0.0, only the bass note is connected to the note beginning at time 0.5, since the other notes are still sounding at time 0.5 – they have length 1.0

The simultaneity of vertices in a group is represented by connecting them with `SIMULTANEOUS` edges. These are shown in the graphic representation as small grey vertical arrows. They allow us to locate chords in the graph more easily. In the screen shots, often it is impossible to see the full connection of `SIMULTANEOUS` edges between all vertices in a group because the arrows are

drawn on top of each other, but the SIMULTANEOUS relation is naturally transitive, and it should be possible to imagine the full connection from the vertical arrangement and the visible SIMULTANEOUS edges.

Definition 4.3 *w is* SIMULTANEOUS *with* $v \iff startTime(v) = startTime(w)$

Notice that the SIMULTANEOUS relation is indifferent to the end time of notes. We could have called it the SIMULTANEOUS_BEGINNING relation and introduced a SIMULTANEOUS_END relation connecting notes that share the same end time, defined by:

Definition 4.4 *w is* SIMULTANEOUS_END *with* $v \iff startTime(v)+length(v) = startTime(w) + length(w)$

But the information added by introducing SIMULTANEOUS_END edges in the graph is already implicit in a graph with only FOLLOW and SIMULTANEOUS_BEGINNING edges; SIMULTANEOUS_END groups of notes may be found by taking the notes connected to all incoming edges in a SIMULTANEOUS_BEGINNING group. E.g. in figure 20, vertices in the group beginning at $time = 1.0$ have incoming edges from the notes (69;1.0), (65;1.0), (62;1.0), and (52;0.5), which would constitute a SIMULTANEOUS_END group. Thus the information is present, though implicitly, and we stick to the SIMULTANEOUS_BEGINNING edges and call them simply SIMULTANEOUS edges.

When we begin comparing different parts of a graph with each other, we'll need some special kinds of subgraphs.

Definition 4.5 *A sequential subgraph* $s_1 = (V_1, E_1)$ *of the mothergraph* $G = (V, E)$ *is a set of vertices* $V_1 \subseteq V$ *that are connected by* FOLLOW *edges* $E_1 \subseteq E$, *where* $\forall v \in V_1$: *in-degree*$(v) \leq 1 \wedge$ *out-degree*$(v) \leq 1$.

Put informally, a sequential subgraph is a string, or sequence, of vertices connected by FOLLOW edges. (See Figure 21 for an example of a sequential subgraph).

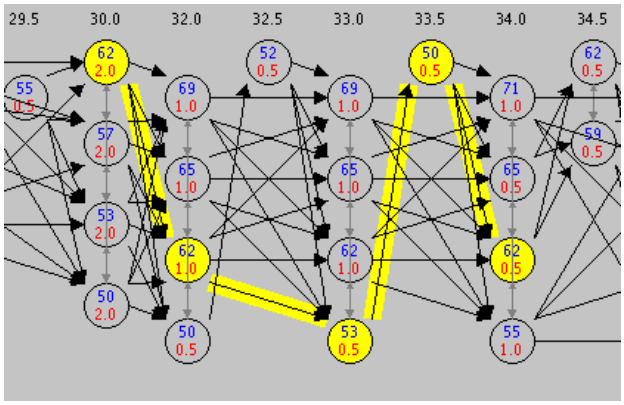


Figure 21: A sample sequential subgraph

Let $in(v)$ and $out(v)$ be the respective sets of in-edges and out-edges of the vertex v.

Definition 4.6 *A non-sequential subgraph* $\sigma_1 = (V_1, E_1)$ *of the mothergraph* $G = (V, E)$ *is a connected set of vertices* $V_1 \subseteq V$ *whose connecting edges* $E_1 \subseteq E$ *are of type* `FOLLOW` *or* `SIMULTANEOUS` *and where*

$\forall\ v_1, v_2 \in V_1 : (\exists\ e \in E : e \in out(v_1) \wedge e \in in(v_2)) \Rightarrow e \in E_1$

Informally, vertices in the subgraph that are connected by an edge in the mothergraph will also have that edge present in the subgraph. This amounts to including all available relationships between those vertices included in the non-sequential subgraph. This is required to avoid ambiguities in the subgraph. (See Figure 22 for an example of a non-sequential subgraph).

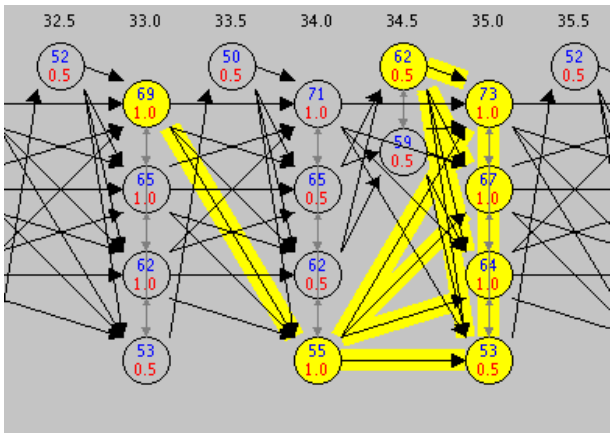


Figure 22: A sample non-sequential subgraph

The size of a subgraph, sequential or non-sequential, we define to be its number of vertices. The overlap of two subgraphs is the total number of vertices *and* edges of the mother graph that the subgraphs have in common. We will speak much about subgraphs; the mother graph is the original graph constructed from a music file in the first place.

4.2.2 Using the music graph

At present, the SimFinder system is able to build MusicGraphs from midi files and from MuseData files. A mother graph built from a midi file contains only the most basic information shown above (pitch and length of individual vertices), whereas mother graphs originating from MuseData files are richer. But even more information could be included.

Some types of information are directly connected with each note. We store this information as attributes in the MusicVertex objects. At present in a MusicVertex we always represent the basic information available from most midi files: pitch, starttime, the part to which the note belongs, and the `FOLLOW` and `SIMULTANEOUS` edges inferred from the start times of notes in general. From a MuseData file we additionally read: the pitch spelling name (C=0, D=1, E=2, F=3, G=4, A=5, B=6, rest=-1), the octave ($0 \ldots 8$), the key (a number telling the number of fixed accidentals – if the number is negative, the accidentals are flats, else sharps) and time signature (denominator and numerator). This is

used for the diatonic viewpoints described in Section 4.3.1. Of expression signs, we read only fermatas. Other accents could be stored here as well.

We could have implemented other types of information that relate to a collection of notes, or are common to all notes in the collection. These we would represent in a special object. For example we could have a measure object, or measure vertex, in the graph which all notes (vertices) in the measure would have knowledge of. The measure object should hold the time signature of the measure, and could also hold metric information. Other relevant information representable in the graph is: cue notes and grace notes, beam codes, ties, slurs, tuplets, ornaments, performance related indications, articulations and accents, and text underlay.

4.3 Sequential similarities

Our goal now is to find musically similar subgraphs in the mother graph. We could do this by picking a subgraph and searching for occurrences of it in the mother graph. There are a few difficulties and a major problem with this: we want the located similarities to be as large as possible, we want them to be situated at well-chosen spots in the graph, we don't want the located similar subgraphs to overlap, and we want to first locate the most important similarities (which to some extent can be argued to be the most occurring patterns). This complicates the first choice of a subgraph to search for. Now the major problem: searching for the occurrence of a subgraph in a graph is to solve *subgraph isomorphism*, a problem known to be NP-complete[24]. Our solution is to use a GA to choose subgraphs to compare. The GA is a stochastic search method and will not give us an exhaustive search, but it allows us to approximate an optimal satisfaction of the above mentioned requirements. The size of the located subgraphs and constraints on the location (with respect to grouping boundaries in the music) can be incorporated into the fitness evaluation of the GA. So can the overlap prohibition, and the GA is more likely to locate very frequent subgraphs than infrequent ones, because picking a random subgraph is more likely to stumble upon the frequently occurring subgraph.

But we still need to compare the musical attributes such as pitch and length of the notes. In order for the GA to be able to compare subgraph matchings, we have constructed numerical ratings of the similarity of subgraphs. Sequential subgraphs are easier to deal with than non-sequential subgraphs, so we begin with the sequential case. Inspired by the 'viewpoints' of Conklin in [CW95] and [Con02] (see section 2.4.9) we first set out to find similar strings, or sequences, of vertices inside the constructed MusicGraphs.

4.3.1 Viewpoints

A viewpoint is, well, a distinct way to see things. It is something applicable to the treated subject that allows us to focus on some particular aspect of it, like wearing a pair of coloured glasses that filters the information we receive, and gives us a special *view* on things.

Roger Dannenberg has proposed to use *views* in graphical score layout systems, where editing different parts separately poses problems of consistency between the score and the individual parts: "A view of a data structure contains

[24][CLR90, p. 960]

a subset of the information in the data structure and sometimes provides alternate or additional data to that in the data structure. The idea is to keep shared data in one place so that a change in the score will automatically be propagated to the parts, and part-specific layout information can be maintained for each part (view)" [Dan93, p.25] He goes on to suggest that views could be used to represent repeated sections of music, possibly with variations from occurrence to occurrence; or *views* could be the general mechanism to structure music: "Imagine a representation where motives are represented only once, and each occurrence is some kind of view, perhaps with local alterations and transformations, of the motive." [Dan93, p.26] This in fact is not far from our goal, although it is not what we speak of as views and viewpoints; we shall look closer at the general 'motive' vs. 'motive occurrence' idea in section 5, but for the time being, let's stick to views and viewpoints.

We take a more narrow view on viewpoints than Dannenberg. Darrell Conklin defines a viewpoint as "a mathematical function that computes features of music objects in a sequence, including features that denote relationships between objects within the sequence" [Con02, p.35]. This allows us to compare two sequences of music objects from various viewpoints, such as the pitches of the notes of the two sequences, the intervals between the pitches of consecutive notes, the lengths of notes, etc. Note that this definition does not include context models, as did Conklin and Witten's definition from 1995, see section 2.4.9. The viewpoint is now solely the function that transforms a sequence of music objects into a particular *view* on it, separated from the learning mechanism used to find patterns in views.

Instead of defining music objects in a framework of compound *Sequence* and *Simultaneous* objects as Conklin does in [Con02], we let music objects be the vertices of our MusicGraph. A sequence then is a sequential subgraph, i.e. a string of vertices connected by `FOLLOW` edges. Each vertex has properties such as pitch, length etc. so comparing two passages, represented as sequential subgraphs, amounts to comparing these properties vertex by vertex. A sequential subgraph does not contain any notes that sound simultaneously. This is implied directly by the nature of the `FOLLOW` relation which connects consecutive vertices in a sequential subgraph. Thus seeing a passage from a given viewpoint transforms the sequential subgraph representing the passage into a sequence of numbers expressing some aspect or internal relation of the entire sequence. The resulting sequence of numbers is called a *view* on the passage. Although we often speak of 'evaluation' or 'matching' in this section on viewpoints (because that is our purpose with viewpoints and views), it is actually the job of a *view comparator* to evaluate how well two views match. This is the subject of section 4.3.2.

Using sequential viewpoints To search for similarities in music, we must specify what kind of similarity we are looking for. We have defined several viewpoints, each capable of locating a different kind of musical similarity when comparing two passages. Some viewpoints are more general, catching more similarities when compared, but are less explicit about which similarities. E.g. the *pitch contour* viewpoint; as a mathematical function, pitch contour is not injective, since many different sequences of notes may have the same pitch contour. Therefore it is more general (it will capture more matches) than, say,

the *absolute pitch* viewpoint, under which only exactly identical pitches will be evaluated as equal[25].

Viewpoints can also be designed to catch similarities that are specific to the type of music they are used upon – e.g. the diatonic viewpoints described below. These viewpoints are specifically designed to capture similarities in Western tonal music.

Viewpoints based on pitch and length We shall use the abbreviations in the parentheses to refer to each of the viewpoints.

- Absolute Pitch (AP): This viewpoint looks at the midi pitches of the notes. Each note has a pitch (a numbered semitone), and the resulting view has as many members as the sequential subgraph has vertices. Two sequences of notes must have the exact same pitches to be evaluated as equal under the absolute pitch viewpoint.
- Pitch Interval (PI): the interval in number of semitones between one absolute pitch and the next. Two sequences of notes with equal pitch intervals are a direct transpose (of an integer number of semitones) of one another.
- Pitch Contour (PC): whether the absolute pitch changes upwards, downwards or stays the same between consecutive notes.
- Pitch Class (PS): the pitch class of a note is its absolute pitch modulo 12. Seen from the pitch class viewpoint, notes that are an integer number of octaves apart are equivalent.
- Absolute Length (AL): the lengths of the notes (whole, half, quarter, eighth, sixteenth, etc.).
- Length Interval (LI): since most note lengths are separated by factors that are powers of 2, we define the length interval from length l_1 to length l_2 to be $log_2(l_2) - log_2(l_1)$. This is the number of times l_1 can be doubled before reaching l_2.
- Length Contour (LC): whether the length of notes increases, decreases or remains the same from note to note.

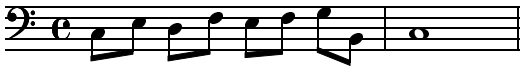
See also figure 23 on page 66 for an example of the viewpoints.

Viewpoints based on diatonic scales These viewpoints are used to search for diatonically related passages. Remember that the diatonic information tells "which note line the note was written on" in the score.

- Diatonic Absolute Pitch (DAP): The note as it is written in the score: the name (*C*,*D*,*E*,...), the accidentals (♯'s and ♭'s) and the octave. It is often the same as Absolute Pitch, but will not accept an *E*♭ for a *D*♯ (which are indeed very different things). This is useful for finding exact repetitions.

[25] We discuss the general to specific ordering of viewpoints further below under the heading 'The connection between viewpoints'.

- Diatonic Interval (DI): the change in interval diatonically (second, third, fourth etc. up/down). A major and a minor third looks the same from this viewpoint. A theme in major will hence look the same as a theme in minor (even though the major and the minor scales differ in 3 notes). This is useful for finding diatonic transpositions.
- Diatonic Forward Interval (DFI): the upwards change in note lines (note names), that is: how many steps there is to the next note name going only forward. The note names are (*C,D,E,F,G,A,B,C,D,* etc.), so from C to E is 2 (a third) and from E to C is 5 (a sixth). This viewpoint is the most robust we have so far – it is robust to both transposition and octave displacements of the melody. Imagine a theme A, and another theme B, which is diatonic transposition of A (e.g. transposed down a third), where some parts of B have been octave displaced because the instrument's ambitus[26] doesn't allow it to play it otherwise. Themes A and B would be similar when seen from this viewpoint. See figure 33 on page 89.
- Diatonic Inversion Interval (DII): The inverted (negated) diatonic interval. For example if DI between two notes is i then DII is $-i$.



Viewpoint	View
Absolute Pitch	[48,52,50,53,52,53,55,47,48]
Pitch Interval	[4,-2,3,-1,1,2,-8,1]
Pitch Contour	[1,-1,1,-1,1,1,-1,1]
Pitch Class	[0,4,2,5,4,5,7,11,0]
Absolute Length	[0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,4.0]
Length Interval	[0,0,0,0,0,0,0,4]
Length Contour	[0,0,0,0,0,0,0,1]
Diatonic Absolute Pitch	[C3,E3,D3,F3,E3,F3,G3,B2,C3]
Diatonic Interval	[2,-1,2,-1,1,1,-5,1]
Diatonic Forward Interval	[2,6,2,6,1,1,2,1]
Diatonic Inversion Interval	[-2,1,-2,1,-1,-1,5,-1]

Figure 23: Example of the viewpoints.

To make things more concrete, figure 23 shows an example of how a melody looks from different viewpoints. Notice that the viewpoints denoted with *absolute* are reading their information from each note independently, whereas the others, the *relative*, relate to changes *between* the notes. The latter kind therefore have one entry less in their view vector. Notice that under the diatonic forward interval, a sixth down (*G-B*) gives the value 2, as does a third up (*C-E*). So if the melody had ended with the two last notes one octave higher, the DFI would give the same view – octave transpositions are ignored.

[26] The range of playable notes of the instrument.

Each viewpoint is able to recognise a simple transformation function as described in section 3.2.2 on parallelisms types. By designing the right viewpoints we can search for exactly what we want. We are able to find the most basic similarities with our viewpoints regarding pitch: exact repetition (with AP, DAP), transposition (PI), diatonic transposition (DI), inversion (DII), and diatonic transposition with octave displacements (DFI). With respect to durations of the notes, we are able to recognise the exact same durations (AL), but also the relations between lengths (LI) - two otherwise equal passages where all durations of the second passage are doubled will be recognised by this viewpoint.

To be able to find elaborations and simplifications of a piece of music, we have to define some more advanced viewpoints. The problem is that there are to a less extent standard ways of making elaborations and simplification. A viewpoint is designed to recognise a single special idea.

The connection between viewpoints Some viewpoints are more fine grained than others. By inspecting the lists of viewpoints, we see that for perfect matches, some viewpoints are actually covered by others. Let $perf(x)$ be the set of pairs (s_1, s_2) of sequential subgraphs that will have a perfect match according to viewpoint x [27]. Then

$$perf(AbsolutePitch) \subset perf(PitchInterval)$$

This means that an exact match in AbsolutePitch will also be seen as an exact match from the PitchInterval viewpoint. Similarly,

$$perf(PitchInterval) \subset perf(DiatonicInterval) \subset perf(DiatonicForwardInterval)$$

and

$$perf(PitchInterval) \subset perf(PitchContour)$$

$$perf(AbsolutePitch) \subset perf(PitchClass)$$

$$perf(AbsolutePitch) \subset perf(DiatonicAbsolutePitch)$$

$$perf(AbsoluteLength) \subset perf(LengthInterval) \subset perf(LengthContour)$$

An exact match in PitchInterval (PI) will also give an exact match when we look at it from the DiatonicInterval (DI) viewpoint, and also, by further inclusion, under the DiatonicForwardInterval (DFI) viewpoint. By looking at different views of two subgraphs, we can determine what kind of parallelism we are dealing with. For example, if the `seqDC_DFI` view comparator is 0 (a perfect match) and the `seqDC_DI` is too, but the `seqDC_PI` is nonzero, then the parallelism is bound to be a diatonic transposition (by virtue of the DFI), with the exact same melodic contour (by virtue of the DI). A similar hierarchy can be given for viewpoints concerning length. This logical hierarchy of perfect matches enables us to reason about which viewpoints to use, and when, and to distinguish and label the similarities we discover in the segmentation algorithm presented in section 5.4.1.

[27] For some given view comparator, but that is unimportant right now.

Other viewpoints We could continue deriving viewpoints from the basic information in music vertices. E.g., we could have implemented a Diatonic Pitch Contour viewpoint: this is the contour of the melody as written in the score, disregarding any accidentals. The viewpoint gives the same view as Pitch Contour, when there are no accidentals in the music. This viewpoint is therefore more general than the normal pitch contour.

Conklin and Witten (C&W) describe a great number of basic and derived viewpoint types [CW95, p.63]. In the table below, we compare these with the corresponding sequential viewpoints we have defined above. The first seven C&W viewpoint types are basic types, the next eleven are types derived from the basic types, and the last four are threaded types that compare the pitch interval of notes that begin a quarter, a bar, or a phrase with the previous such note. The threaded types are therefore not defined on all notes, only at these regular temporal intervals. nav(α) means "not a viewpoint", i.e. it has not been implemented in our system, but that the information needed to construct such a viewpoint is present in graphs constructed from music files in format α.

C&W type	Our sequential viewpoint type
Start time	nav(Midi,MuseData)
Pitch	Absolute Pitch
Duration	Absolute Length
Key signature	nav(MuseData)
Time signature	nav(MuseData)
Fermata	nav(MuseData)
Delta-time from preceding note's end time	nav(Midi,MuseData)
Delta-start time	nav(Midi,MuseData)
Position in bar	nav(MuseData)
Is first in bar	nav(MuseData)
Sequential melodic interval (SeqInt)	Pitch Interval
Contour	Pitch Contour
Referent[28]	nav(MuseData)
Interval from referent	nav(MuseData)
Is in referent major mode scale	nav(MuseData)
Interval from first event in bar	nav(MuseData*)
Interval from first event in piece	nav(Midi,MuseData*)
Interval from phrase beginning	nav(MuseData)
SeqInt at bars	nav(MuseData)
Length of phrase	nav(MuseData)
SeqInt at phrases	nav(MuseData)
SeqInt at quarters	nav(MuseData)

*: the information needed to find the first event in the bar, or in the piece, is present, but in a non-monophonic piece, there might be several first notes, so the interval to *the* first note is undefined. This could probably be remedied by choosing e.g. the highest or the longest first note.

As seen in the right column of the table, if the piece is provided as a MuseData file, we are be able to construct all viewpoints listed by Conklin and Witten. There remains five of our viewpoints that have no match in C&W's

list. Length contour and Length interval are irrelevant, Conklin and Witten say, to the Bach chorale melodies they use as source material, although they may have some significance to e.g. augmentation of fugue subjects. The remaining three diatonic viewpoints we have implemented (DAP, DI, and DFI) are new viewpoints that are not mentioned in [CW95]. We think this is an important improvement. In section 3.1.1, we showed the importance of diatonic information (pitch spelling) in a search for variations. In particular, the Mozart example (see p.36) shows a variation that cannot be found if we do not know the pitch spelling but only the pitch of the notes. Figure 39, p.104, is an example of the use of the non-sequential edge comparison method, where we show two subgraphs that would not have been found to be similar without diatonic information.

Viewpoints on a music graph need not be constricted to pitch and length of the notes corresponding to vertices in the graph. E.g. one could imagine viewpoints telling how many `FOLLOW` in-edges or `FOLLOW` out-edges, or how many `SIMULTANEOUS` edges each vertex has. These numbers would not make much sense in the comparison of sequential subgraphs, since the vertices at a given index i in two sequential subgraphs of the same length would always have the same number of in-, out-, and `SIMULTANEOUS` edges. But, as we shall see in section 4.4, these are very useful viewpoints when comparing non-sequential subgraphs.

The problem with rests So far we have avoided rests. Some of the viewpoints are not really defined when a rest occurs. There is no well defined value for Pitch Interval (the difference in pitch) when one of the two musical objects involved is a rest.

A similarity between three quarternote rests and another set of three quarternote rests is not important, since there really is nothing to compare. In our implementation of the viewpoints, we have used some special values for note-to-rest and rest-to-note transitions. A note-to-rest transition can then be matched to another note-to-rest transition.

A special problem occurs when every second note is a rest (note, rest, note, rest...). Then, using a pitch interval viewpoint, such a passage can be matched with another passage of that kind, no matter what notes there actually are in the passage. We could solve this by determining if the rests are used as "commas", i.e. small breaks, where the melodic line is not broken, or "punctuations" that end a phrase. Then the "comma-rests" could be removed, and the phrase regarded as one, continuous melody. This way, the rests would not give undefined values from the small breaks. We have not implemented the comma/punctuation distinguisher, because it is a non-trivial project in itself. Instead, through the grouping viewpoint we have tried to punish a similarity statement whose subgraphs contain many rests.

Grouping Structure Searching for similarities through the viewpoints described above will not guarantee any logical or natural segmentation of the musical material, except that the compared fragments are somehow similar. We have experimented with a few *grouping preference rules* (GPR's) inspired by the GTTM (see section 2.4.7). The GPRs are supposed to show where to draw grouping boundaries. Grouping boundaries are supposed to be the natural

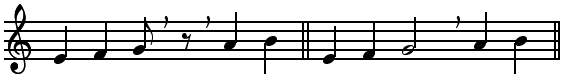
pauses for the performer to take a single breath. We use the rules to decide if a given subgraph has an appropriate extension – i.e. if it fits with the natural bounds in the music.

Please notice that in this section the word group is *not* referring to what we earlier have described as groups of simultaneous sounding notes. We use it in this section as in the GTTM.

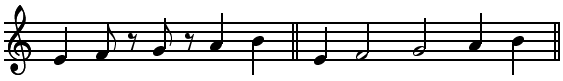
We have concentrated on two rules (and subrules) taken directly from the GTTM, which form the core in the grouping evaluation. The rules are GPR 2 and GPR 3. These rules are defined for monophonic music. GPR 2 is a proximity rule, stating that a sequence of notes $n_1 n_2 n_3 n_4$ (rests can occur inbetween) may be divided between n_2 and n_3 if:

- a. (Slur/Rest) the interval of time from the end of n_2 to the beginning of n_3 is greater than that from the end of n_1 to the beginning of n_2 and that from the end of n_3 to the beginning of n_4, or if
- b. (Attack-Point) the interval of time between the attack points of n_2 and n_3 is greater than that between the attack points of n_1 and n_2 and that between the attack points of n_3 and n_4.[LJ83]

The rules apply for example in these situations (the breathe marks suggests the ending/beginning of the phrases):[29]



but not in these:

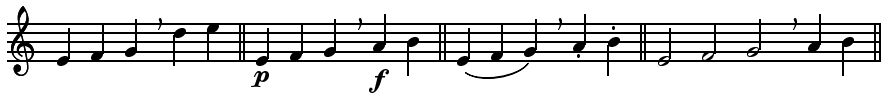


We hurry on presenting GPR 3. It is about change. Consider a sequence of notes $n_1 n_2 n_3 n_4$ (rests can occur inbetween). All else being equal, the transition $n_2 - n_3$ may be heard as a group boundary if:

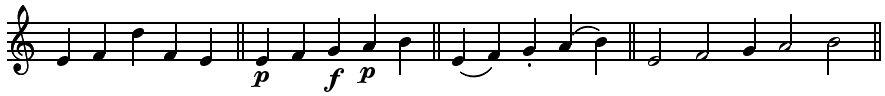
- a. (Register) the transition $n_2 - n_3$ involves a greater intervallic distance than both $n_1 - n_2$ and $n_3 - n_4$, or if
- b. (Dynamics) the transition $n_2 - n_3$ involves a change in dynamics and $n_1 - n_2$ and $n_3 - n_4$ do not, or if
- c. (Articulation) the transition $n_2 - n_3$ involves a change in articulation and $n_1 - n_2$ and $n_3 - n_4$ do not, or if
- d. (Length) n_2 and n_3 are of different lengths and both pairs n_1,n_2 and n_3,n_4 do not differ in length.[LJ83]

The rules apply in these situations:

[29] All examples are from GTTM [LJ83, p. 44f]



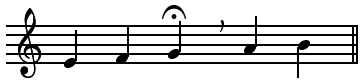
but not in these:



The rules GPR 2a, GPR 2b, and GPR 3d are all talking about change in duration patterns. These have been the most applied in our experiments.

GPR 3a is a view on pitch. This one is important to melodic phrases as for example polyphonic music or a single melody. On the other hand, it is less useful for accompanying voices (not melodically independent), since their contour is of a totally different nature. This gives some unwanted groupings. The rule was designed for monophonic music as the other GPR's, but this one does not seem to generalise to non-monophonic music like we believe the others do. We must be careful when to apply it.

GPR 3b-c are about articulation and dynamics. We have not yet taken that kind of information into account, since rules are much less applicable than the duration based rules. We have however made another rule, which we believe is a natural extension to the rules: Prefer groups ending on fermatas (and groups starting after fermatas):



The way we represent music in graphs includes representation of rests as well as notes. The grouping structure rules evaluate a sequence of four notes and not four MusicVertices (notes *and* rests). It is implicitly given that a phrase cannot start with a rest. Phrases begin with a sound – not with silence. We are not saying that rests are not important in phrasing, they certainly are (GPR 2 is an example of that), but it does not make sense to say that a phrase starts before any sound is produced. In our computational approach we have to take into account that a subgraph can begin or end with a rest. Therefore we have made one more evaluation rule (the rest prohibition rule) that simply punishes a phrase that starts with or ends with a rest. This does not forbid a subgraph to include rests, but our hope is that the subgraphs found constitute natural phrases.

The rest of the GPR's are dealing with grouping of already found groups – how to organise them into larger hierarchical structures. We will return to this topic in section 5.3.1 on page 122.

Grouping structure as sequential viewpoint We are using the GPRs in our GA search for similarities. The idea is that the rules should contribute to the fitness of a similarity statement along the other factors involved. Better grouping bounds gives a better fitness. So we want to evaluate a subgraph according to the grouping rules. We have implemented the rules as a special viewpoint. We call the viewpoint:

- Grouping Value (Grp): A value of how well a subgraph's endpoints (start and end) are locally grouped.

To evaluate a sequential subgraph, we examine the start vertex (source) and the end vertex (sink) of the graph according to the grouping rules. The graph's grouping fitness is the average fitness of the start and the end vertex. The grouping rules are applied to four consecutive vertices. When we want to examine the source, we then must find the two preceding notes and one following. (The case is similar to what we do with the sink. The difference is that for a sink we want to examine just one preceding and two following). We do this by following the `FOLLOW` edges (in the opposite direction in the source case) to a depth of two. Since the mother graph can be non-partwise there can be more than one way of selecting the two preceding notes. To deal with this, we find all possible combinations of two preceding notes (depth two) and the one following (depth one). We then apply the GPR's to each of the sequences of four notes. The grouping value of the source vertex is the average evaluation value of the found combinations.

We use one function to evaluate how suitable a bound between two notes is as a grouping bound. Let n be the number of rules that apply. Then the grouping value is γ^n, where $\gamma \in [0, 1]$. A small γ-value favours satisfying the grouping rules, whereas a large one gives them less importance. We have been experimenting with the γ-value, and we will state the exact value when examples are shown.

So the evaluation of a source (sink) goes like this: if the source (sink) is a rest, no bound is found so the grouping value $\gamma^0 = 1.0$ is returned. If not, we find all sequences of four notes having the source as its third (the sink as its second) note. These are each evaluated according to the four rules mentioned (GPR 2a, GPR 2b, GPR 3d Length and the fermata rule) and each sequence is given the value γ^n (n is the number of applied rules). The value of the source (sink) is the mean of the evaluated sequences.[30]

Figure 24 shows an example of the grouping viewpoint in use on a monophonic fragment. On this melody, we show which grouping rules apply where. The GPR 3d applies a lot here, but also the fermatas have some influence.

Perceptionally, most people will hear this melody as consisting of three phrases, each phrase ending after the fermatas. According to the grouping rules, this is also the case, since there are two rules which apply after each fermata (the appliance of GPR 2b have been calculated with respect to the continuation of the piece). Let $\gamma = 0.6$. The view of a subgraph consisting of the middle phrase (notes 7-12) is therefore: for the source, the value $(0.6)^2 = 0.36$ and for the sink

[30] One remark: If we are not able to find two preceding notes from a given source, it is because the subgraph starts too close to the beginning of the piece or it starts too close to the end of the piece.

In the beginning of the piece there are two cases: either the subgraph starts with a source of the mothergraph. This is seen as an appliance of a grouping rule (the start of the piece is always a grouping boundary) and is rewarded and given the value γ. Otherwise the subgraph starts with a note with distance 1 from the source of the mother graph. This is punished with the value 1.0 since it is isolating the beginning note which is violating GPR 1. We will return to this on page 122.

If on the other hand the source is too close to the end of the piece (it will then be the very last note – otherwise it would have been possible to find one note following it) it is also not preferred, and receives no reward (it is given the value 1.0).

The cases are similar for sinks.

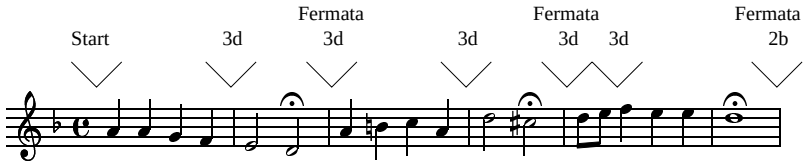


Figure 24: Applying the grouping preference rules.

also $(0.6)^2 = 0.36$. The overall grouping value is then the average: 0.36. If we for example include the sixth note in the melody as well, the grouping value of the subgraph rises to $((0.6)^0 + (0.6)^2)/2.0 = 0.68$.

To sum up, the grouping value is the average of how well the subgraph begins and ends according to the grouping rules. The result is a number in $[0, 1]$.

Experiments have shown that the presence of the preference rules is invaluable in segmenting the graph into musical belonging phrases. See the results from the similarity segmentation algorithm in section 5.5.

4.3.2 View comparators

Applying a viewpoint p to two sequential subgraphs gives us two views v_1 and v_2. A view comparator is an algorithm that compares the views and thereby reduces the two of them to a single number that tells us how similar the passages are when seen from viewpoint p. In Section 4.4 we will introduce view comparators comparing views of non-sequential subgraphs, so we define the first type of view comparator as "Sequential view comparators":

Definition 4.7 *Let $p(s_1) = v_1$ and $p(s_2) = v_2$ be the two views obtained by applying sequential viewpoint p to the sequential subgraphs s_1 and s_2.*
A sequential view comparator is a mathematical function
$c(v_1, v_2)$ of two views v_1 and v_2 that expresses the difference of the underlying sequences s_1 and s_2 when seen from viewpoint p. The result returned by a view comparator is called a view difference.

If the view comparator returns a view difference between 0 and 1 (0 meaning complete equality, and 1 meaning complete difference) we call it a 'percentual' view comparator. View comparators need not be percentual.

It is worth pointing out that using different viewpoints and combining them is rather a flexible method. The obvious viewpoints to use on a MusicGraph include simple pitch and length information, but viewpoints may include any amount of information computed from these basic data (e.g. what harmonic context a given note is in, which requires some analysis to be done). The only real constraints are the programmer's imagination and the increasing computing cost of calculating more and more complex viewpoints. As will be seen in Section 4.4, view comparators that compare non-sequential subgraphs are much more costly than sequential view comparators. We have attempted to keep the view comparator evaluation, and hence the view comparators themselves, as

simple as possible, since they will be used heavily inside the computation of the fitness measure for the GA.

Sequential view comparators used in the SimFinder The job of a sequential view comparator is mainly determined by the viewpoint used in it, but still the comparison of each pair of values must be done, and all comparisons along the two sequences must be combined to yield a single value. How would *you* calculate a view comparison? One could evaluate ratios between the compared pairs of numbers, or sizes of the intervals separating them. E.g. if we have an absolute pitch viewpoint, comparing a c with a d♯ would give a pitch difference of 3. But we would like our view comparators to be independent of viewpoint. The two comparators we have used most for sequential similarities simply focus on the number of *differences* when pair-wise comparing the numbers of the two views:

- *Difference count*: a difference count comparator first counts the number of differences between the two views by comparing numbers at the same indices in the views. It then returns the number of differences. This view comparator, then, does not return a result in [0;1]. In the similarity measures that use this view comparator, we use the number of mismatches returned from the different viewpoints on the two sequences to decide what relation the two subgraphs have. For example, a difference count of the absolute pitch views [49,54,55,50] and [49,56,54,50] will return 2 because $54 \neq 56$ and $55 \neq 54$.
- *Mean difference count*: This type of comparator first counts the number of differences between the two views by comparing numbers at the same indices in the views. It then returns the number of differences divided by the length of the views. Since the resulting view difference is in [0;1], this is a percentual view comparator. As an example, a mean difference count of the absolute pitch views [49,54,55,50] and [49,56,54,50] will return $2/4 = 0.5$.

Combining viewpoints and view comparators A more interesting variation in view comparators begins when they take account of structure. We could define view comparators that check if one view is the reverse of another view. This is the case if

$$view_1 = [v_0, v_1, ...v_n] \text{ and } view_2 = [v_n, v_{n-1}, ..., v_0]$$

If the view comparator is a difference count comparator, it gives a quantitative evaluation of 'reverseness' in terms of how many differences there are between $view_1$ and $reverse(view_2)$. Let's call that a *reversed view comparator*. A reversed view comparator could be combined with any of the viewpoints we have defined in section 4.3.1 to search for reversed motifs.

Another musically useful similarity could be inversion; e.g. in fugue, a motif can occur in an inverted form, such that whenever the original motif moves x diatonic steps upward, the inverse motif moves x diatonic steps downward. An *inverse view comparator*, coupled with e.g. the *diatonic interval* viewpoint, then, is able to locate inverted themes.

The diatonic viewpoints were constructed specifically for similarity searches in Western tonal music. But using inverse and reversed view comparators together with the absolute pitch viewpoint, the SimFinder may as well search for similarities in 12-tone music.

We most often use one of the difference count view comparators; there are enough details to keep track of with the many viewpoints, so when there is no need to mention the view comparator, we shall speak simply of viewpoints and assume that a difference count view comparator is at work.

4.3.3 Similarity measures

The beauty of the concept of a "viewpoint" is its flexibility. A viewpoint is a transformation under which two structures may be equal or not. Two passages may be very similar from one point of view but very dissimilar from another point of view. As discussed in Section 3.2, similarities in music may arise from a number of factors. Since a viewpoint (and its comparator) focuses on a simple factor, we need different viewpoints to locate them. The problem of finding good similarity matches now becomes one of building good combinations of viewpoints so as to pinpoint many different kinds of similarities. This also allows us to match passages that are not exact copies of each other. Exact repetitions do occur in music, but much more frequently, a passage is repeated in a modified form as described in section 3.2 on parallelisms. These are similarities that are often immediately obvious to the human listener but to the software analysis seem buried behind a rigid numerical representation of the music. Our approach is to construct *similarity measures* that express the combined similarity of two subgraphs as seen through a number of (*viewpoint*, *viewcomparator*) pairs.

Definition 4.8 *Let C be an ordered set of sequential view comparators and V be an ordered set of sequential viewpoints.*
A sequential similarity measure is a mathematical function $M_{(C,V)}(s_1, s_2)$ *combining the results* $c_i(p_i(s_1), p_i(s_2))$ *for all pairs* $(c_i, p_i) \in (C, V)$ *to produce a rating of the similarity of two sequential subgraphs* s_1 *and* s_2. *A similarity measure that returns a result in [0;1] is a percentual similarity measure.*

In short, the similarity measure applies the viewpoint p_i to the two subgraphs s_1 and s_2. Then it compares the obtained views using view comparator c_i. The resulting view difference is stored temporarily while this process is repeated for all (viewpoint p_i, view comparator c_i) pairs. A single number is now calculated as a result of all the obtained view differences. This single number expresses the similarity of s_1 and s_2 according to the combined evaluation of all (viewpoint, viewcomparator) pairs of this similarity measure.

Size When used as a fitness function for the GA, a similarity measure should be able to tell a good match from a bad one to allow the GA to search, and also, preferably, to tell how much better the good match is. Also, we want the GA to know that longer matches are better, to a certain extent. Otherwise, it will settle on very small perfect matches, which is boring when we know that there are larger similarities. To be able to compare a match of two subgraphs [31] of, say, size 3, with a match of two subgraphs of, say, size 68, we need to

[31] What we shall introduce below as *Similarity Statements*.

balance two conflicting requirements: on one hand, a good match is a match of two subgraphs whose vertices are very much alike; but on the other hand, the longer a subgraph matching, the better. A perfect match of size 2 is probably less interesting than a reasonably good match of size 12, so we have a tradeoff between quality and size. Although as a matter of forcing the GA to search for larger similarities, this consideration really belongs in the GA, we have chosen to incorporate the size reward in the individual similarity measures, because it allowed us to easily try different size weightings. In the end, we have settled on two different schemes described below.

Sequential similarity measures used in the SimFinder Our sequential similarity measures fall in two categories according to which view comparator they use: Difference Count (DC) or Mean Difference Count (MD). Each similarity measure therefore uses the same view comparator for all its viewpoints. The practical reason for this is that we have constructed a similarity measure evaluation method each. The DC set of measures therefore share the central evaluation method that computes a result from all the view differences. So do the measures in the MD set. Therefore, we may profitably describe the measures under the two headings *seqDC* and *seqMD*.

seqDC Søren's favourite set of measures use difference count and the diatonic viewpoints and the grouping preference rules viewpoint.

- `seqDC_DAPALGrp`: uses the diatonic absolute pitch, absolute length and grouping viewpoints.
- `seqDC_PIALGrp`: pitch interval, absolute length and grouping.
- `seqDC_DIALGrp`: diatonic interval, absolute length and grouping.
- `seqDC_DFIALGrp`: diatonic forward interval, absolute length and grouping.
- `seqDC_DIIALGrp`: diatonic inversion interval, absolute length and grouping.

Although from our measures it would seem that the number of viewpoints used in a similarity measure was restricted to three, there are no such constraints in the SimFinder. We have also experimented with similarity measures containing for example the viewpoints {AP,PI,PC,PS,AL,LI,LC} using an MD comparator. Some of the viewpoints are more useful than others. Distributing pitch-related viewpoints into different similarity measures allows us to reason about the found similarities, when searches alternate between different similarity measures. This is used in the segmentation algorithm described in section 5.4.1. Hence the triple-wise viewpoint use, where a tonal (pitch-related), a rhythmic (note length-related) and the grouping viewpoint are combined. The following description of the DC measures would need to be extended, though, if more viewpoints were added.

The idea of the DC similarity measures is in itself quite simple. We will illustrate it using the `seqDC_DFIAL` measure; the other `seqDC` measures are similar, combining a tonal viewpoint with a rhythmic (note length) and the grouping viewpoint. Let the DFI view difference (DC) of the subgraphs be

$$dfi = DC(DFI(s_1), DFI(s_2)),$$

and let the AL view difference of the subgraphs be

$$len = DC(AL(s_1), AL(s_2))$$

and the Grp view of the subgraphs

$$group = \frac{Grp(source(s_1)) + Grp(sink(s_1))}{2} + \frac{Grp(source(s_2)) + Grp(sink(s_2))}{2}$$

Then

$$\texttt{seqDC\_DFIALGrp}(s_1, s_2) = dfi + \frac{len}{1000} + (1 - \frac{size(s_1)}{100}) + group$$

The idea is that the algorithm first finds a match where dfi is zero. The dfi value dominates all other elements in the expression. Remember that dfi and len here are the number of mismatches in the DFI and AL views of the two subgraphs. If the match can be enlarged and still keep a dfi value of zero, then it can also benefit from a bonus for the subgraph size[32]. As $size(s_1) = size(s_2)$, it doesn't matter which one we pick. The size has to "compete" with the value of the grouping rules – depending on the γ value used to restrict the constraint of the rules. The length information (AL) is given very little effect. This is useful, when the melodic content is the most important (which it often is). See figure 33 on page 89 for an example. See section 4.3.5 for more results.

seqMD Martin's set of measures use mean difference count and some combinations of the absolute-, interval- and contour-viewpoints. No grouping viewpoints are included.

- `seqMD_APAL`: uses the absolute pitch and absolute length viewpoints
- `seqMD_PIAL`: pitch interval and absolute length
- `seqMD_PILI`: pitch interval and length interval
- `seqMD_PCLC`: pitch contour and length contour

In contrast to the `seqDC` measures, the `seqMD` measures treat all view differences equally, i.e. there is no weighting of the view differences built into the `seqMD` measures. The `seqMD` measures are percentual similarity measures, and the calculation method is rather more complicated than that of the `seqDC` measures, so we will go through it in several steps: finding the product of the view differences, scaling the product to [0;1], and modifying the result according to the size of the subgraphs.

First, we calculate a product of the view differences. Let D be the set of view differences obtained by evaluating all (viewpoint, viewcomparator) pairs on the sequential subgraphs s_1 and s_2, and let $d_i \in D$ be the i'th view difference. Then the product of view differences would be

$$prod_1(D) = \prod_{i=0}^{|D|-1} d_i$$

[32] Up to a length of 100 in this definition. For segmentation purposes, we never even reach subgraph sizes of 100.

but this quickly becomes a very small number, since all $d_i \in [0;1]$. Instead we add 1 to every d_i, so every element in the product is $1 + d_i \in [1;2]$. Also, to encourage similarities with one exact match above similarities with a number of mediocre matches (according to the different viewpoints), we want to be able to reward those similarities that have an exact match under at least one viewpoint. We have an exact match under (viewpoint p_i, view comparator c_i) if d_i=0.0. Then we assign a "bonus value" to this element of the product. The bonus value is set between 0 and 1, so that it really improves (lowers) the product value when substituted for one of the product elements that would otherwise have had the value $1 + d_i = 1 + 0.0 = 1$. Formally, the bonus value can be described as a function β:

$$\beta(d_i) = \begin{cases} bonusValue & \text{if } d_i = 0.0 \\ 1 + d_i & \text{else} \end{cases}$$

$bonusValue$ could be set to anything between 0 and 1. We often set it to a value such as $bonusValue = 0.5$ or $bonusValue = 0.9$. The product now becomes:

$$prod_2(D) = \prod_{i=0}^{|D|-1} \beta(d_i)$$

This product is not in [0;1], so we want to scale it to get a percentual similarity measure. The maximum value of $prod_2$ is obtained if all view differences show maximal difference (i.e. have a value of 1.0):

$$max(prod_2) = \prod_{i=0}^{|D|-1} 2 = 2^{|D|},$$

and the minimum is obtained if all view differences show exact matches (i.e. have value of 0.0, thus being substituted by the bonus value):

$$min(prod_2) = \prod_{i=0}^{|D|-1} bonusValue = bonusValue^{|D|}$$

So we have that $min(prod_2) \leq prod_2 \leq max(prod_2)$. To scale $prod_2$ to [0:1] we apply the following to $prod_2$:

$$scaledProduct_1 = \frac{prod_2 - min(prod_2)}{max(prod_2) - min(prod_2)}$$

What now remains to be done is some scaling for the size of the subgraphs, matches of long sequences being better than matches of short sequences. Here we run into a problem with $scaledProduct_1$ because it may assume the value 0.0. If the scaled product is 0, it doesn't matter how much we scale it for size; the net effect is that a perfect match of size 10 will not be evaluated as better than a perfect match of size 3. The size modification is cancelled for perfect matches. To overcome this, we add a very small number, μ, to both the numerator and the denominator in the fraction of $scaledProduct_1$:

$$scaledProduct_2 = \frac{prod_2 - min(prod_2) + \mu}{max(prod_2) - min(prod_2) + \mu}$$

This way, there will always be a small μ to be scaled by the size modifier, so that *size matters*!

Let n be the size of the sequential subgraphs. The simplest way to scale a value for size is simply to divide the value by n, but we have more requirements for the scalar. The size modifier is designed to be a number between 0 and 1 to be multiplied with the value given by $scaledProduct_2$. The idea is to scale down even further $scaledProduct_2$, if the similarity is a long one, but only if it is also a good match. The $sizeMod$ scalar should therefore be 1 for the smallest possible match ($n = 1$) and come closer to 0 as n grows; but it should not come that much closer to 0 if the match isn't good; a very large awful match is not much better than a small awful match.

Imagine the size scalar begins with a value of 1. Our idea is to control how much of the scalar can be chopped off by size considerations. As shown in figure 25, a fixed amount $0 \leq valWeight \leq 1$ determines how much weight the unmodified value (in our case the $scaledProduct_2$ result) should have, and the rest of the scalar, $sizeWeight = 1 - valWeight$, may now be decided by the size.

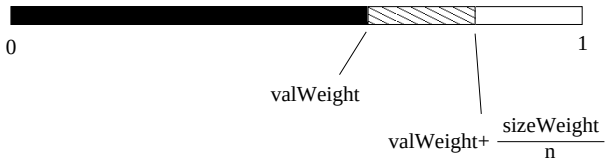


Figure 25: Constructing a size modification scalar that can only shrink down to '$valWeight$'.

$$sizeMod_1 = valWeight + \frac{sizeWeight}{n} = valWeight + \frac{1 - valWeight}{n}$$

Figure 26 shows a 3D plot of $sizeMod_1$ as a function of n and $valWeight$. When $valWeight$ is close to 1, the modifier is almost one, no matter the size n. When $valWeight$ is close to 0, n has a big role to play and reduces the $sizeMod_1$ scalar almost to 0.

There is one last thing to fix before the size modifier can be put to use. We find that when varying n from 1 and up, $sizeMod_1$ decreases too quickly. We would like to flatten the curve so that the benefit of increasing in size is a little more equally distributed over size. We introduce the flattening constant φ to do this:

$$sizeMod_2 = valWeight + \frac{(1 - valWeight)\varphi}{n + \varphi - 1}$$

For $\varphi = 1$, $sizeMod_2 = sizeMod_1$, so 1 is the neutral value. When $\varphi > 1$, the curve is flattened, and when $0 < \varphi < 1$, the opposite happens. This is illustrated in figure 27 where $sizeMod_2$ is shown for $\varphi = 40$ and for $\varphi = 0.5$. We have chosen $\varphi = 40$ for the `seqMD` similarity measures.

We need to decide what value to use for $valWeight$. But here is exactly the connection that we wanted between the size modifier and and the scaled product: if we set $valWeight = scaledProduct_2$, the size modifier has almost

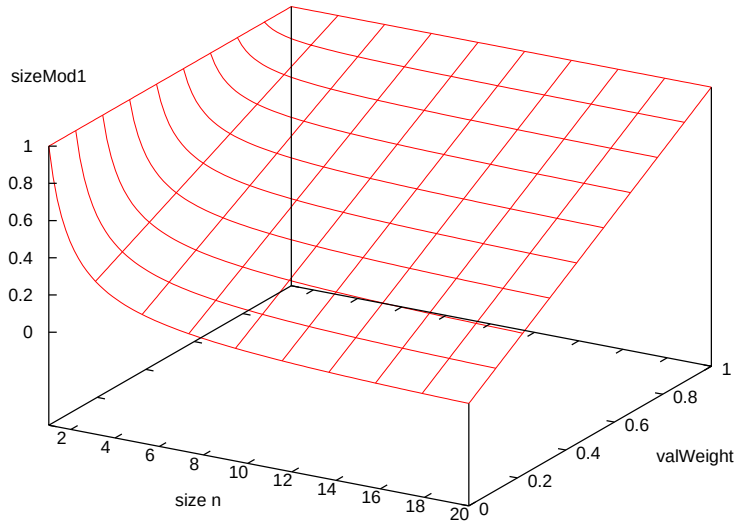


Figure 26: The $sizeMod_1$ size modifier as a function of the size n, shown on the x-axis with values from 1 to 20, and of $valWeight$ shown on the y-axis with values from 0 to 1.

no influence, when the scaled product is close to 1 (when we have a bad match); and when the scaled product is close to 0 (when we have a good match), the size modifier has a lot of influence, scaling the value even further down towards 0. For convenience, we rewrite the size modifier as a function of the number $valWeight$, which we shall call simply $weight$:

$$sizeMod_3(weight) = weight + \frac{(1 - weight)\varphi}{n + \varphi - 1}$$

A few words may be appropriate on the interpretation of diagrams like figure 27. It illustrates the basic dilemma in a GA with individuals of mixed size. The population consists of pairwise subgraph comparisons, and though the two subgraphs in a pair must have the same size, pairs may differ in size. To decide if comparison pair 1 is better than comparison pair 2, the fitness function must balance two search goals: match quality and match size. We both want the best matches and the largest matches. φ decides how quickly the size modifier becomes very low, when the match size n rises (low φ=very fast, highφ=very slowly). For the limited match sizes that have occurred in our graphs ($n < 50$), a high flattening constant like $\varphi = 40$ also determines that the size modifier overall level doesn't come close to 0. The higher values of φ seem to be a good setting, though it is difficult to detect any real difference in performance (we have tested different values of φ, see appendix B.1). For all settings of φ the size modifier is 1 whenever either $n = 1$ or the value weight $weight = 1$, and for all other values of n and $weight$, $sizeMod_3$ will have a lower value if either

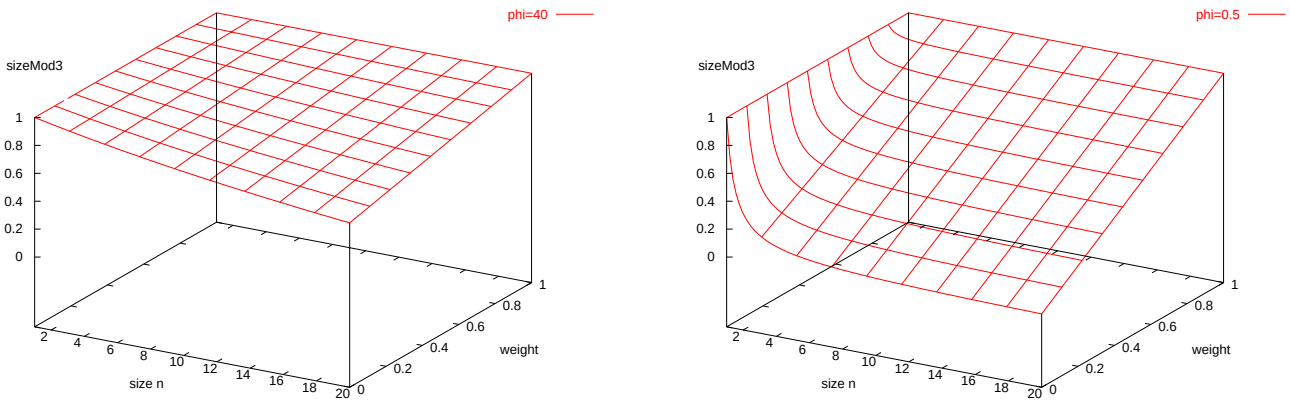


Figure 27: The effect of the flattening constant, φ, on the $sizeMod_2$ size modifier. n is shown on the x-axis, and the $weight$, which is actually the $scaledProduct_2$ is shown on the y-axis. The left diagram shows $sizeMod_2$ for $\varphi = 40$, the right for $\varphi = 0.5$

n is increased or $weight$ is decreased. Thus all settings of φ will drive the GA search towards larger matches and better matches.

To sum up on the construction of the `seqMD` similarity measures, they are constituted of a product of view differences, which is scaled to [0;1] and modified for size. Recall that D is the set of view differences obtained by applying all (viewpoint,view comparator) pairs of `seqMD` to the sequential subgraphs s_1 and s_2. Then

$$\texttt{seqMD}(D) = scaledProduct_2 \times sizeMod_3(scaledProduct_2)$$

where

$$scaledProduct_2 = \frac{prod_2 - min(prod_2) + \mu}{max(prod_2) - min(prod_2) + \mu} = \frac{\prod_{i=0}^{|D|-1} \beta(d_i) - bonusValue^{|D|} + \mu}{2^{|D|} - bonusValue^{|D|} + \mu}$$

and

$$\beta(d_i) = \begin{cases} bonusValue & \text{if } d_i = 0.0 \\ 1 + d_i & \text{else} \end{cases}$$

and

$$sizeMod_3(weight) = weight + \frac{(1 - weight)\varphi}{n + \varphi - 1}$$

As explained in appendix B.2, we have found no clear indication of an optimal setting for the bonusValue parameter. Test runs seem to work nicely with $bonusValue = 0.9$.

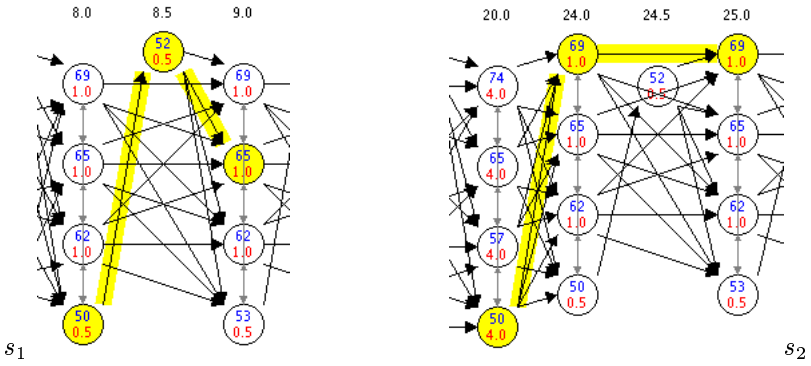
An example set of parameter values we have used is:

$$bonusValue = 0.9,\ \varphi = 40,\ \text{and}\ \mu = 10^{-7}$$

It has turned out that it is fruitful to weight the pitch viewpoints a little heavier than the note length viewpoint. In our implementation, weighting is

not incorporated in the seqMD measures as such, but the weighting can be accomplished by adding more instances of the pitch related than of the length related viewpoints. E.g. the seqMD_APAL measure therefore has two identical 'absolute pitch' viewpoints and one 'absolute length' viewpoint.

Example of the use of the seqMD measures The two subgraphs in figure 28 are compared using seqMD_APAL – i.e. with the absolute pitch and absolute length viewpoints. The evaluation output of the SimFinder is shown below the graphs.




```
SeqSimStmt of
        Size = 3
        Overlap=0
seqMD_APAL = 0.524622588884182
        absPitchMD = 0.6666666666666666
        absPitchMD = 0.6666666666666666
        absLengthMD = 0.6666666666666666
```

Figure 28: Two sequential subgraphs s_1 and s_2 and the SimFinder's evaluation output using seqMD_APAL

The mean difference view comparisons on absolute pitch and absolute length both give a value of 0.67, so with the double instance of the absolute viewpoint, we have three 0.67 values in the product:

$$scaledProduct_2 = \frac{\prod_{i=0}^{|D|-1} \beta(d_i) - bonusValue^{|D|} + \mu}{2^{|D|} - bonusValue^{|D|} + \mu} = \frac{\prod_{i=0}^{2} \beta(0.67) - bonusValue^{3} + \mu}{2^{3} - bonusValue^{3} + \mu}$$

$$= \frac{\prod_{i=0}^{2} 1.67 - 0.9^3 + \mu}{2^3 - 0.9^3 + \mu} = \frac{4.63 - 0.73 + 10^{-7}}{8 - 0.73 + 10^{-7}} = 0.54$$

Applying the size modifier yields:

$$\texttt{seqMD\_APAL} = scaledProduct_2 \times sizeMod_3(scaledProduct_2)$$

$$= 0.54 \times (0.54 + \frac{(1 - 0.54) \times 40}{3 + 40 - 1}) = 0.54 \times (0.54 + \frac{18.5}{42}) = 0.54 \times 0.98 = 0.52$$

This is a mediocre value, reflecting the fact that s_1 and s_2 are equal only in one third of the notes (both with respect to pitch and note length).

4.3.4 Genetic algorithm

We have now come to the heart of the SimFinder. A GA is a non-deterministic algorithm that is useful for searching immense search spaces, and also very spiky search spaces, where optima are not necessarily surrounded by many points of similar value, leading up to – and thus pointing the search algorithm to – the optimum. As we have attempted to point out (appendix B.8), the search space of the SimFinder is indeed rather spiky, and the search space of possible subgraphs in a music graph is quite overwhelming.

Similarity statements Once a similarity measure for sequential subgraphs is established, we can begin using a search method such as a genetic algorithm (GA) to find similarities in the music graph. The population in our GA will consist of similarity statements that basically are tentative assertions of the form:

"According to similarity measure M_1, subgraph s_1 is similar to subgraph s_2."

As such, a similarity statement can be assigned a value – a fitness value – depending on how similar s_1 and s_2 really are; and how similar they really are is defined by the similarity measure M_1 used. Through the generations, the GA will throw away bad similarity statements and keep the good similarity statements; the latter will survive from generation to generation, cross-breeding new similarity statements that receive a copy of a subgraph from each parent, and mutating now and then to point to slightly different subgraphs. This is how the GA searches for the best possible matches of sequential subgraphs.

We have not experimented with co-evolution of similarity statements and similarity measures, but we do give it a thought in section 6.1.8. It is important in our implementation that the similarity measure be the same for all fitness evaluations. The size of a sequential similarity statement is the size of each of its sequential subgraphs, which are required to have the same size for the sequential viewpoints, view comparators and similarity measures to work.

Figures 29 and 30 show the two sequential subgraphs of the best similarity statement in the population of a small SimFinder test run. Population size was 100 and the SimFinder ran for 30 generations, taking about five seconds to complete. Note that s_2 is identical to s_1 except that it is transposed two semitones upwards.

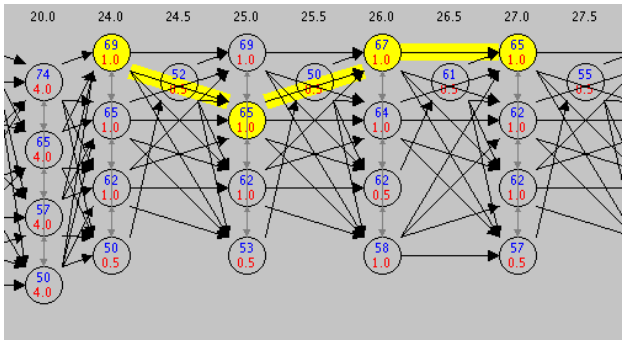


Figure 29: Sequential subgraph s_1

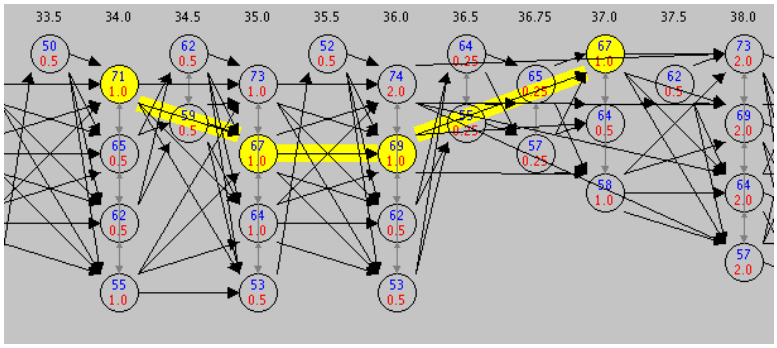


Figure 30: Sequential subgraph s_2

Overlap Recall that the overlap of two subgraphs is the total number of vertices *and* edges of the mother graph that the subgraphs have en common. Evolutionary algorithms are masters in finding stupid mistakes in one's definition of the fitness function, and if overlap of the subgraphs s_1 and s_2 is not taken into account, the GA very quickly settles on a wonderful match of two subgraphs that, upon closer inspection, turn out to be exactly the same spot in the mothergraph. Fortunately we don't need a GA to tell us that a subgraph is equal to itself. To overcome this, we have experimented with overlap punishments that are a multiplication of the general result by a base number $k > 1$ that is raised to the power of the size of the overlap. We have also tried adding some constant value to the result when overlap is detected. Finally we have decided on a very simple method. We don't need the additional weighting introduced by elaborate overlap countermeasures; we simply don't want subgraphs to overlap, so we set the fitness of a similarity statement with $overlap > 0$ to an astronomical number. This way the overlappers are discarded through selection[33]

Selection, crossover, and mutation As mentioned, selection uses the chosen similarity measure as a fitness function. As similarity measures incorporate rewards for size, the SimFinder is able to operate with a mixed population of similarity statements of different sizes.

The SimFinder uses a steady state selection to create the next generation. The next generation is composed of three parts: a given percentage is chosen through selection (given in the *selection* parameter), another percentage is created by crossover of two individuals of the selected individuals (given in the *crossover* parameter, and the remaining percentage of the next generation is created through mutation of the selected or crossbred individuals (given in the *mutation* parameter). The percentages must sum to one, otherwise we would change the population size from generation to generation. Finally, after the new population is created, a number of random mutations, also specified by the *mutations* parameter, are applied to the new generation. The SimFinder always

[33]The ultimate reason for simply discarding overlapping similarity statements is that the segmentation algorithm presented in section 5.4.1 cannot substitute compound vertices for two subgraphs that overlap. It does not seem to hinder the GA search, compared to the more elaborate weighting schemes.

lets the most fit similarity statement in each generation survive to the next. In GA parlance, this is an example of *elitism*. If it is turned off, the SimFinder takes longer to find good matches.

$$selection + crossover + mutation = 1$$

Examples of typical parameter setup could be:
$selection = 0.5, crossover = 0.1, mutation = 0.4$, for a `seqDC` run, or
$selection = 0.5, crossover = 0.0, mutation = 0.5$, for a `seqMD` run.

There is a reason why we have set the crossover parameter so low. The crossover operation takes two similarity statements and combines them by picking one sequential subgraph from each. The two parent statements most often have different sizes, so the smallest of them is extended at random until the sizes match. However, the chances that this will yield a good similarity statement are very low, because often the two subgraphs that are chosen are very different. Instead, the main search of SimFinder takes place in the mutation operation, which is also unconditionally applied to statements that result from the crossover before they are released into the new population. Experiments have shown that *crossover* values of 0 or 0.1 are better for finding large matches quickly (see appendix B.3).

Mutation on a sequential similarity statement with sequential subgraphs s_1 and s_2 can take several forms. The different mutation operations add and remove edges and vertices from the subgraphs. Notice that no new vertices or edges are created; in our implementation, it is the structure in the underlying mother graph that is pointed to by a subgraph, and mutations on subgraphs are only manipulations on the membership in the subgraph of the edges and vertices of the mother graph. Common to all is that they must preserve the constraints that make s_1 and s_2 sequential.

- **Substitution** Substitute either s_1 or s_2 with a new and randomly generated subgraph of the same size. The SimFinder has a parameter that controls the probability of applying this mutation instead of the other three mutations when the mutation operation is invoked. The parameter is called *fresh blood chance*.

- **Extension** Extend both s_1 and s_2 once. Extension of a sequential subgraph can be either a left extension or a right extension, chosen at random. A left extension adds a vertex v from which there is a `FOLLOW` edge e to the source of the sequential subgraph in the mother graph; a right extension adds a vertex v to which there is a `FOLLOW` edge e from the sink of the subgraph in the mother graph. In both cases e is also added to the sequential subgraph, and the size of the subgraph is increased by 1.

- **Shortening** Shorten both s_1 and s_2 once. As with extension, a subgraph can be shortened either to the left or to the right, which is chosen at random. A shorten-left operation removes the source and the edge that connected it to the rest of the sequential subgraph, a shorten-right operation removes the sink and its edge. The size of the subgraphs is decreased by 1.

- **Slide** Slide both subgraphs once to the left or to the right (s_1 and s_2 need not slide the same way). A slide-left operation is equivalent to doing an extend-left and a shorten-right operation, and conversely, the slide-right operation is equal to an extend-right and a shorten-left operation. The size of the subgraphs is not altered by a slide. As an example of a slide operation, figure 31 shows the two subgraphs of a similarity statement before and after the slide.

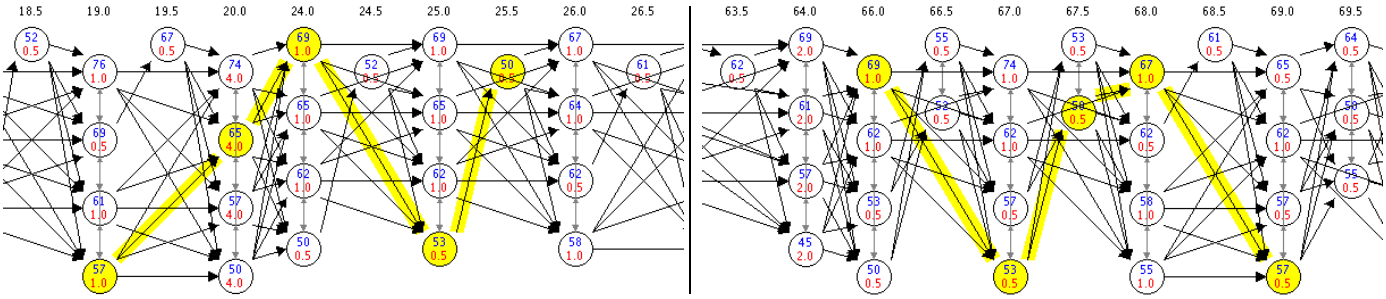
- Another mutation that would make sense for sequential subgraphs could be to make a number of '**simultaneous swap**' operations on either s_1 or s_2. A simultaneous swap exchanges a vertex v_1 in the subgraph with another vertex v_2 in its group, removes the edges connecting v_1 to the rest of the subgraph and adds edges from v_2 to the vertices that v_1 was connected to. This can only be done, of course, if the appropriate edges exist in the mother graph. The simultaneous swap has not been implemented in the SimFinder.

If we have found a good match, chances are that some more of the surroundings will also match. The mutation operations allow the GA to extend the best found matches to surrounding areas in the graph. Experiments varying the number n of mutation operations per mutation have shown that few are better than many.

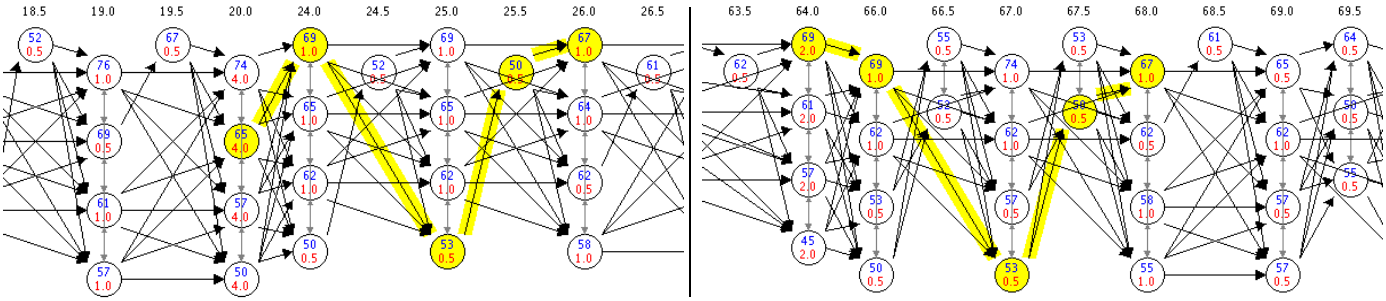
Mutation and vertex usage Figure 32 shows the *vertex usage* at the end of a SimFinder run. The midi file used is the Bach chorale Jesu Meine Freude (BWV358); it was loaded into a non-partwise graph. Each music vertex in the graph is represented by a small square, ordered in time from left to right. Vertically aligned squares are simultaneous and thus form a group. The colour of each square indicates how many subgraphs, in the final population of a SimFinder run, the vertex is included in. The lighter the colour, the more used. The vertex usages were obtained with the following SimFinder setting:

Population size	100
Generations	300
Crossover	0.0
Mutation	variable
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`
μ	0.000001
φ	40
Bonus value	0.9

When the mutation parameter is low, the SimFinder concentrates very narrowly on the first larger similarities it finds and explores them. The first two thirds of the chorale is constituted of two exactly identical copies of the theme. Therefore this is the easiest region to find similarities in, which is why almost no vertices from the last third are used for the low (0.1, 0.2, 0.3) mutation values. For mutation=0.1, it looks like the SimFinder has located one similarity that it hasn't been able to break out of and search for other similarities: there is only



```
SeqSimStmt of
        Size = 5
        Overlap=0
seqMD_APAL = 0.7643451026479342
        absPitchMBD = 1.0
        absPitchMBD = 1.0
        absLengthMBD = 0.6
```



```
SeqSimStmt of
        Size = 5
        Overlap=0
seqMD_APAL = 0.12662088279314987
        absPitchMBD = 0.2
        absPitchMBD = 0.2
        absLengthMBD = 0.2
```

Figure 31: The subgraphs of a sequential similarity statement before and after the slide operation, the evaluation according to similarity measure `seqMD_APAL` is also shown before and after the slide. Subgraph 1 is slid to the right, and subgraph 2 is slid to the left. This slide is a lucky one; it changes the fitness evaluation of the similarity statement from 0.76 to 0.13.

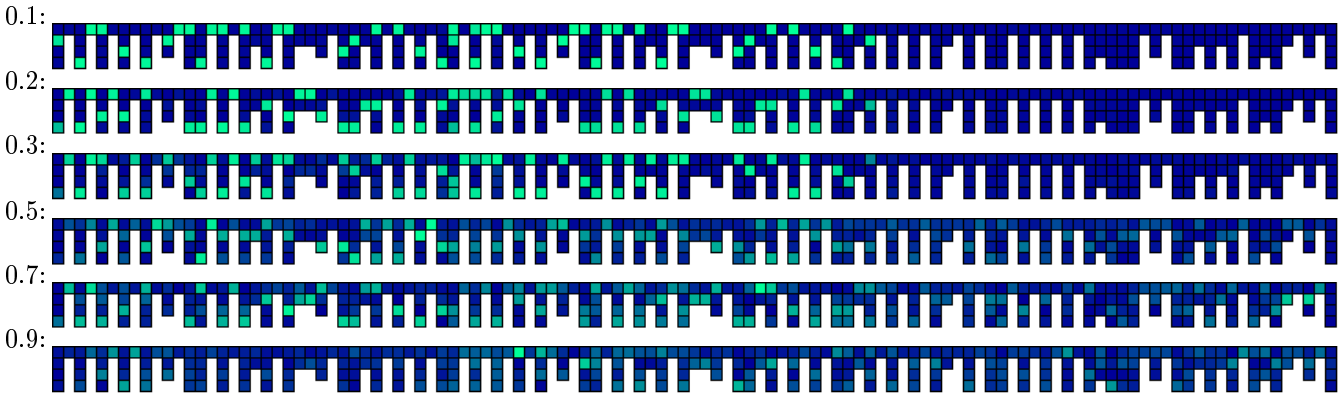


Figure 32: Vertex usage for different values of the *mutation* parameter. Colour values range from dark blue (not used at all) to light green (has largest usage for this population). The colours may be difficult to see on print, but the differences are much clearer when the paper is held up against a bright light source.

one very bright green (i.e. used by almost all subgraphs) string of vertices used – the others are dark blue and therefore unused. When mutation is raised, the final population uses more and more of the other vertices and also manages to explore to some extent the last third of the chorale. The SimFinder runs that produced the vertex usage graphics ran for 300 generations; the development of the size and the fitness of the best individual over the 300 generations can be seen in appendix B.7.

It seems that small values (0.0 to 0.2) of the mutation parameter are worse than higher values, as explained in appendix B.4. But we want to balance the SimFinder so that it is able to both *exploit* found similarities by enlarging them if possible, and *explore* other areas, like the third part of the chorale above in figure 32. This will be particularly important in the segmentation algorithm, when we use the SimFinder to search for all occurrences of a given phrase. The segmentation algorithm stops searching for more occurrences, if the SimFinder doesn't find any matches that are good enough, so it is vital that it doesn't get stuck in a mediocre match if there is a good one to be found. Otherwise the segmentation algorithm will fail. We have mostly used mutation values of 0.3 to 0.5, as it seems to be an acceptable tradeoff.

The most drastic method of mutating is controlled by the *fresh blood chance* parameter. This is the probability that, when mutating a similarity statement, instead of extending, shortening or sliding its subgraphs, one of the subgraphs is replaced by a completely random subgraph of the same size. It looks like a value of 0.2 to 0.3 is best for this, see appendix B.5 for the details on the test results.

Yet another variable concerning mutation is the number of random mutations applied in a mutate operation. Increasing this number over 1 worsens the exploitation ability of the GA, so we only do one mutation per call to the mutate operator (see appendix B.6).

The fitness landscape To create an idea of what the fitness landscape looks like, we have tried sliding a sequential subgraph s_1 at random through an entire

piece and at each moment comparing it with another sequential subgraph s_2. s_2 was fixed, but initially randomly chosen inside the non-partwise mothergraph of the piece. The changing values of the overall similarity measure `seqMD_APAL` is shown in appendix B.8. The piece used was the Bach chorale BWV358 "Jesu, meine Freude" and the length of the subgraphs was 4 (i.e. 4 vertices, or notes). The figures should give an idea of how the similarity measure value may change as a result of mutations on its subgraphs. As the 'slide' operation example in figure 31 showed, the value of a similarity statement can change quite drastically as a result of only two mutations (one slide per subgraph). The plots in appendix B.8 confirm that the fitness landscape is indeed rather spiky.

4.3.5 Results

We here present some small examples of what we can find in a sequential search. This is not a systematical test, but just some small examples of what the viewpoints are designed to do, and how they are correlated. In none of the cases, grouping rules have been calculated. We will present some examples with grouping rules applied in the section on similarity segmentation (see section 5.5).

Results from the seqDC_DFIAL similarity measure as described in section 4.3.3.

We have made some example runs on a J. S. Bach Inventio no. 13 in *A* minor (BWV 784). The inventios are for 2 parts and the search have been made in a partwise graph. The inventios contains a lot of parallelisms as the name reveals.

Figure 33 shows a match of size 13 in the Bach Inventio. The first example is subgraph 1 (starting in the middle of measure 10) and the next is subgraph 2 (starting in measure 11). In the search, only the DFI and AbsoluteLength viewpoints are used, but the values from the stronger view comparators (of type difference count) have been calculated as well: `seqDC_DAP`: 13, `seqDC_PI`: 7, `seqDI_PI`: 1, and `seqDC_DFI`: 0. No grouping structure rules were used in this search (no additional grouping value was added to the fitness). The values from the other viewpoints shows that the DFI is the only viewpoint that can recognise this parallelism as a perfect match. The first 4 notes in each sequence have been

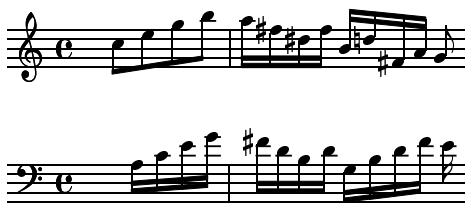


Figure 33: Example of perfect DFI match.

matched together even though they have different rhythmic values. This is no mistake. The difference in durations was not given great importance in this search. We hope most people will agree that it is the melodic similarity that

creates the illusion of parallelism – the melodic similarity is more significant than the rhythmic difference. The difference in durations is: `seqDC_AL`: 5.

Figure 34 shows a match of size 22 from the same Inventio. Here the view comparators gave the following values: `seqDC_DAP`: 22, `seqDC_PI`: 9, `seqDC_DI`: 1, `seqDC_DFI`: 0, `seqDC_AL`: 0. What you hear here is again a harmonic parallelism.

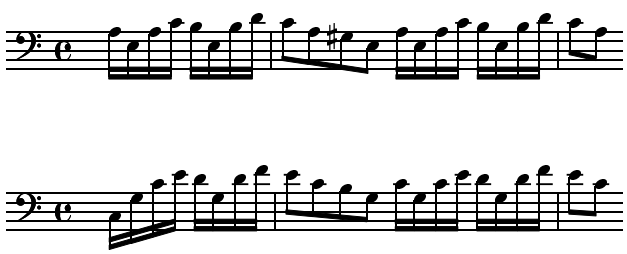
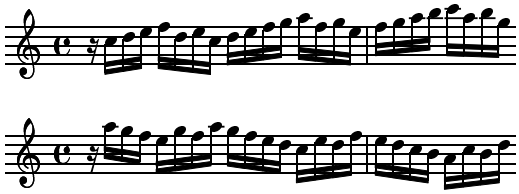


Figure 34: Example of perfect DFI match. Theme in a-minor and theme in c-major.

The first piece is the main theme in A minor (beginning in measure 1), and the second is the theme in C major (beginning in measure 6). The theme have been transposed to its relative key.

Results from the seqDC_DIIAL similarity measure We here show the largest *inversion* match found in the J. S. Bach Invention 1 in C major (BWV 772). It has a length of 23 notes.



The first seven notes of the first passage shown origins from the main theme. The next 2×8 notes continue this idea in a small sequence in thirds. The second figure shows the diatonic inversion. All jump-intervals have been inverted.

4.4 Non-sequential similarities

The search for sequential similarities doesn't give us a way to compare e.g. chords or other non-monophonic material. Of course we may search for sequential similarities in a non-partwise graph, thus including strings of notes going across parts, but it would still require a comparison of *parallel combinations* of sequential subgraphs to compare non-monophonic fragments.

At first, the task to compare two non-sequential subgraphs like σ_1 and σ_2 in figure 35 could seem at bit like finding the similarity between a cucumber and an elephant. Our task in this section is to find properties of non-sequential

subgraphs that allow us to measure and give a numerical rating of their similarity. We wish to exploit our graph representation to construct a flexible and topologically aware comparison instead of comparing versions of the subgraphs that are reduced to nested sequential/parallel structures as discussed in section 3.1.3. The following three sections describe different approximations to subgraph matching.

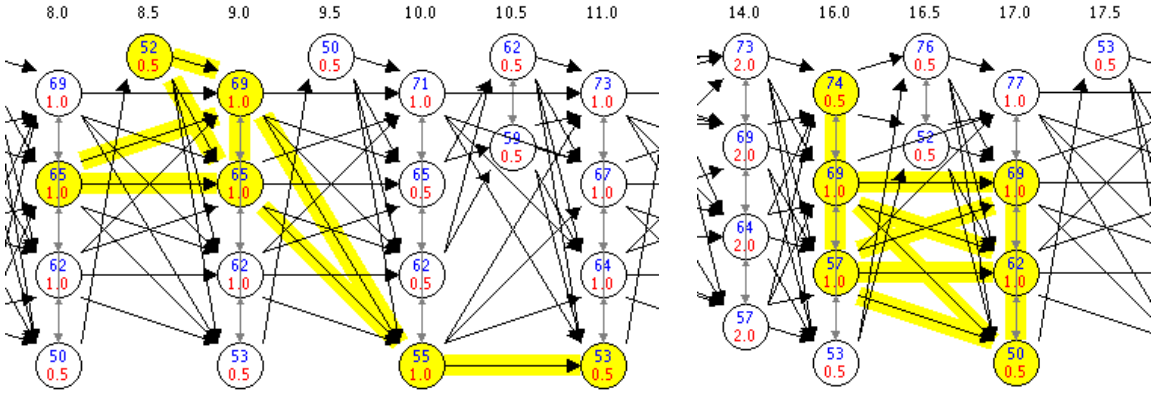


Figure 35: Comparing two non-sequential subgraphs σ_1 and σ_2.

The music-specific nature of our graphs is another part of the challenge here; it is not sufficient to compare subgraphs exclusively on their topology. The topology of a music graph reflects only temporal relations[34], but the music graph is an attributed graph with note information too. We therefore have to develop musically relevant approximations to subgraph matching.

4.4.1 Introducing non-sequentiality in the SimFinder

We use the framework we have already defined for looking at musical structures from different viewpoints using the SimFinder. But some of the non-sequential viewpoints take on different forms[35]. This means that it is difficult to put down an overall definition of what a non-sequential viewpoint, view comparator or similarity measure is. However the basic idea remains the same: a viewpoint filters a particular aspect of a musical fragment, and view comparators give us a measure of how different two fragments are with respect to a given viewpoint. A non-sequential similarity measure, then, is the overall weighting and combination mechanism that produces a numerical rating of similarity based on one or more viewpoint-view comparator pairs. Like for the sequential measures, overlapping similarity statements are punished promptly by setting their fitness to a very high value. Size modification is also incorporated into each similarity measure, as the methods used differ.

Mutation The mutation operator for sequential subgraphs was constructed specifically to ensure that sequentiality is preserved through extension, shortening or slide. When mutating a non-sequential subgraph, there are no such

[34] We are tempted to say: the topology reflects only temporal relations *yet*, since it is possible to include additional musical information as described in section 4.2.2.

[35] E.g. in the vertex comparison and edge comparison methods, a viewpoint gives information on only a single vertex, or edge respectively.

restrictions. Extension is done by picking a vertex at random inside the subgraph until we find one, which has an edge *in the mothergraph* to a vertex v outside the subgraph. Then v and all edges connecting it to the subgraph are added. Shortening consists in finding a vertex that we may remove without splitting the subgraph. This is checked using the transitive closure of the subgraph if we consider it a non-directed graph. The found vertex is then removed along with all edges connecting it to the rest of the subgraph. We have only implemented a very simple sort of slide operation, which consists in performing first an *extend* operation and then a *shorten* operation. This could no doubt be improved.

An interesting feature of this small and very simple set of non-sequential mutations is, that subgraphs evolve towards 'dense' subgraphs. That is, we don't often find non-sequential subgraphs that cover the mothergraph thinly, in the sense that they stretch out long 'arms' without including all vertices connected to these arms. Rather, if a vertex lies as an island in the middle of a subgraph while not being included in the subgraph, the probability that it will be chosen for inclusion is higher than the probability for vertices that are connected to the outer edges of the subgraph (in the mothergraph). The reason, we think, is that 'island' vertices are connected to the subgraph by more edges than vertices on the outskirts of the subgraph are. The 'dense' tendency avoids completely strange configurations of subgraphs that would perhaps be in greater danger of being meaningless than the dense ones. This is worth more experimentation, though; the non-sequential GA search needs improvements that balance its exploration and exploitation abilities better, and introducing non-sequential mutation operators that are 'musically aware', i.e. make more meaningful mutations than the random *extend* or *shorten* operations, could perhaps be a good idea.

Grouping The viewpoint for grouping structure is easily extended. Since subgraphs now can be non-sequential, there can be more than one source, or starting point, in a subgraph. Instead we have an entire set of sources, and a set of sinks. To evaluate if a non-sequential subgraph has a stable grouping, we evaluate all sources and sinks (as already described in section 4.3.1) and then take the average value. The result is again a number in $[0, 1]$.

We would like to show two runs of the SimFinder: the first with grouping preference rules applied, and the second without. Figure 36 shows one of the subgraphs from the best similarity statement in each run. The runs found similarities in the same area, so they can easily be compared. The subgraphs found with the grouping rules applied has size 21, and the other ended with a graph of size 20. The parameters of the experiment was these: Generations = 100, Popsize = 80, init size = 5, crossover = 0.0, mutation = 0.5 (fresh-blood-chance = 0.4), the similarity measures used was `nonSeqBagIR_DAP_Grp` (with grouping base value $\gamma = 0.97$) and `nonSeqBagIR_DAP` (with no grouping). The similarity measures will be described in section 4.4.4, so please be patient – we will only discuss the effect of the grouping rules here.

The SimFinder found in both cases perfect matches of the subgraph (the first six bars (24 beats) are repeated unchanged). The natural grouping of this piece of music would be to take a breath after the fermatas. So a phrase ends before tick 16.0 and a new phrase runs from tick 16.0 to after 23.0. Notice the

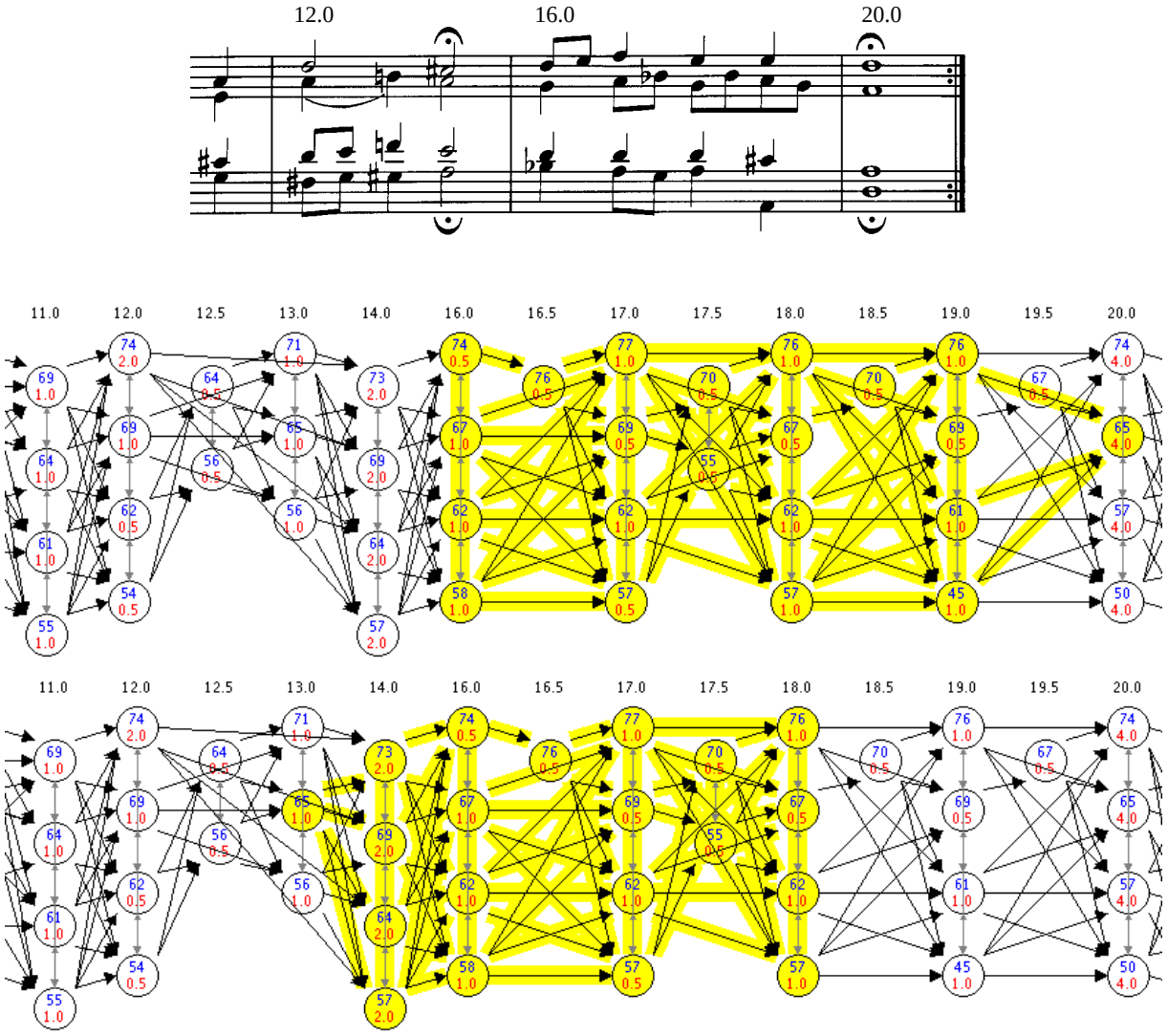


Figure 36: Running the SimFinder with and without the grouping preference rules.

difference in the extension of the graphs. The grouped graph starts rigth on the phrase, and extends until the end of it. A natural continuation of that search would be to include more of the notes starting on tick 20.0 since they will give it a better grouping value than the notes on tick 19.0.

The un-grouped graph on the other hand starts and ends in the middle of two phrases. It could expand in both directions as long as there are similarities to be found, since only the size of the graph can make the fitness better (as long as the match stays perfect).

The four grouping rules we have implemented, and the way we use them on non-sequential graphs does not guarantee a perfectly well formed grouping of the piece (that it finds the smallest pieces first), but it certainly has a remarkable

impact. The search for well bounded passages in the music is very important in analysing and understanding the music in terms of its smaller components. So when trying to segment, it is extremely important not to make phrases which contains notes belonging to other phrases. The grouping structure rules help us to do this correct. We will return to this issue in the section about the similarity segmenter (section 5.4) and in the section describing the results from the algorithm (section 5.5).

4.4.2 Comparison of vertices

We now turn to the first non-sequential similarity measure. This method tries to match the vertices of σ_1 to those of σ_2 in order to determine how similar they are. The algorithm can be sketched like this:

Algorithm NonSeqVertexCompare

1. Compare all vertices of σ_1 to all vertices of σ_2, giving a $size(\sigma_1) \times size(\sigma_2)$ matrix of vertex comparisons.
2. Using the resulting matrix of comparisons, decide which vertices from σ_1 match up with which vertices from σ_2, giving an optimal vertex matching.
3. Compute a final similarity measure of σ_1 and σ_2 based on the individual vertex comparison values in the best matching, and on how many multiple matchings the best matching contains.

Comparison of two vertices can be done in different ways according to which attributes we choose to focus on. It is natural to pick some viewpoints that we have already used for sequential similarities, e.g. the note pitch of each vertex, and the note length. But we also need some topological information to compare the structure of two subgraphs. The viewpoints we use here on each vertex v are:

- The number of in-edges of type `FOLLOW` of v
- The number of out-edges of type `FOLLOW` of v
- The number of in-edges of type `SIMULTANEOUS` of v (which is the same as the number of `SIMULTANEOUS` out-edges, since both ways are represented in the graph)

The vertex comparison thus both incorporates note relevant information and graph topological information.

When constructing a new vertex comparison-based non-sequential similarity measure μ in the SimFinder, you add a number of view comparators to μ's list of comparators, and supply μ with a way to combine the view differences resulting from each view comparator into an overall measure of similarity (just like for sequential similarity measures). Step 1 of Algorithm NonSeqVertexCompare above is accomplished in each view comparator using its associated viewpoint, and steps 2 and 3 are taken care of by μ in the algorithm supplied to it. We have written a standard algorithm that can be used when constructing new vertex-based non-sequential similarity measures. It tries to minimise the number of

multiple matchings while letting each vertex be matched to its favorite partner in the other subgraph. If several vertices in the other subgraph match up equally well with a vertex v, then v has several favorites. If many of the vertices have several favorites, the combinatorial explosion makes the search for a best match substantially slower. At subgraph sizes of about 30 and higher, the search becomes painfully slow, but it runs on, and finds some beautiful examples of matches in the graphs.

Please note that when we do single vertex-vertex comparisons, the delta-viewpoints that look at changes from vertex to vertex (e.g. PitchInterval) cannot be implemented; a single vertex does not have a PitchInterval, there need to be another vertex preceding or following it. For the delta-viewpoints to work, then, we need at least two times two successive notes. Or put another way, it is a comparison of edges – since an edge has two notes. This is used in the *edge comparison* similarity measures. Not having delta-viewpoints in the vertex comparison viewpoints significantly reduces the specific-general ordering of viewpoints that we're able to reason about in the segmentation algorithm. We have therefore chosen to only build one vertex-based non-sequential similarity measure. It is called `nonSeq_PLIOS`. The letter combination 'PLIOS' signifies that it uses the five viewpoints cited above: note pitch (P), note length (L), `FOLLOW` in-edges (I), `FOLLOW` out-edges (O), and `SIMULTANEOUS` in-edges (S). The view comparators used in `nonSeqVertex_PLIOS` all compare individual vertex views in the same way, designed to give a number between 0 and 1. The further the views of v_1 and v_2 (say, the pitches of v_1 and v_2) are from each other, the closer the vertex view difference comes to 1; and if the views are equal, the view difference becomes 0. Such a vertex view becomes an entry in the view difference matrix of each view comparator:

$$VertexViewDifference_i(v_1, v_2) = 1 - \frac{1}{1 + |viewpoint_i(v_1) - viewpoint_i(v_2)|}$$

The similarity measure `nonSeq_PLIOS` begins by computing a weighted mean matrix M of its five view difference matrices. Experiments have shown that it is necessary to weight the pitch and length matrices stronger than the topological viewpoints. Pitch and length each are given $weight = 3$, and the three topological viewpoints are given $weight = 1$. Then, using this linear combined matrix M of similarities, a best match for each vertex in σ_1 is found in σ_2.

Best match Let's assume that σ_1 and σ_2 have the same size, n. Now we let each vertex in σ_1 choose a preferred match in σ_2. I.e., for each vertex $v_i \in \sigma_1$, we let its match be the vertex $v_j \in \sigma_2$ that has the lowest vertex comparison value with v_i. In other words, we find the smallest (best) value of row i of the weighted similarity matrix; the index of this value we call j. This gives us a set of matched vertex pairs (v_i, v_j). Let $bestmatch(i)$ be the number j of the vertex matched to v_i, and let $bestvalue(i)$ be the value of this match. This does not prevent several vertices from σ_1 from matching up with the same vertex $v_k \in \sigma_2$. Let $matches(k)$ be the number of vertices in σ_1 matched to vertex $v_k \in \sigma_2$.

Defining the similarity measure `nonSeqVertex_PLIOS` The final value of the similarity measure is computed as a mean value of the vertex comparisons

(v_i, v_j) in the best match. Only, we reward perfect vertex matches and punish multiple matches where v_i has chosen a v_j that has been chosen by other vertices, i.e. $matches(j) > 1$. This is done through the γ-function. Perfect matches we let have their 0.0 value if there are no other vertices from σ_1 that have preferred $v_j \in \sigma_2$ as a match. In other words, there should not be multiple matchings with v_j. We add a value $nonPerfectPenalty$ to non-perfect matches that are not multiple; and multiple matches we add an even greater value $multiplePenalty$ to.

$$\gamma(val, matches) = \begin{cases} 0.0 & \text{if val=0.0 and matches=1} \\ val + nonPerfectPenalty & \text{if val} > 0.0 \text{ and matches=1} \\ val + multiplePenalty & \text{if matches} > 1 \end{cases}$$

Experiments have shown the settings $nonPerfectPenalty = 3$ and $multiplePenalty = 6$ to be reasonable. Now we sum the γ-values of all (v_i, v_j) pairs in the best match. Remember that $j = bestmatch(i)$, so the sum can be written:

$$\sum_{i=1}^{n} \gamma(bestvalue(i), matches(bestmatch(i)))$$

n is the size of the subgraphs. We want the similarity measure to be a mean of the n γ-values, so we divide by n; also, we scale the value to be in [0;1] by dividing the sum by the maximal γ-value: $max(val + multiplePenalty) = 1 + multiplePenalty$, which is equal to 7 with our parameter settings. We can now write the similarity measure as:

$$\texttt{nonSeqVertex\_PLIOS}_1(\sigma_1, \sigma_2) = \frac{\sum_{i=1}^{n} \gamma(bestvalue(i), matches(bestmatch(i)))}{(multiplePenalty + 1) \times n}$$

We shall use the same size modifier as described for the sequential `seqMD` measures. This means that we have a problem if the γ-values sum to 0.0 – there would be no value for the size modifier to modify, cancelling the bonus that guides our GA towards larger matches. The GA then stays locked with very small ($n = 2$ or $n = 3$) matches. We solve this problem by adding 1 to both numerator and denominator in the fraction:

$$\texttt{nonSeqVertex\_PLIOS}_2(\sigma_1, \sigma_2) = \frac{1 + \sum_{i=1}^{n} \gamma(bestvalue(i), matches(bestmatch(i)))}{1 + (multiplePenalty + 1) \times n}$$

We have kept the equal size assumption on σ_1 and σ_2 although it is not really necessary for the method in `nonSeq_PLIOS`. For comparison of subgraphs with different sizes, we could have chosen to let all vertices in the largest subgraph choose a preferred match in the other subgraph; this would necessarily assign more multiple matches, because not all vertices in a larger subgraph can be matched to a single vertex in the smaller subgraph; and due to the added multiple matches, the mean γ-value will be worse, making subgraphs of equal size potentially more similar than subgraphs of unequal size. We have chosen not to use this for vertex comparison, but for edge comparison, it is necessary (see section 4.4.3). As it is, we apply the size modifier described in section 4.3.3:

$$sizeMod_3(weight) = weight + \frac{(1 - weight)\varphi}{n + \varphi - 1}$$

to the mean value, giving the final value:

$$\texttt{nonSeqVertex\_PLIOS}(\sigma_1, \sigma_2) = \texttt{nonSeqVertex\_PLIOS}_2(\sigma_1, \sigma_2) \times$$

$$sizeMod_3(\texttt{nonSeqVertex\_PLIOS}_2(\sigma_1, \sigma_2))$$

n is still the size of the subgraphs, and φ we set to 40 as for the sequential similarity measures.

An example of the use of `nonSeqVertex_PLIOS` Let's look at how the two subgraphs in figure 35 fare when measured using `nonSeqVertex_PLIOS`. They were taken from generation 40 of a SimFinder run with the following parameters:

Population size	50
Generations	40
Crossover	0.0
Mutation	0.3
Initial size	3
Max size	N/A
Fresh blood chance	0.2
Similarity measure	`nonSeqVertex_PLIOS`
φ	40

The parameter settings have been chosen loosely through experimentation. The SimFinder outputs:

```
NonSeqSimStmt of
        Size = 6
        Overlap=0
   nonSeqVertex_PLIOS=0.7504180887641042
   Similarities matrix=
        [0.61][0.39][0.42][0.70][0.48][0.47]
        [0.46][0.61][0.57][0.44][0.70][0.69]
        [0.62][0.20][0.51][0.59][0.11][0.40]
        [0.65][0.47][ 0.5][0.59][0.37][0.36]
        [0.68][0.53][0.44][0.57][0.44][0.42]
        [0.44][0.69][0.64][0.37][0.61][0.59]
   Best match=
        bestMatch[0]=1([0.39])
        bestMatch[1]=3([0.44])
        bestMatch[2]=4([0.11])
        bestMatch[3]=5([0.36])
        bestMatch[4]=5([0.42])
        bestMatch[5]=3([0.37])
   Matched notes=
  (SV of Note=<65,1.0> start=8.0)=(SV of Note=<69,1.0> start=16.0)
  (SV of Note=<52,0.5> start=8.5)=(SV of Note=<50,0.5> start=17.0)
  (SV of Note=<69,1.0> start=9.0)=(SV of Note=<69,1.0> start=17.0)
  (SV of Note=<65,1.0> start=9.0)=(SV of Note=<62,1.0> start=17.0)
  (SV of Note=<55,1.0> start=10.0)=(SV of Note=<62,1.0> start=17.0)
  (SV of Note=<53,0.5> start=11.0)=(SV of Note=<50,0.5> start=17.0)
   Multiple matchings in best=2
```

The output of the SimFinder for this non-sequential similarity statement tells us that the value `nonSeqVertex_PLIOS` $= 0.75$ is the overall value, and a bad one, reflecting the difference of σ_1 and σ_2. Below we're given other information:

1. The similarity matrix M that is the mean of the five view difference matrices. Take for example the first entry (0.61) which compares vertex 0 in σ_1 (v_{10}) to vertex 0 in σ_2 (v_{20}).

$$v_{10} = \begin{cases} \text{pitch} & 65 \\ \text{length} & 1.0 \\ \text{in-edges} & 0 \\ \text{out-edges} & 2 \\ \text{simult-edges} & 0 \end{cases} \quad \text{and} \quad v_{20} = \begin{cases} \text{pitch} & 74 \\ \text{length} & 0.5 \\ \text{in-edges} & 0 \\ \text{out-edges} & 0 \\ \text{simult-edges} & 2 \end{cases}$$

so this entry in the five individual view difference matrices will be:

$$\begin{array}{ll} \text{pitch} & 1 - \frac{1}{1+\lvert 65-74 \rvert} = 1 - \frac{1}{10} = 0.9 \\ \text{length} & 1 - \frac{1}{1+\lvert log_2(0.5) - log_2(1.0)) \rvert} = 1 - \frac{1}{2} = 0.5 \\ \text{in-edges} & 1 - \frac{1}{1+\lvert 0-0 \rvert} = 1 - \frac{1}{1} = 0.0 \\ \text{out-edges} & 1 - \frac{1}{1+\lvert 2-0 \rvert} = 1 - \frac{1}{3} = 0.66 \\ \text{simult-edges} & 1 - \frac{1}{1+\lvert 0-2 \rvert} = 1 - \frac{1}{3} = 0.66 \end{array}$$

$$\frac{3 \times 0.9 + 3 \times 0.5 + 0.0 + 0.66 + 0.66}{3+3+3} = \frac{2.7 + 1.5 + 0 + 0.66 + 0.66}{9} = \frac{5.53}{9} = 0.61$$

For note lengths, we use the more meaningful 'length interval' instead of the simple difference between the note lengths, hence the log_2 function.

2. The best match of each vertex in σ_1 as well as the value in the similarity matrix of that match. The pitch and length of the matched notes is also shown under 'Matched notes' to make it easier to see the exact matching. It is an important feature of the vertex comparison method that it actually gives us a complete best matching of the vertices. We use this information to intelligently mutate the subgraphs: an extension of two similar subgraphs is more likely to randomly pick another pair of similar vertices for inclusion if we extend the subgraphs from two vertices that already match in the subgraphs. We could have implemented the same trick in the edge comparison, but time has forbidden it.

3. The number of multiple matchings is given. As seen in the "best match" vector, both v_{11} and v_{15} in σ_1 are matched to v_{23} in σ_2, giving one multiple matching, and both v_{13} and v_{14} are matched to v_{25}, giving another multiple matching. This totals to two multiple matchings. The number of multiple matches is the number of vertices from σ_1 that would be without a match if the vertices of σ_2 were allowed to pick one, e.g. the best, of its "solicitors" from σ_1. 'SV' means *simple vertex* – as opposed to the *compound vertices* that we introduce in section 5.1.

Since σ_1 and σ_2 were so different, lets have a short look at the two subgraphs of another similarity statement in the same population as the similarity statement containing σ_1 and σ_2.

The SimFinder reported:

```
NonSeqSimStmt of
        Size = 6
        Overlap=0
```

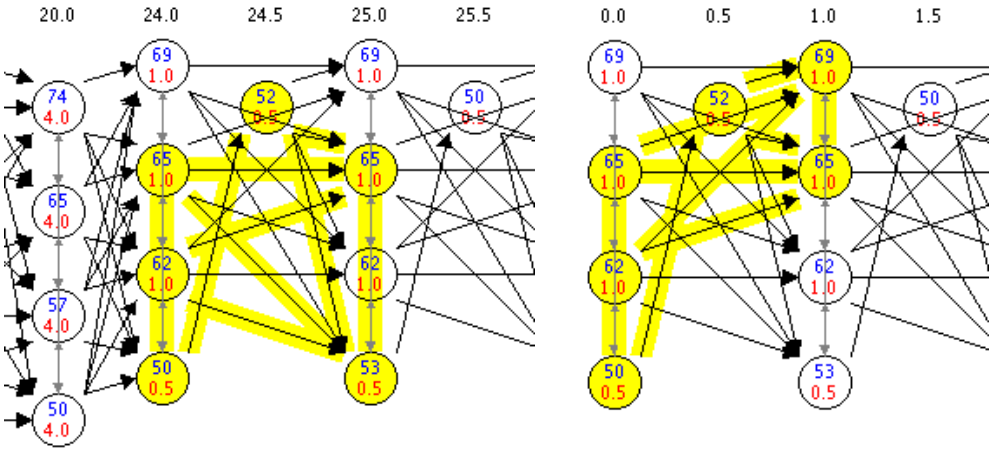


Figure 37: σ_3 and σ_4 are a much better match than σ_1 and σ_2 in figure 35.

```
nonSeqVertex_PLIOS=0.28713263693577823
Similarities matrix=
     [ 0.0][0.53][0.52][0.40][0.67][0.67]
     [0.53][ 0.0][0.25][0.60][0.47][0.21]
     [0.52][0.25][ 0.0][0.59][0.50][0.46]
     [0.40][0.60][0.59][ 0.0][0.68][0.67]
     [0.44][0.68][0.67][0.37][0.48][0.47]
     [0.67][0.21][0.46][0.67][0.26][ 0.0]
Best match=
     bestMatch[0]=0([ 0.0])
     bestMatch[1]=1([ 0.0])
     bestMatch[2]=2([ 0.0])
     bestMatch[3]=3([ 0.0])
     bestMatch[4]=3([0.37])
     bestMatch[5]=5([ 0.0])
Matched notes=
(SV of Note=<50,0.5> start=24.0)=(SV of Note=<50,0.5> start=0.0)
(SV of Note=<65,1.0> start=24.0)=(SV of Note=<65,1.0> start=0.0)
(SV of Note=<62,1.0> start=24.0)=(SV of Note=<62,1.0> start=0.0)
(SV of Note=<52,0.5> start=24.5)=(SV of Note=<52,0.5> start=0.5)
(SV of Note=<53,0.5> start=25.0)=(SV of Note=<52,0.5> start=0.5)
(SV of Note=<65,1.0> start=25.0)=(SV of Note=<65,1.0> start=1.0)
 Multiple matchings in best=1
```

The overall value close to 0.29 is clearly much better than the 0.75 of σ_1 and σ_2. Only the lower right note (53, 0.5) in σ_3 has trouble finding a good match – its best match has value 0.37, whereas all the other notes in σ_3 have 0.0. It thus chooses to be matched to the note (52,0.5) starting at 0.5. This produces one multiple match.

In the appendix (B.9) we show how the best found similarity in a SimFinder population evolves over 100 generations. The example is taken from a test run using the `NonSeqVertex_PLIOS` measure, and it shows that the GA is able to exploit found matches using this measure. The located match begins from the start of each of the two repetitions of the theme in BWV358 and is extended to the right. We think the series of pictures give a good impression of the way in which the SimFinder proceeds in its search. However, the GA does not always stumble upon a match of the two repetitions of the theme in BWV358, which

are rather large and allow for great expansion of the match, as shown in the series of pictures in B.9. More often, a small (but perfect) similarity of three to six notes is located in a place where the surroundings are different, so the match cannot be enlarged. This is somewhat problematic, because the GA seems to get stuck on these smaller matches, unable to jump to better locations in the graph. As we discuss in section 6.1, running several SimFinders with a periodic exchange of genetic material – the subgraphs – could perhaps improve on this.

Self-similarity of a graph through vertex comparison Vertex comparison can be used on its own – outside the SimFinder – to give a picture of the inner similarities in a piece. Figure 38 is a pictorial version of the weighted mean similarity matrix of the entire Bach chorale Jesu Meine Freude (BWV358), comparing all vertices to each other. The vertices are sorted by start time so that it is possible to see development in the temporal structure; from left to right, and from top to bottom, the vertices proceed from first to last. Thus a comparison of the first vertex and the last vertex in the piece may be found in the upper right corner, or alternatively (the matrix is symmetric) in the lower left corner. Darker colours mean little similarity, brighter colours mean more similarity. The whitest pixels have been replaced by red colour so that we can spot the areas of great similarity more easily.

The diagonal reflects the fact that the entire piece is equal to itself. The parallel line appearing after it shows that the chorale contains an exact repetition of the first six measures. In general, lines parallel with the diagonal show repetitions, and lines perpendicular to the diagonal show reverse occurrences of the same passage. There are no reverse examples here, but generally, small reversed passages occur not infrequently. The broad dark lines show where all four chorale voices end on a whole note; it is the length that makes these notes differ so much from the rest.

Complexity and evaluation To summarise, the vertex comparison is an approximation that allows us to compare non-sequential subgraphs on the basis of both graph topology and the musical attributes of vertices. The main drawback of this method is, that the computation is so heavy. The vertex comparison itself runs in $O(n^2)$, but where the real problem lies in the current implementation is in the computation of the best match: if a vertex is equally well matched with several vertices from the other graph, we have to choose its preferred match out of these. But the choice may, in conjunction with the choices of the other vertices, affect the number of multiple matches. We run through all possible combinations of best matches of all vertices in one subgraph. In the worst case this is an $O(n^n)$ task. But the worst case occurs only if the subgraph is a group (i.e., a chord) of notes with the exact same pitch and length, otherwise the topological or pitch and length viewpoints would make the matches differ, so that not all matches had the same similarity rating. In practice it never occurs, but once in a while the SimFinder hangs for a number of seconds due to such a computation. Generally, the more inner similarities (be it simultaneous similarities or temporally spaced similarities, i.e. repetitions) the two subgraphs have in common, the more severe the complexity of the vertex comparison. Bearing in mind, that this is only meant as an approximation to graph matching, we have built in a *skip* mechanism in the 'find best combination'-algorithm; if the computation

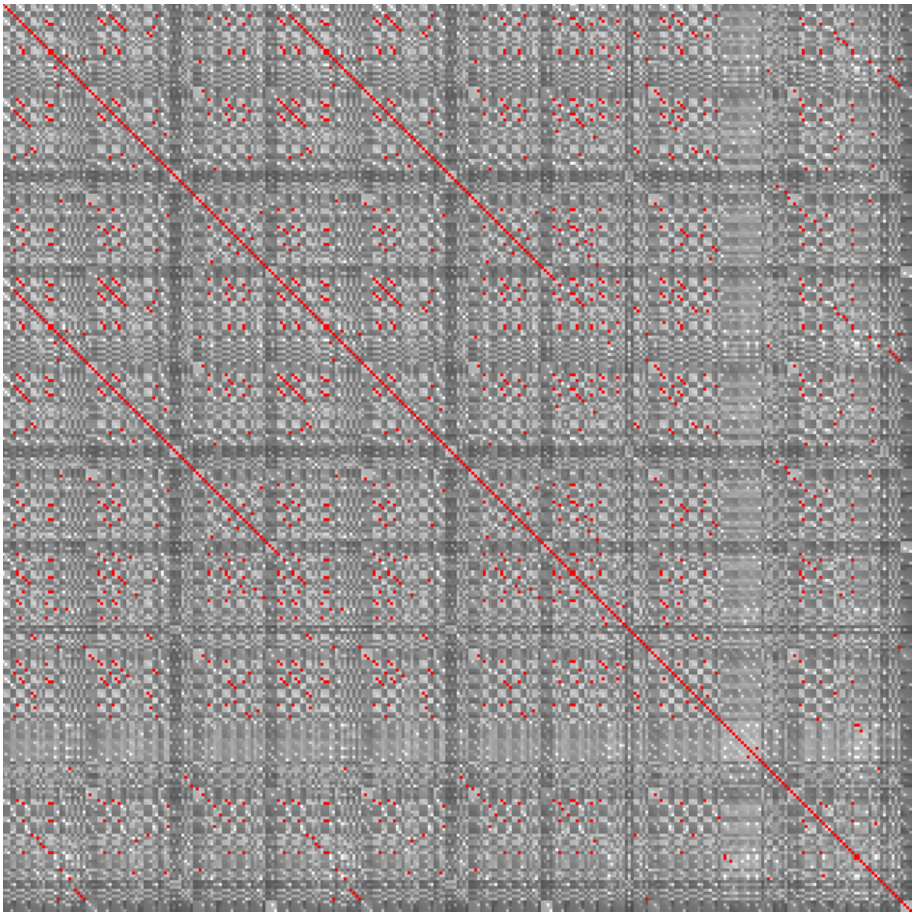


Figure 38: Self similarities of the Bach chorale Jesu Meine Freude BWV358 according to the nonSeqVertex_PLIOS similarity measure, shown in a 314×314 matrix. Computation time: approximately one minute. Red was substituted for similarity values under 0.01 (i.e. for the highest similarity values).

time for a given best combination exceeds some fixed time, say, one minute, the evaluation is skipped, and the similarity statement in question is assigned 1.0 in fitness. The combinatorial problem is much worse for the comparison of edges, where simply skipping complex calculations would ruin our chances of finding anything at all, and we have had to take other countermeasures.

In the vertex comparison, we are unable to use delta-viewpoints[36], since we compare vertices one against one. The "comparison of edges" method described in the next section is a solution to this.

Let's point out once more that the vertex comparison is an approximation. It is one-sided in that only vertices from one subgraph σ_1 are allowed to choose a preferred match in the other subgraph σ_2. This does not necessarily mean that

[36] That is, viewpoints giving the increase or decrease in some attribute, e.g. pitch, from vertex to vertex.

the chosen vertices in σ_2 would also have preferred the vertices that chose them. This is an area where a study of graph matching algorithms may improve very much on our method. We do however come up with an approximation to *the* best matching, which allows us to mutate more intelligently. The best match information might also be useful for more general reasoning on the relationship of the given subgraph with its surroundings in the mothergraph. We have not pursued this further yet.

4.4.3 Comparison of edges

The trouble with the vertex comparison method is that information about changes from note to note cannot be incorporated, since we look at only one vertex at a time. The 'comparison of edges' approach focuses on edges instead, thereby allowing comparison of the change between the vertices that one edge connects, with the change between the vertices that another edge connects.

The method used is almost exactly the same as for vertex comparison. For each of a number of viewpoints, we compute a similarity matrix showing comparisons of all edges in subgraph σ_1 with all edges in subgraph σ_2. These similarity matrices are combined into a weighted mean matrix; a vector of best matches is found, and a final score is computed on the basis of the values of the best matches. For details of the method, see the previous section (4.4.2) on vertex comparison. It turns out that it is beneficial to have a more strict evaluation of pitch and length than the view comparator described for vertex comparison. If the pitches (or lengths) of the notes connected by the two edges are exactly equal, a 0.0 value is given, otherwise a 1.0 value. The viewpoints in the list below which have been used in combination with this more strict view comparator have been labelled '(strict)'. Otherwise the vertex view comparator is used and a mean of the two vertex comparisons (compare the from-vertices of the two edges, and compare the to-vertices of the two edges) is given. The viewpoints we have defined are:

Acronym	Viewpoint	View comparator
AP	Absolute pitch	(strict)
PI	Pitch interval	(strict)
PC	Pitch contour	(strict)
DAP	Diatonic absolute pitch	(strict)
DI	Diatonic interval	(strict)
DFI	Diatonic forward interval	(strict)
AL	Absolute length	(strict)
LI	Length interval	
LC	Length contour	
-	In-edges	
-	Out-edges	
-	Simultaneous-edges	

Like for sequential similarity measures, we have opted to combine only one pitch related viewpoint and one length related viewpoint in each measure; except that for these non-sequential edge-based measures, the topological viewpoints concerning in-edges, out-edges and sim-edges are *always* included, so we don't include their acronyms in the names of the measures. We have defined the following similarity measures, naming them after the viewpoints above that are

included:

`nonSeqEdge_APAL`
`nonSeqEdge_PIAL`
`nonSeqEdge_PCAL`
`nonSeqEdge_DAPAL`
`nonSeqEdge_DIAL`
`nonSeqEdge_DFIAL`

Differences from the vertex comparison When comparing two non-sequential subgraphs of size n, they often have a different number of edges. The similarity matrices then are not square. We have chosen to let the vertices of the subgraph with *most edges* (say, $m + k$ edges) choose preferred matches in the subgraph with fewer edges (say, m edges). This way, there will automatically be more multiple matches, since $m+k$ edges cannot each have a singular match with one of m other edges. The increase in multiple matches then serves to automatically punish comparisons of subgraphs of different numbers of edges, which must have a different topology.

Another significant difference from the vertex comparison is that the number of edges most often grows much faster than the number of vertices when non-sequential subgraphs are extended. As an example, consider the evolving (i.e. growing) subgraphs shown in appendix B.9.

Generation	Vertices	Edges
8	3	2
14	6	17 (12 sim., 5 follow)
24	10	48 (24 sim., 20 follow)
80	19	99 (48 sim., 51 follow)

This makes the computation of edge comparisons much heavier, so we have introduced some countermeasures:

First, we have limited the edges considered to `FOLLOW` edges. There is really no need to do this, other than the wish to reduce the computation time. An obvious improvement to comparing all edges of σ_1 to all edges of σ_2 would be to compare `SIMULTANEOUS` edges of the two subgraphs separately, and `FOLLOW` edges separately; this is a good idea since comparing a `SIMULTANEOUS` edge with a `FOLLOW` edge will give a rating of 1.0 (i.e. 'unsimilar') anyway. We have not implemented this, only the mixed comparison of all edges has been tried out. It ran very slowly but otherwise seemed to work well. Ignoring `SIMULTANEOUS` edges in the present running version of SimFinder has one major drawback: a subgraph that is a group (i.e. a chord) is only connected by `SIMULTANEOUS` edges and therefore cannot be compared to any other subgraphs; it has no `FOLLOW` edges to compare.

Secondly, using the strict view comparator, where either 0.0 or 1.0 values are given, results in a lot of edges having solely 1.0 values for comparisons with all edges from the other subgraph. The algorithm that finds the best combination of matches now sees a combinatorial explosion in the number of match combinations it has to run through to find the best one. The solution has been to assign not exactly 1.0 but 1.0 minus some random microscopic amount. The net effect of this is that if an edge has only very awful matches to choose from, a random one is picked. This should not be too much of a problem

– choosing a match at random doesn't matter if all possible matches to the edge are equally bad anyway? But it does nevertheless make the `nonSeqEdge` measures non-deterministic. The bad subgraph matchings are more variable than the good ones.

The edge comparison in action The following screen shots show the best match located in a Bach 2-part invention (Inventio 1, BWV 772) after 30 generations.

SimFinder Settings	
Population size	50
Generations	30
Crossover	0.0
Mutation	0.5
Initial size	3
Fresh blood chance	0.4
Similarity measure	`nonSeqEdge_DIAL`
φ	40

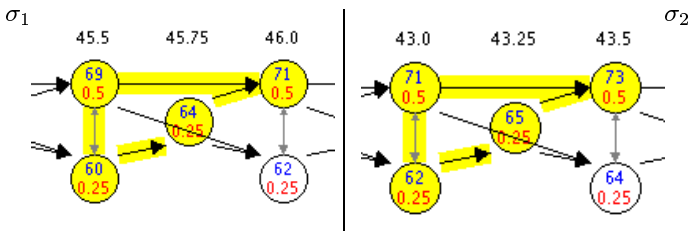


Figure 39: Subgraphs σ_1 and σ_2 in a similarity found using edge comparison with the diatonic interval viewpoint (DI).

There are only three edges in each of these small fragments, and they match perfectly, as the 0.0 values in the best match vector show. The similarity measure used diatonic interval as its tonal viewpoint. This means that the two fragments are diatonic transpositions of each other. Notice that the interval between pitches 60 and 64 in σ_1 is 4, while the interval between pitches 62 and 65 in σ_1 is 3. But the diatonic interval in both is 2. Thus a simple pitch interval (PI) viewpoint would not have found this similarity. The calculation of the similarity measure proceeds in the same way as described for vertex comparison.

```
NonSeqSimStmt of
        Size = 4
        Overlap=0
   nonSeqEdgeDIAL_nocompounds=0.042427445704401316
   Similarities matrix=
        [ 0.0][0.80][0.72]
        [0.80][ 0.0][0.86]
        [0.72][0.86][ 0.0]
   Best match=
        bestMatch[0]=0([ 0.0])
```

```
      bestMatch[1]=1([ 0.0])
      bestMatch[2]=2([ 0.0])
Multiple matchings in best=0
```

Summary and evaluation Generally, the search using vertex and edge comparison is heavier computationally, but also more difficult to control in the GA. The evaluation scheme seems to work reasonably well (modulo the $O(n^n)$ complexity of the algorithm that produces the best combination of best matches), and the found matches really are similar. But it seems we cannot expect to always find what there is to be found. It is difficult to tune the SimFinder so as to be able to both explore and exploit. Often, it gets stuck in a moderately bad match, e.g. when the match cannot be improved by extending or changing it, and it seems that the algorithm cannot escape this local optimum. If mutation is raised to alleviate this, we often find a best match that *could* profitably be extended, but in fact isn't extended, so we guess that the mutation has been raised too much. It is a more difficult balancing act to find suitable mutation settings for the non-sequential vertex and edge comparisons.

Due to the uncertainty thus involved in the present implementation, we have chosen to focus more on the last non-sequential method, we have devised: using an intersection ratio of bags of sequential subgraphs of the non-sequential subgraphs. Particularly in the segmentation algorithm, which repeatedly uses the SimFinder to find building blocks for the segmentation, uncertainties of successive SimFinder runs multiply and add up to a more unstable segmentation. We have also included grouping viewpoints in this third and last non-sequential similarity measure, making it even more suitable for segmentation purposes.

4.4.4 Comparison of bags of subgraphs

Instead of explicitly matching single vertices and edges to each other and in the same relations, we have tried another simpler approach by examining if a vertex or edge in one subgraph σ_1 relate to any vertex or edge in the other subgraph σ_2 and by counting the differences give a value for how similar σ_1 and σ_2 are. This can be done by comparing vertices and edges (`SIMULTANEOUS` and `FOLLOW`), but also by comparing the sets of all sequential subgraphs that can be found in each of the two graphs (all melodies) to be compared. Actually these sets are bags of sequential subgraphs, since a sequential subgraph may occur more than once in a graph. If the bags of sequential subgraphs are alike for all length of the subgraphs, the graphs truly are alike. The graphs in Figures 40 will have the

Figure 40: The graphs have equal sets of `FOLLOW` edges.

same bags of sequential subgraphs of length 2 (their edges), but their sequential subgraphs of length 3 will differ. Therefore we would like to evaluate the bags of all sequential subgraphs of length $m > 2$ as well as the `FOLLOW` edges.

BagIR Let bag_i be the bag of all desired elements of a subgraph σ_i. Then we define the similarity of the content of two bags in terms of this function:

$$BagIR(bag_1, bag_2) = 1 - \frac{1}{2}(\frac{|bag_1 \cap bag_2|}{|bag_1|} + \frac{|bag_1 \cap bag_2|}{|bag_2|})$$

The co-domain of BagIR is [0,1], where 0 means complete equality of the bags and 1 means complete inequality, i.e. the intersection of the two bags is empty. $BagIR(bag_1, bag_2)$ is thus a measure of to what extent all elements of the two bags (no matter of what kind the elements are) are alike.

Applying BagIR In our implementation of BagIR, we have used a string encoding of the elements of the bags, and a Trie datastructure is used to compare the content of the bags.

The idea is to fill the bags with encodings of elements (vertices, edges or sequential subgraphs) of the non-sequential subgraphs and to do this according to a desired sequential view of the elements.

Given two bags (one for each subgraph) of whichever content we would like to compare, each element of the first bag is encoded as a string (according to what we are comparing) and added to the trie. The trie counts the occurrences of doublets. The content of the other bag is then also converted to the same string format. Each string is tested for membership in the trie. If it is a member, the occurrence of the string is removed from the trie. We count the number of strings found in the trie. In the end we know how many strings the bags have in common, and we can calculate the intersection ratio.

As mentioned we have experimented with four uses of the BagIR algorithm:

- $BagIR_{Vertex}$ compares encodings of all vertices in the two subgraphs. This can only be done with an absolute view of the vertices, since we try to match the vertices with each other, not regarding their relations.
- $BagIR_{Sim}$ compares encodings of all `SIMULTANEOUS` edges in the two subgraphs according to the wanted view (absolute or relative) of the edge. Also some information about the vertices the edge is connecting can be encoded.
- $BagIR_{Follow}$ for `FOLLOW` edges compares encodings of all `FOLLOW` edges in the two subgraphs according to the wanted view (absolute or relative) of the edge. Information about the vertices the edge is connecting can be encoded.
- $BagIR_{SS(n)}$ for sequential subgraphs compares encodings of views (both absolute and relative) of all possible sequential subgraphs of size n in the two subgraphs. The former case is a special case of this one, where $n = 2$.

The BagIR idea can thus be used for both the absolute views (only nodes – not their relation) and the relative views (delta views) on nodes and edges. Since the BagIR compares sequential subgraphs, vertices and edges, we use some of the same views as in the sequential case: Diatonic Absolute Pitch (DAP), Diatonic Interval (DI), Diatonic Forward Interval (DFI), and also the midi pitch based Pitch Interval (PI).

We will now describe the encodings: The absolute viewpoint we have used for encoding vertices is the DAP, so that each note is encoded as string telling the name, octave, accidentals, and duration. The relative viewpoints used are PI, DI and DFI. An edge is encoded as the duration of the first vertex, followed by the view of the edge (the view of the relation between the vertices) followed by the duration of the second vertex. To be more explicit, vertices and edges are encoded like this:

encode absolute	Vertex	$enc(v)_{DAP}$	$\rightarrow$	$name_v\_octave_v\_accidentals_v\_duration_v$
	Edge	$enc(v_1, v_2)_{DAP}$	$\rightarrow$	$enc(v_1)_{DAP}\_enc(v_2)_{DAP}$
encode relative	Vertex	$enc(v)_{vp \in \{PI,DI,DFI\}}$	$\rightarrow$	$duration_v$
	Edge	$enc(v_1, v_2)_{view \in \{PI,DI,DFI\}}$	$\rightarrow$	$enc(v_1)_{view}\_view(v_1, v_2)\_enc(v_2)_{view}$

When encoding a sequential graph we simply use the latter encoding scheme too:

$enc(\sigma = (v_1, v_2, \ldots, v_n))_{view \in \{DAP,PI,DI,DFI\}}$	$\rightarrow$	$enc(v_1, v_2)_{view}$; $enc(v_2, v_3)_{view}$; $\ldots$; $enc(v_{n-1}, v_n)_{view}$;

For example, the first graph in figure 40 would under the view PI for $BagIR_{SS(3)}$ be encoded as one string: `1.0_5_1.0;1.0_-5_1.0;`. This string is then put in the first bag (and will be the only element) for the $BagIR_{SS(3)}$ algorithm.

We have 5 view comparators: 4 using the BagIR algorithm on different sequential viewpoints and one non-sequential grouping structure viewpoint. The first rely on all four uses of the intersection ratio, whereas the next three only use the relative sequential viewpoints.

- $bagIRTrieDAP$ which calculates $BagIR_{Vertex,DAP}(\sigma_1, \sigma_2) + BagIR_{Sim,DAP}(\sigma_1, \sigma_2) + BagIR_{Follow,DAP}(\sigma_1, \sigma_2) + BagIR_{SS(n),DAP}(\sigma_1, \sigma_2)$ for a suitable n.
- $bagIRTriePI$ which calculates $BagIR_{Sim,PI}(\sigma_1, \sigma_2) + BagIR_{Follow,PI}(\sigma_1, \sigma_2) + BagIR_{SS(n),PI}(\sigma_1, \sigma_2)$ for a suitable n.
- $bagIRTrieDI$ which calculates $BagIR_{Sim,DI}(\sigma_1, \sigma_2) + BagIR_{Follow,DI}(\sigma_1, \sigma_2) + BagIR_{SS(n),DI}(\sigma_1, \sigma_2)$ for a suitable n.
- $bagIRTrieDFI$ which calculates $BagIR_{Sim,DFI}(\sigma_1, \sigma_2) + BagIR_{Follow,DFI}(\sigma_1, \sigma_2) + BagIR_{SS(n),DFI}(\sigma_1, \sigma_2)$ for a suitable n.
- nonSeqGroupingStructure

The length n of the sequential subgraphs used in $BagIR_{SS(n),DFI}$ is calculated according to the subgraphs in question. We calculate the length of the longest possible sequential subgraph in each of the graphs. n should be shorter than both of them (to get more than one string in the bag from one of the graphs), so we take half the length of the shortest of the longest:

$n = min(maxLengthSS(\sigma_1), maxLengthSS(\sigma_2))2/2$.

The viewcomparators are combined into four NonSeqSimMeasures which combine the BagIR based view comparators with the grouping structure:

- `nonSeqBagIR_DAP_Grp`

- `nonSeqBagIR_PI_Grp`
- `nonSeqBagIR_DI_Grp`
- `nonSeqBagIR_DFI_Grp`

The NonSeqSimMeasures are calculated in ways similar to each other, so we present the standard one here:

$$\texttt{nonSeqBagIRTrie\_VIEW} = bagIRTrie_{view}(\sigma_1, \sigma_2) + grouping(\sigma_1, \sigma_2) + 1 - \frac{Size(\sigma_1)}{100.0}$$

Remember that the nonSeqBagIRTrie calculates and adds the intersection ratios for vertices (only under the DAP viewpoint), `SIMULTANEOUS` edges, `FOLLOW` edges and sequential subgraphs of size n. All this topological information contributes to the overall fitness of the similarity statement. There is a punishment if $BagIR_{Sim}$ and $BagIR_{Follow}$ returns 1.0 (no matches). This is to help the GA to forget about this similarity statement right away.

It is not always possible to calculate the respective ratings. For example if a graph does not have `FOLLOW` edges we cannot calculate $BagIR_{Follow}$ or $BagIR_{SS(n)}$. Then the view comparators get a bit degenerate. If this happens, the GA can have a hard time to extend the subgraphs to contain `FOLLOW` edges, since whenever it tries to expand vertices along `FOLLOW` edges, it gets a worse score if they do not match right away, and it is happier with the graph with only `SIMULTANEOUS` edges. We do of course have to allow subgraphs existing solely of simultaneous edges in order to match for example chords.

The BagIR method counts *exact* matches in the encoded elements of the bags. Two similar subgraphs, for example differing only on one duration of one note will get different encodings and count as a mismatch, so we cannot judge this as a 'almost perfect' match as we are able to do when searching for sequential subgraphs in general. The problem is that we don't know which strings to compare to each other. One thing we can do is to encode the graphs less explicitly – for example by excluding the rhythmic material. This would make the search totally pitch dependent however.

The BagIR similarity measures have proven to work very well in searching for similarities in a non-sequential graph. The calculation time however seems to explode when the subgraphs grow too large. We present here two examples of using the SimFinder with this measure. Furthermore we give a larger example in the evaluation of our system in chapter 5.5.3. The first is a run for a couple of minutes. It found a perfect match of two subgraphs, and a perfectly grouped one as well – see figure 41.

Population size	80
Generations	200
Crossover	0.0
Mutation	0.5
Initial size	5
Fresh blood chance	0.4
Similarity measure	`nonSeqBagIR_DI_Grp`
γ	0.5

Here is the evaluation from the last generation:

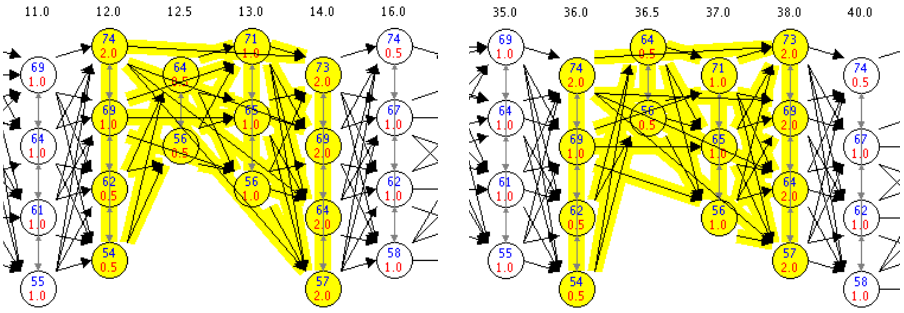


Figure 41: Similarity found with the `nonSeqBagIR_DI_Grp`.

```
NonSeqSimStmt of
        Size = 13
        Overlap=0
nonSeqBagIRTrie_DiatonicInt_Grp = 1.3425543478260868
        Size Penalty=0.87
        Sequence rating, BagLength=3=0.0
        { Match size=48
          Bag 1 size=48
          Bag 2 size=48 }
        Sequence rating, BagLength=2=0.0
        Sequence rating for Vertices=-1.0
        Sequence rating for SimEdges=0.0
        nonSeqGroupingStructure = 0.47255434782608696
```

The size of the match is 13. The value of the measure is = 1.3425. The $BagIR_{SS(n)}$ was calculated for $n = 3$. The graphs have 48 of 48 matching sequential subgraphs of size 3, giving a score of 0.0. The `SIMULTANEOUS` edges and `FOLLOW` edges are also equal. The grouping value is 0.47255. The size value is 0.87. Both of the subgraphs have their last notes on a fermata. So the overall score 1.3425 is the sum of $BagIR_{SS(n)}$ + $BagIR_{Follow}$ + $BagIR_{Vertex}$ + sizeValue + groupingValue = 0+0+0+0.87+0.47255 = 1.3425.

By changing one note in each graph (see figure 42), the result is remarkable. The value of the measure is now = 2.4003. The $BagIR_{SS(n)}$ was calculated for $n = 3$. The bags of sequential subgraphs of the two subgraphs now have different sizes: The size of the first bag is 51 (sequential subgraphs of size 3) and the size of other bag is 60. The match size is 40, so the intersection ratio is 0.2745. $BagIR_{Follow}$ = 0.1927, $BagIR_{Sim}$ = 0.2307. The grouping value is now 0.8322.

The last example is less detailed, but a nice example of what similarities there are to find in a small chorale. Figure 43 shows a cadence in two different keys – the first cadencing on F major and the last on D major. The measure was `nonSeqBagIR_PI` (no grouping, and no $BagIR_{Vertex}$ was calculated).

4.5 Summary

We have now presented the graph representation and we hope that the reader is convinced of its versatility and that it really does let us out of the sequen-

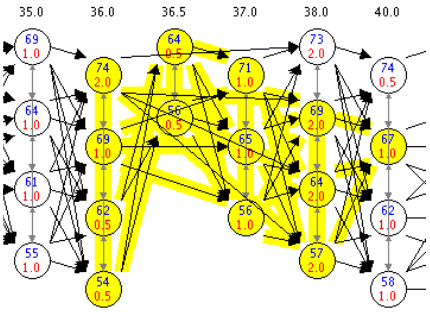


Figure 42: By changing one note in each subgraph of figure 41, the fitness drops dramatically.

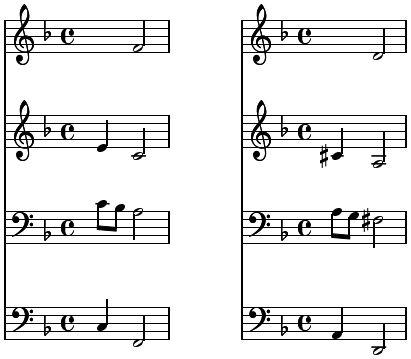


Figure 43: Two similar cadences from the chorale BWV 358 found with the `nonSeqBagIR_PI` similarity measure. The cadences are direct transpositions of each other.

tial/parallel dichotomy. The graph is rather a complicated structure, however[37], and searching for exactly matching subgraphs is NP-complete. The chosen approximation is a GA imbued with a multiple viewpoint system and with several other requirements on chosen subgraph matches built into the fitness function. Sequential subgraphs are much easier to compare, so we implement the search for sequential similarities as one type of search and the search for non-sequential similarities as another.

Viewpoints We have defined a number of absolute and relative viewpoints operating on the note attributes of vertices. Most of our viewpoints are also included in the set of viewpoints proposed by Conklin and Witten [CW95], and all of their viewpoints are possible to implement, if we operate on MuseData files. But our diatonic viewpoints have no equivalent in Conklin and Witten's

[37] As Curtis Roads predicted, see the citation on p.17.

article. As we argued in section 3.1.1, there are important variations, that would not be found without the diatonic viewpoints, so this is an improvement.

Viewpoints concerning pitch and length respectively have a specific-to-general ordering; a perfect subgraph match under a very specific pitch viewpoint will also be a perfect match under a more general pitch viewpoint. A viewpoint is a transformation under which we hope to find subgraphs to be equal, and the type of the viewpoint therefore tells us which kind of relation two similar subgraphs are in. This we shall use more in the next section on graph grammars and the segmentation algorithm. The grouping viewpoint is not a real viewpoint in this respect but more of a heuristic introduced to guide the search of the GA into comparing more musically logically connected subgraphs. It is one of five fundamental and conflicting goals for the GA, which we shall come back to in a moment.

The SimFinder is an experimental system. The design is based on the modularity of viewpoints and similarity measures, which allows us to easily invent and replace viewpoints and measures.

We have experimented with two sequential view comparators, one percentual (mean difference) and one which isn't (difference count). This gives rise to two families of sequential similarity measures. We have found both the `seqDC` and the `seqMD` families to work. The seqMD measures are more complex, because we have striven to constrain their value to [0;1] in a smooth way. In practice the `seqDC` measures are more easy to interpret because they are so much simpler.

Sequential and non-sequential search We have tuned the GA a little for sequential search. Crossover is a bad operation, so it is very important to get the mutation operations correctly tuned. The SimFinder usually settles on an area and exploits it. Exploration of other areas *does* happen, but most often the algorithm has a favourite spot in the graph where most of the vertex usage is concentrated.

A convenient exploration/exploitation balance seems harder to strike for non-sequential searches. In particular, the mutation parameters are difficult to tune into a good balance. The most stable of the three non-sequential comparison methods seems to be the one based on the intersection of bags of encodings of information of smaller parts of the two subgraphs, so we have also incorporated the grouping viewpoint in a non-sequential version into this (bag intersection ratio) measure. All the proposed non-sequential comparison may be used on subgraphs of different sizes, preferring matches where the subgraphs have the exact same size[38]. A nice feature of the vertex comparison however is that it actually gives us a matching of all vertices in the two compared subgraphs. This information can be used intelligently. In general, the non-sequential similarity measures are much more computationally complex, and in some cases, we need to restrict the computation time spent on evaluation in order to keep the SimFinder running.

Computability Let's recapitulate on the points of critique of the GTTM (see the summaries in sections 2.5 and 3.3). We have described in more detail how to use a graph for the representation and analysis of non-monophonic music, and

[38] In the implementation, however, we still require the SimFinder to locate subgraphs of the same size, i.e. having the same number of vertices.

we have described how we use a multiple viewpoint system to detect a subset of musical parallelisms, namely repetition and simple transformations. And through the implementation of the SimFinder, the third goal of *computability* is reached.

We have chosen to construct numerical evaluations of similarity, embodied in our similarity measures. Such constructs are bound to include a lot of fiddling with equations and numbers to produce any sensible rating. But in the SimFinder, we have avoided thresholding, as Lerdahl and Jackendoff criticise (see p.22). The search of the SimFinder is based only on the pairwise *relative* fitnesses of randomly chosen similarity statements in the selection process. Thus it does not guarantee the best found match to be good, only that it is better than the other matches considered. In the next section on the segmentation algorithm, we will introduce thresholds as a means to control the SimFinder from the SimSegmenter.

The SimFinder is a search that balances the five conflicting goals that we mentioned in section 4.3:

- maximise the musical similarity in terms of equality under viewpoint transformations,
- maximise the size of subgraph matches,
- prohibit overlap,
- maximise subgraph agreement with grouping and phrasing boundaries in the surrounding graph, and
- maximise the 'importance' of the located subgraphs, in terms of the number of occurrences present in the entire graph.

We hope that the test runs in the next section will give a better impression of how good a balance or tradeoff we have struck between these five goals.

5 Graph grammars

A good similarity measure gives us a starting point for building a grammar for a given piece of music. If monophonic music represented in strings of notes can be described using a grammar, non-monophonic music represented in a graph could possibly be described using a graph grammar. This section describes our efforts to use the developed similarity measures to segment a piece and build a graph grammar for it.

The passages identified as similar by the SimFinder are used as building blocks in the overall structure induced from the music. Segmentation is a recursive process of finding interesting recurrent patterns, identifying all instances of these patterns, and substituting the instances in the graph with compound vertices that represent the different versions of each pattern.

The idea here in fact is close to Dannenberg's vision that we cited in section 4.3.1. The method of locating patterns and substituting all occurrences of them with compound vertices in the graph is akin to the idea that each motif be represented globally only once, while each occurrence of the motif is a local view on it. Dannenberg points out some naturally arising problems concerning how shallow the local view should be. "If a note in a view is edited, and then the original note in the motif is deleted, should the view's note be deleted as well? Can the user control such decisions? Can views be nested?" [Dan93, p.26] The answers to these questions depend on implementation. Let's first look more closely at the mechanisms at play, and we'll return to the answers in section 5.4.3.

5.1 Extensions of the MusicGraph

Before we can explore the segmentation algorithm further, we need to extend the definition of a MusicGraph, as well as many ideas introduced in the SimFinder, to accommodate the new so-called *Compound MusicVertices*.

Definition 5.1 *A simple vertex is a vertex that contains information on a single note. A compound vertex C is a vertex that contains a subgraph σ_C instead of note information. Substitution of a subgraph σ of the mothergraph G with a compound vertex C such that $\sigma_C = \sigma$ is called a compound substitution. If a graph contains no compound vertices, it is a simple graph; if it does contain at least one compound vertex, it is a compound graph.*

A first requirement for compound substitution is that the operation be reversible, so that we may accurately restore the original simple graph from a compound graph. We also want the `FOLLOW` and `SIMULTANEOUS` relations represented by edges in the graph to hold for the compound graph resulting from the compound substitution.

Consider the example graph in figure 44[39]. Imagine that the vertices *c-e-f* (the letters refer to vertex names – not pitch names) have been identified as

[39] Please note that the example graph given in figure 44 is a somewhat artificial one, since most often, parts are monophonic. In that case, figure 44 would represent at least a two-part passage (since there are several occurrences of two simultaneous notes). In that case, the notes *e* and *h* each would also have a simultaneous rest of length 1.0 and 4.0 respectively, because rests are explicitly given in traditional music notation. This would solve some of the problems discussed below, but on the other hand, figure 44 might still occur, e.g. in a piano part; each part corresponds to a hand, which can play several simultaneous notes. Or figure 44 could

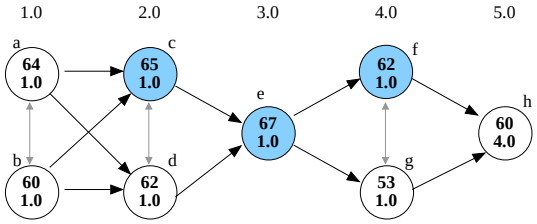


Figure 44: An artificial graph example. Vertices c-e-f will be substituted by a compound vertex.

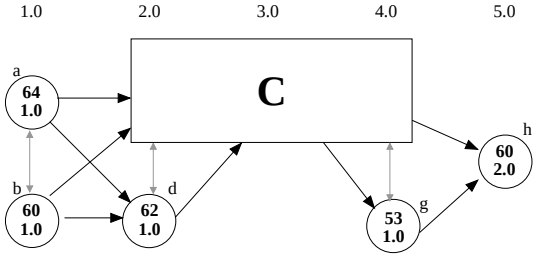


Figure 45: Leaving all existing edges in place, only going to and from C instead of to and from σ_C. Both the `FOLLOW` and the `SIMULTANEOUS` relations are wrecked.

a pattern to be substituted by a compound vertex C. The vertex substitution yields the graph in figure 45. Here all edges going between a vertex v_x outside C and a vertex v_y in σ_C have been replaced by edges going between v_x and C. The problem with this edge substitution is that it wrecks the `FOLLOW` and the `SIMULTANEOUS` relations. According to figure 45, d is simultaneous with C, which in turn is simultaneous with g. Thus, by transitivity of the `SIMULTANEOUS` relation, d should be simultaneous with g, which it clearly isn't. Also, there is something wrong with the temporal relations when e.g. C follows immediately upon a, while C also follows immediately upon d, that follows immediately on a. It seems that d would need to have a length of 0.0 for this to be possible, but it has length 1.0. d and C having both `SIMULTANEOUS` and `FOLLOW` edges between them also requires that d have length 0.0 to be possible, so it seems that this strategy cannot be used.

Another strategy for edge substitution could be to remove all edges to and from vertices in σ_C and regard C as any simple vertex, i.e. having a start time and a length. This situation is shown in figure 46. C begins at the same time as d and thus has `SIMULTANEOUS` edges to and from d. C follows immediately on a and on b, and h follows immediately on C, giving `FOLLOW` edges (a, C), (b, C), and (C, h). A problem arises here with d which no longer has any immediately following vertices to connect to at its ending time 3.0. Surely, g follows on d but not immediately. Therefore it would be wrong to add a `FOLLOW` edge between

be the subgraph σ_C contained in a compound vertex C, in which case there might have been other notes or rests simultaneous with e and h and formerly connected to them but that are now outside C.

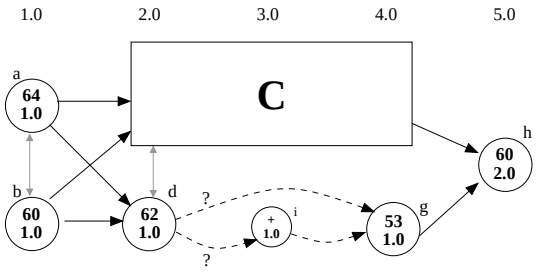


Figure 46: Treating the compound C as a simple vertex with start time 2.0 and length 3.0.

d and g. Another solution could be to add a rest vertex i with the missing length 1.0 between d and g. This solution, we feel, is too complex and difficult to work with. We would have to keep track of which vertices were inserted (and which vertices were original) in order to do the reverse compound substitution. Also, separating C completely from g and inserting i is a change of the graph structure that could possibly influence what similarities can be found in the piece. If there exists a sequence of pitches and pauses [62,+,53] with lengths 1.0 somewhere else in the piece, it will now be matched perfectly with the sequence [d, i, g] although the note e is still sounding inside C during i. It is unclear what such changes would amount to when running the recursive SimSegment algorithm and substituting many different patterns with compound vertices.

Instead we propose the following solution:

1. In a compound substitution, keep all edges to and from σ_C as edges to and from C (just like in figure 45).

2. Distinguish between the *endpoints* of an edge (simple or compound vertices – whatever things that edge connects) and the *anchors* of an edge (simple vertices, be they still present in the graph or abstracted away behind one or more layers of compounds). This is explained further below.

3. Distinguish between *strong* and *weak* `FOLLOW` and `SIMULTANEOUS` relations. Edges representing strong relations uphold the relations we have been speaking of as `FOLLOW` or `SIMULTANEOUS` until now. Edges representing weak relations may e.g. break the transitivity of the `SIMULTANEOUS` relation, but they state that there once existed a strong relation between the anchors of the edge.

Definition 5.2 *Let e_{ij} be an edge. If an endpoint i or j is a compound vertex C, then the anchor of e_{ij} in C is the simple vertex v_a inside σ_C, that e_{ij} connected to before σ_C was substituted by C. Locating the anchor of an edge inside a compound vertex is called anchor resolution. If an endpoint i or j is a simple vertex, then that vertex itself is the anchor of e_{ij}.*

Anchor resolution can be accomplished by sorting vertices in σ_C according to some total ordering on vertices. The index in the sorted list of the anchor v_a is then saved in e_{ij}, such that each edge "knows" what it is anchored to inside a compound. Resolving the anchor of e_{ij} now amounts to looking up the anchor's

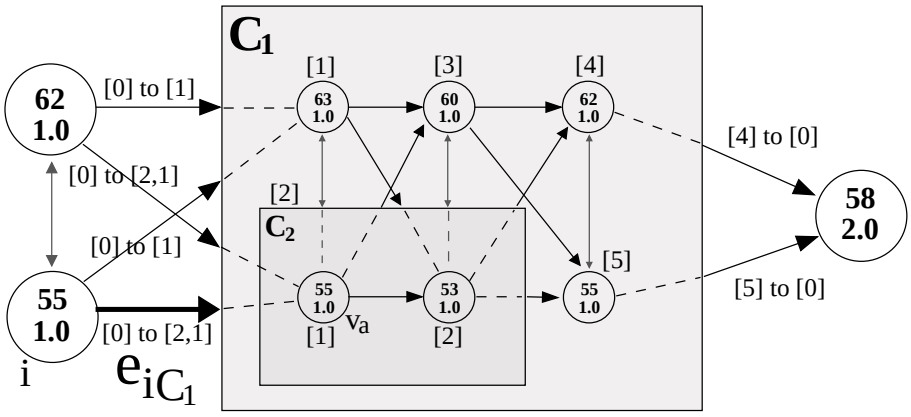


Figure 47: An edge e_{iC_1} connected to a compound vertex C_1 but whose anchor vertex v_a is hidden inside yet another compound C_2. Therefore both the index of C_2 inside C_1 (which is [2]) and the index of v_a inside C_2 (which is [1]) are needed to resolve v_a from e_{iC_1}. Indices of vertices inside the compounds are shown in square brackets, and likewise, lists of indices for edge endpoints are shown in square brackets. For simplicity, the lists of indices on edges are only shown to and from C_1, although all edges having a compound as an endpoint should have indices pointing the way to their anchor vertex.

index inside σ_C's list of vertices. Since a directed edge may be connected to compounds both at the originating vertex and at the destination vertex, an edge should have both an *origin anchor* and a *destination anchor*. In the basic case where the endpoint is a simple vertex, the anchor can be considered as the simple vertex itself. Then that endpoint will have *anchor index* $=[0]$.

What if the vertex C_2 that e_{ij} connects to inside σ_{C_1} is itself a compound vertex? Then clearly C_2 is not the anchor of e_{ij} (since an anchor must be a simple vertex), but some other vertex inside σ_{C_2} must be the anchor. We need another number – the index of the anchor inside σ_{C_2} – to find the anchor. Since compound vertices can contain nested compounds in any number of layers, we may need an entire list of indices to peel through the onion layers of compounds to get to the right anchor vertex of an edge. This situation is illustrated in figure 47, where the index list [2,1] allows us to first resolve e_{iC_1}'s anchor to C_2 (index [2] inside C_1) and then finally to v_a (index [1] inside C_2).

As mentioned, leaving all edges in place as in figure 45 wrecks the `FOLLOW` and `SIMULTANEOUS` relations. Our solution to this is to distinguish between *strong* and *weak* relations. Strong `FOLLOW` and `SIMULTANEOUS` relations are as defined in definitions 4.1 and 4.3. When substituting subgraph σ with compound vertex C, we replace all edges connecting σ and the surrounding graph with edges connecting C and the surrounding graph. Looking once more at figure 45, the `FOLLOW` edges e_{aC}, e_{bC}, and e_{Ch} still hold, since a and b end precisely at the time when C starts. But e_{dC} and e_{Cg} no longer hold. These two edges become edges of type `WEAKFOLLOW`. This means that there once was a `FOLLOW` relation (the strong relation) between the two anchors of each of these edges. This expresses that in some weak sense, C still follows upon d (since it keeps sounding *after* d

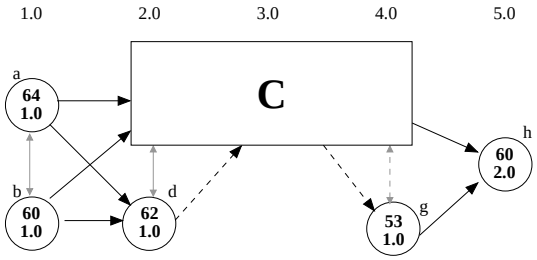


Figure 48: All edges present in figure 44 going to and from σ_C have been replaced with (strong) `FOLLOW` and `SIMULTANEOUS` edges to and from C where the relations still hold, and `WEAKFOLLOW` and `WEAKSIMULTANEOUS` edges elsewhere. The weak edges have dashed lines.

has ended), and likewise g weakly follows upon C.

Similarly, the `SIMULTANEOUS` edges between C and d still hold, because C and d have the same start time. On the other hand, the `SIMULTANEOUS` edges between C and g should be replaced by `WEAKSIMULTANEOUS` edges, expressing that once, before any compound substitutions were made, there existed a strong `SIMULTANEOUS` relation between the anchors of these edges. C and g are simultaneous in a weak sense, since some inner part of C begins simultaneously with g. The `SIMULTANEOUS` relation remains transitive, but the `WEAKSIMULTANEOUS` relation is *not* transitive.

Weak relations will be drawn with dashed lines in the graphical representation, as shown in figure 48. Please note that the GUI part of the SimFinder system is not completely finished; sometimes, `WEAKSIMULTANEOUS` edges are shown as solid lines, because drawing several dashed lines on top of each other creates a solid line. It should be clear from the placement of the edge relative to the compound vertex which kind of edge it is: only `SIMULTANEOUS` edges connected to the *first* beat of the compound should be strong. Also, sometimes `WEAKFOLLOW` edges connected to the last beat of a compound are ending a little to the right of the compound. We apologise for these mistakes and hope that it does not hinder the understanding of the graphs substantially.

We end this section by defining a few properties of compound vertices.

Definition 5.3 *Let C be a compound vertex containing subgraph $\sigma = (V_\sigma, E_\sigma)$. Then*
$startTime(C) = \min_{v \in V_\sigma}(startTime(v))$*, and*
$endTime(C) = \max_{v \in V_\sigma}(endTime(v))$*, and*
$length(C) = endTime(C) - startTime(C)$
Nesting of compounds introduces nested size and depth of vertices:
$size(C) = size(\sigma) = |V_\sigma|$
$nestedSize(C) = \sum_{i=1}^{|V_\sigma|} nestedSize(v_i)$*, where*
$nestedSize(v) = 1$ *for simple vertices.*

$$nestingDepth(C) = \max_{v \in V_\sigma} \left\{ \begin{array}{ll} nestingDepth(v) + 1 & \text{if } v \text{ is a compound vertex} \\ 0 & \text{if } v \text{ is a simple vertex} \end{array} \right\}$$

In other words, the start time of a compound is the earliest start time of all its simple vertices (however deeply nested they may be inside its inner subgraph),

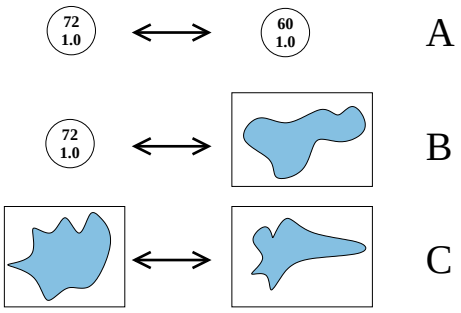


Figure 49: A: Comparing two simple vertices. B: Comparing a simple vertex with a compound vertex and its subgraph. C: Comparing two compound vertices and their subgraphs.

and the end time of the compound is the latest end time of all its contained simple vertices. The nested size of a compound vertex is the number of contained simple vertices, and the nesting depth is the length of the largest string of nestings recursively containing each other in its subgraph.

In summary, replacing edges to and from σ_C with edges to and from C under compound substitution of σ_C with C introduces `WEAKFOLLOW` and `WEAKSIMULTANEOUS` relations. Strong `FOLLOW` and `SIMULTANEOUS` relations are as defined in section 4.2.1. The `WEAKSIMULTANEOUS` relation is not transitive, but the strong one is. Edges are modified with a list of anchor indices for each endpoint so as to allow anchor resolution.

5.2 Extensions of the SimFinder

Having introduced compound vertices and weak edges in the graph, we still need to figure out how these should be interpreted in the similarity measures, view comparators and viewpoints of the SimFinder. Up until now we have only compared simple vertices and done so in terms of their note information. We now need a way to compare arbitrary simple or compound vertices with each other. A simple vertex only contains information on the note it represents, while a compound vertex does not in itself have any note information. Instead it contains a subgraph. The simplest possible subgraph contains only a simple vertex. On the other hand, a (non-sequential) subgraph may be as complex a structure as one can find time to devise, containing hundreds of simple and compound vertices and thousands of strong and weak edges. We thus have three basic comparisons to make (see figure 49):

1. Compare two simple vertices
2. Compare a simple vertex with a compound vertex
3. Compare two compound vertices

The set of sequential subgraphs is contained in the set of non-sequential subgraphs and therefore could be treated under the non-sequential case. But the SimFinder system is built up around these two different kinds of search, which

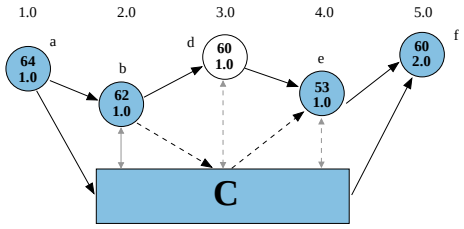


Figure 50: Allowing `WEAKFOLLOW` edges in sequential subgraphs would allow e.g. the coloured subgraph $[a, b, C, e, f]$, although C is sounding simultaneously with b and e. If only `FOLLOW` edges are allowed, the only subgraphs possible are $[a, b, d, e, f]$ and $[a, C, f]$

we shall use separately, and combined, in section 5.4.1. Therefore, we have chosen to extend each search type separately.

5.3 Sequential compound subgraphs

Let's consider first the sequential case. The most important fact about a sequential subgraph is that all vertices in it are connected by `FOLLOW` edges in a string-like fashion – a sequence. Should we allow `WEAKFOLLOW` edges in a sequential subgraph? Including a `WEAKFOLLOW` edge in a sequential subgraph would mean that there is a compound vertex in the subgraph (weak edges are only created when a subgraph is substituted by a compound and are thus always connected to at least one compound vertex) and that, if the `WEAKFOLLOW` edge goes into the compound, the start time of the compound is less than the end time of the preceding vertex in the sequence; or, if the edge goes out of the compound, the end time of the compound is greater than the begin time of the following vertex in the sequence. In other words, part of the compound will be sounding simultaneously with other notes in the sequential subgraph (see figure 50). If such a compound subgraph were unfolded to a simple graph, it would be two sequences of notes that cross in one shared note.

This conflicts with our idea of sequential subgraphs as strings of notes that follow immediately upon each other. The search would incorporate strings of notes possibly with attached branches of notes, and these in turn could have other branches attached etc. We would be matching trees of notes, or even non-sequential subgraphs with cycles, if the edges are regarded as non-directed. Perhaps this would make sense for some purpose, but it is difficult to see the relevance for our search for melodic similarities in and between parts. We have chosen to forbid `WEAKFOLLOW` edges in sequential subgraphs. Thus when a compound vertex appears inside a sequential subgraph, it is connected to preceding and following notes only by `FOLLOW` edges. Subgraph $[a, C, f]$ of figure 50 is an example hereof.

An entire part (i.e. one long string of notes) can then be reduced by substituting substrings of the part by compounds. Compounds may contain other compounds, thus exhibiting a nested structure. This is equivalent to describing the string by way of a grammar, where each compound vertex corresponds to a non-terminal that produces the substring described in its subgraph. Building a

grammar from a music graph using the SimFinder is discussed in section 5.4.4, but we shall use the notion of non-terminal in the comparison of two compound vertices. Compound substitution is done on groups of fragments of the graph that are chosen and grouped for their similarity. In this respect each group is like a family of production rules for a single non-terminal that may produce different, although similar, results. Compounds from the same substitution group therefore share the same non-terminal and in this respect can be thought of as roughly equal, without comparing their inner subgraphs again.

When comparing two sequential subgraphs, two simple vertices may still be compared as before. How a compound vertex should be compared with either a simple or another compound vertex depends on the viewpoint in question. We have made the following choices:

- Absolute pitch: a simple vertex and the entirety of a compound vertex cannot meaningfully be compared according to their absolute pitches. A compound vertex does not have an absolute pitch. We might have chosen the pitch of the first vertex in its subgraph to represent the compound, but of course the first pitch is not always representative for a subgraph as a unit. A counterexample could be a piece where the sequence ends a longer melody, which harmonically is supported by a cadence. The last note then typically would return to the tonic and therefore be structurally more important or representative of the sequence than the first note. Other examples might be given, where a note in the middle of the sequence could be said to be more representative, or where no note can be said to be the most representative of a sequence. On these grounds, we have chosen to simply increase the difference counter whenever a simple vertex and a compound vertex are compared, as though they were maximally different.
  When two compounds are compared, we compare the non-terminal that they represent. If their non-terminals are equal, they are identical (just as if comparing two simple vertices with the same pitch); if not, they're different, and the difference counter is increased. The comparison could also be made on their inner subgraphs, but if the compounds share the same non-terminal, their subgraphs have already been found similar enough by an earlier comparison. Constraining the comparison to the non-terminals therefore is a fair approximation and one that spares an appreciable amount of computing power, especially for the non-sequential similarity measures.

- Pitch interval: the pitch interval viewpoint focuses on the changes from note to note. The transition from a simple vertex to a compound is heard as the change in pitch from the simple vertex' note to the first note in the compound. Similarly, the transition from a compound to a simple vertex is heard not as a transition from all notes in the compound but from its last note to the note represented by the simple vertex. We therefore define the pitch interval along an edge to be the difference in pitch of its two anchors. Thus the first and the last note in a compound still have a role in pitch interval comparison even after they have been abstracted away into a compound vertex. The transition from a compound to a compound then is the transition from the last simple vertex in the first compound to the first simple vertex in the last compound.

- Pitch contour: the pitch contour can be thought of as a more loose, unquantified, version of pitch interval. It therefore does as described above and uses the first and last simple vertex in a compound to represent it when computing the pitch contour for the in-going edge and the out-going edge respectively.
- Pitch class: just as the absolute pitch viewpoint, the pitch class viewpoint compares two vertices. It is unclear what the pitch class of a compound should be, so we regard a simple vertex and a compound vertex as different, and two compound vertices are considered as equal if and only if their non-terminals match.
- Diatonic absolute pitch is also treated like absolute pitch.
- Diatonic interval, Diatonic inversion interval, Diatonic forward interval, and Diatonic pitch contour: these viewpoints all focus on transitions and use the same strategy as the pitch interval and pitch contour viewpoints.
- Absolute length: the length of a compound is defined above in definition 5.3, so a simple vertex and a compound vertex may be compared on length as usual without problems. Similarly with two compounds.
- Length interval and length contour: as there are no problems identifying the length of a compound, these viewpoints also work in the usual manner.

There are now several possibilities for size modifications. We have found it fruitful to use the nested size of subgraphs instead of the size. The nested size counts all simple vertices that may be nested in many layers inside compounds in the subgraph. The size of a similarity statement we take to be the mean nested size of its two subgraphs. With these changes in place, the SimFinder is able to compare nested sequential structures of simple and compound vertices.

5.3.1 Non-sequential compound subgraphs

We now turn to the non-sequential subgraphs, similarity measures and viewpoints. We have found no problems in allowing weak edges in non-sequential subgraphs. This means that definition 4.6 is revised to:

Definition 5.4 *A Non-Sequential Subgraph* $\sigma_1 = (V_1, E_1)$ *of the mothergraph* $G = (V, E)$ *is a connected set of vertices* $V_1 \subseteq V$ *whose connecting edges* $E_1 \subseteq E$ *are of type* `FOLLOW`, `WEAKFOLLOW`, `SIMULTANEOUS`, *or* `WEAKSIMULTANEOUS` *and where*

$\forall\ v_1, v_2 \in V_1 : (\exists\ e \in E : e \in out(v_1) \wedge e \in in(v_2)) \Rightarrow e \in E_1$

Figure 51 shows an example of a non-sequential subgraph containing mostly weak edges. The simple vertices are only connected to the compound via weak edges. The inclusion of weak edges in the non-sequential subgraphs has the following effects.

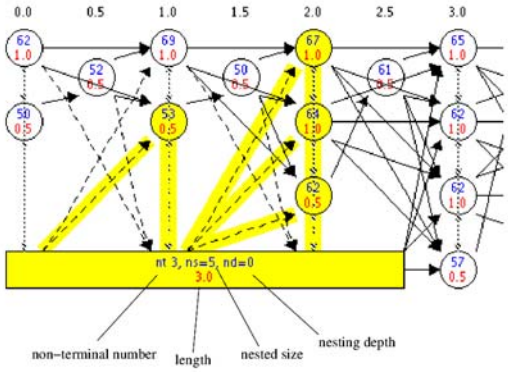


Figure 51: A non-sequential subgraph containing a compound and four simple vertices from the beginning of the Bach chorale "Jesu, meine Freude" (BWV358). The simple vertex on beat 1.0: (53;0.5) is only connected to the compound by weak edges. The other three simple vertices (67;1.0), (64;1.0), and (62;0.5) are internally connected by strong `SIMULTANEOUS` edges, but to the compound they connect only through `WEAKFOLLOW` and `WEAKSIMULTANEOUS` edges.

Applying Grouping Preference Rules to a Compound-graph The grouping structure rules must be modified. We consider a compound vertex to be an already well-grouped entity. This has some relation to GPR 4 that states:

> (Intensification) Where the effect of GPR 2 and GPR 3 are relatively more pronounced, a larger-level group boundary may be placed [LJ83, p. 49].

We use a graduation of how many grouping rules that apply, so that more preferred groupings will be found first. We will also use the compounds as indicators for good further divisions, since they in the first place were found as having a good grouping value.

In addition to the grouping rules already described, we have made a small handful of rules taking care of what to do when some of the vertices in the four note sequences we calculate the grouping value from (remind section 4.3.1 page 72) are compound vertices.

We have two rules: Prefer groups starting/ending with a compound. This is an incarnation of GPR 4 presented. The second rule is more like a prohibition rule like the rest-prohibition already described: Do not prefer groups starting with the sequence (simple, compound, ...) nor groups ending with (..., compound, simple).

This is a choice we have made. This will encourage the SimFinder to extend its subgraphs until they are adjacent to another compound. The intention is to prevent single notes in being left in between two compound nodes, if we can avoid it. That is not a desirable grouping structure, as GPR 1 states: "Strongly avoid groups containing a single event."[LJ83, p. 43] If a note is in between two compounds, it is implicitly bound to be in its own group.

We now turn to describing the additions to our search-system needed to take care of compound vertices.

Vertex comparison Compound vertices have in-, out-, and sim-edges just as simple vertices, so there is no problem in comparing vertices on number of edges of these kinds. In addition to the single-vertex viewpoints that look at the number of `FOLLOW` in- and out-edges, and the number of `SIMULTANEOUS` edges, we now also have versions of these viewpoints that count the corresponding number of weak edges:

- The number of in-edges of type `WEAKFOLLOW` of v
- The number of out-edges of type `WEAKFOLLOW` of v
- The number of in-edges of type `WEAKSIMULTANEOUS` of v (which is the same as the number of `WEAKSIMULTANEOUS` out-edges, since both ways are represented in the graph)

The weights of the similarity matrices according to the different viewpoints also need to be changed. We set the weight of the pitch and length viewpoints each to $weight = 5$, while the now *six* topological viewpoints each are given $weight = 1$.

The absolute pitch vertex comparison can still be done when comparing two simple vertices. Simple-compound comparisons we have chosen to always evaluate to 1.0 (the worst possible value), because it is unclear in what sense a compound could be equal to a simple vertex with respect to pitch. The other choice concerns compound-compound comparisons: here we evaluate them to be equal (value=0.0) if their non-terminals are the same and completely different (value=1.0) otherwise. An improvement of this could be to additionally check if there is a match between the viewpoints under which each compound was found to be a member of its non-terminal.

Absolute length of compounds is easily evaluated, so there are no changes there.

Edge comparison Edge comparison is affected by the introduction of weak edges. We still keep the requirement that edges be of the same kind to be comparable. This means that a `FOLLOW` and a `WEAKFOLLOW` edge are always maximally different (value=1.0). If they *are* of the same type, we now compare edges on the basis of their anchors, not their endpoints. This solves the pitch-related problems we have with vertex comparison and with the absolute sequential viewpoints, that compounds have no pitch as such. Each anchor is a simple vertex with a pitch, which represents the compound when seen from the edge whose anchor it is. An edge is "entering" the compound, so to speak, at a particular point, reaching into it, and being tied to it, at its anchor. Therefore the anchor's pitch is relevant to the edge.

The one exception in edge comparison is the absolute length viewpoint, where it is still the lengths of the endpoints, not the anchors, that we compare.

Comparison of bags of subgraphs We would still like to regard non-sequential subgraphs as consisting of solely sequential information. The subgraphs can now contain compound vertices, but this is not a big change in the algorithm. We can still pick out all vertices and edges in the graph and compare these to each other, and we can still find all sequential subgraphs of the non-sequential graph. Only the type of the vertices have changed.

We need only to change our encoding scheme. Encoding simple vertices (SV) proceeds as before. When encoding (comparing) compound vertices (CV), we encode the production rule number and the measure under which the compound was found to be similar to its corresponding compound. This is the information that determine the character of the compound. Encoding edges proceeds also almost as before. Edges can now point from simple vertices to compounds, but the edge is always anchored to a simple vertex inside the compound. This is valuable information which we will use. Since we can find the simple vertices on any edge, we can take a sequential view of it no matter which vertices it connects. So in the encoding of edges, we encode some information about each of the vertices and also the sequential view of the edge. Strong and weak edges are encoded similarly. Thus for two non-sequential subgraphs to be judged equal in the $BagIR_{SS(n),view}$ setting alone, they must consist of equal sequential subgraphs (of size n) where equal means that they have the same vertices at the same places. Here is the extended encoding scheme of vertices and edges:

encode absolute	Vertex	$enc(SV)_{DAP}$	$\rightarrow$	$name_v\_octave_v\_accidentals_v\_duration_v$
	Vertex	$enc(CV)_{DAP}$	$\rightarrow$	$nonterminalNumber_{CV} + measure_{CV}$
	Edge	$enc(v_1, v_2)_{DAP}$	$\rightarrow$	$enc(v_1)_{DAP}\_enc(v_2)_{DAP}$
encode relative	Vertex	$enc(SV)_{vp\in\{PI,DI,DFI\}}$	$\rightarrow$	$duration_v$
	Vertex	$enc(CV)_{vp\in\{PI,DI,DFI\}}$	$\rightarrow$	$nonterminalNumber_{CV} + measure_{CV}$
	Edge	$enc(SV_1, SV_2)_{view\in\{PI,DI,DFI\}}$	$\rightarrow$	$enc(SV_1)_{view}\_view(SV_1, SV_2)\_enc(SV_2)_{view}$
	Edge	$enc(SV, CV)_{view\in\{PI,DI,DFI\}}$	$\rightarrow$	$enc(SV)_{view}\_view(SV, CV.anchor)\_enc(CV)_{view}$
	Edge	$enc(CV, SV)_{view\in\{PI,DI,DFI\}}$	$\rightarrow$	$enc(CV)_{view}\_view(CV.anchor, SV)\_enc(SV)_{view}$
	Edge	$enc(CV_1, CV_2)_{view\in\{PI,DI,DFI\}}$	$\rightarrow$	$enc(CV_1)_{view}\_view(CV_1.anchor, CV_2.anchor)\_enc(CV_2)_{view}$

When encoding a sequential graphs we take the same approach as before:

$enc(\sigma = (v_1, v_2, \ldots, v_n))_{view\in\{DAP,PI,DI,DFI\}}$	$\rightarrow$	$enc(v_1, v_2)_{view};$ $enc(v_2, v_3)_{view};$ $\ldots;$ $enc(v_{n-1}, v_n)_{view};$

5.4 Segmentation

We have described the graph representation and the SimFinder with all its viewpoints. We now want to start using multiple runs of the SimFinder to discover and categorise different kinds of similarities inside a piece. By looking at individual similarity patterns as delimited objects, we can find further similarities encompassing these patterns. This is done by substituting similar subgraphs of the mothergraph with compound vertices of the same non-terminal. A subsequent SimFinder run on the graph now needs to also take account of compound vertices in the graph and compare these with other vertices on equal terms with simple vertices – as described above. This subsequent SimFinder run again locates similarities which it substitutes with compound vertices, and so forth. The segmentation of the piece by substitution of subgraphs with compound vertices, and in several layers, gives a hierarchical structure of nested abstractions that are essentially production rules in a graph grammar.

5.4.1 Recursive segmentation algorithm

The program that uses the SimFinder to produce a segmentation is called the Similarity Segmenter, or the SimSegmenter for short. It can instruct the SimFinder to use different similarity measures in different runs and can thus use the specific to general ranking of the viewpoints that we described back in section 4.3.1, p.67. The idea is similar to the algorithm of Smaill et al. (see 3.2.2), where *motives* and all *derivations* of them are located and labelled according to the measure that deemed them to be similar. The label describes the transformation that can turn the derivation into its originating motif. Recall that patterns which are related by a transformation function can be thought of as being in some music-semantic relation to each other. In our case, it is a music-semantic relation that is a paradigmatic relation of *similarity*, akin to the linguistic relation of *synonymy*. A motif and its derivation can be argued to be musically synonymous, because they are instantiations of the same musical idea. The viewpoint that was used to find the similarity between motif and derivation designates more specifically which relation they have to each other (e.g. the DI viewpoint means that they are diatonic transposes of each other). In linguistics, *hyponymy* is a relation of general-to-specific ordering, which could perhaps also be argued to exist between different levels of elaboration/simplification of motives. We have not dug deeper into this, since we have restricted ourselves to repetition and simple transformations. *Meronymy* designates whole-part relations which are implicit in the grammatical structure of a context-free grammar. The analogy with the last linguistic example we have given of paradigmatic relations, *antonymy* (opposite meaning), is less clear. We could say that two phrases that are pitch interval inversions of each other are opposites, although they are also in a sense very much alike. We think that a systematic account of such relations would be a useful basis for a more well-structured segmentation than we have achieved here by using mostly similarity relations among subgraphs. We find similar subgraphs, substitute them with compounds, and label the compounds with the name of a non-terminal (and also the similarity measure used), so that a compound's meronymy relation to its contained subgraph belongs to a certain non-terminal, or class. Common to the subgraphs produced by compounds of the same non-terminal is a certain similarity relationship which is dependent on the viewpoint with which we located the matches.

Algorithm SimSegment The idea of the algorithm is first to find one pattern that occurs twice. This similarity should be seen from as strong a viewpoint as possible, so we begin with the most specific viewpoint[40]. Then we find all occurrences of these two patterns when seen from the successively more general viewpoints; e.g. if there are no more direct transpositions (i.e., with the PitchInterval viewpoint) of some pattern p, there could perhaps still be patterns that had the same *contour* as p, i.e. according to the PitchContour viewpoint, which is a more general viewpoint than PitchInterval. By ordering the viewpoints and by searching with the strongest first, we can categorise the occurrences with as strong a viewpoint as possible. The compound node that replaces a pattern when we find it is labelled with the similarity measure whose

[40]It is the similarity measure we are changing between SimFinder runs, but the measures we have defined have only one pitch viewpoint each, so in the area of pitch, they can still be ordered by the specific-to-general ranking.

viewpoint combination was used to locate it; and each found pattern and all of its repetitions and transformed repetitions belong to the same non-terminal, which we will use in the graph grammar. We therefore know what kind of relation a given pattern has to the original motif of the same non-terminal. In this way, the non-terminal categorises all transformations and shades of a given motif, and labels them with the transformation – the paradigmatic relation – that relates them to the original motif.

One problem is that there is no guarantee that the musically most important version of a motif is found first. Then it is a less important derivation that is considered as the original motif, while the original one is labelled as e.g. a 'transposed version' of it. We don't consider this a real problem however, since the relationships still hold, no matter which of the two patterns in a *transposition* relation we take to be the original. If the importance of different versions can be ranked by some algorithm, then such an algorithm may be applied in the SimSegmenter after all occurrences in a family of motives/derivations are found. And in a simple way there is some weighting built into the SimSegmenter: we can consider the version occurring most frequently under the strongest possible viewpoint to be the most important. If a pattern occurs more frequently, there is a greater chance that the SimFinder will stumble upon it, so there *is* a primitive importance weighting built into the search. E.g. in the `nonSeqEdge` search example in section 5.4.4, p.131, there is a greater probability of finding the pattern of the very frequently occurring non-terminal NT5 than of finding other recurring patterns seen only two or three times.

When we are done with one pattern and all of its occurrences according to the successively more general viewpoints, we move on to the next. The search is repeated until no more relevant similarities are found. When that happens, hopefully we have found, categorised, and substituted the most significant patterns.

The algorithm looks like this:

```
double limit
int nonterminalNumber=-1

for(SimMeasure sm_i = 1..k){
    nonterminalNumber++
    //search for new repeated patterns
    while(there are more repeated patterns to be found){
        let s_1 and s_2 be the most similar subgraphs of a SimFinder run with sm_i
        //search for more occurrences
        if( SimMeasure(s_1,s_2) is within the similarity limit){
            substitute s1 and s2 in the graph with compound nodes with type nonterminalNumber
            for(SimMeasure sm_j = i..k){
                while(there are more occurrences of one of s_1 and s_2 to be found){
                    let s_i be the graph found in an Occurrence search for s_1 with sm_j
                    if( SimMeasure(s_1,s_i) < limit){
                        substitute s_i in the graph with a compound node with type nonterminalNumber
                    }
                    let s_i be the graph found in an Occurrence search for s_2 with sm_j
                    if( SimMeasure(s_2,s_i) < limit){
                        substitute s_i in the graph with a compound node with type nonterminalNumber
                    }
                }
            }//advance to next SimMeasure in the occurrence search
        }//no more patterns to be found with the SimMeasure
    }//advance to next SimMeasure
```

}

When an occurrence search is started, the used viewpoint is always the same or weaker than the viewpoint that located the original pattern. This means that when we have given up looking for new motives with viewpoint vp_1 (e.g. AbsolutePitch), and moved on to the search of motives using vp_2 (e.g. PitchInterval), it is impossible to find similarities according to a stronger viewpoint than vp_2. The rationale in this is that we ought to have already found all of these stronger similarities – that was the termination condition for the search using vp_1. In the algorithm above, the termination condition is stated as "while(there are more repeated patterns to be found)". In practice, we have imposed a limit value, or threshold, on the fitness of the best found similarity statement. The value of *limit* depends on the similarity measure used; it specifies how bad the best found match should be before we give up searching with the current viewpoint[41] We have used the following limits:

`seqDCGrp`	2.0
`seqMD`	0.0001
`vertexComparison`	0.05
`edgeComparison`	0.06
`nonSeqBagIR`	1.7

E.g. when the similarity value of the best found match using `edgeComparison_APAL` rises above 0.06, we change to similarity measure `edgeComparison_PIAL`.

If we didn't find and substitute all occurrences of all transformations of a motif A before searching for other motives, we might locate fragments of these occurrences in other motives and substitute them. It would thus be impossible to find the connection with motif A later on, because part of A would be substituted away into some compound vertex. On the other hand, by substituting all occurrences at once, the inserted compounds can constitute new patterns in conjunction with other fragments of the graph. Since the algorithm builds the hierarchic structure in a bottom-up fashion, it is advantageous to begin with patterns that are not too large. Large repeated patterns are most often extensions of smaller repeated patterns, which may in fact be more significant as a pattern, so we want to substitute these with their own compound before substituting the encompassing, larger pattern. Also, on the practical side, the SimFinder is exceedingly slow as the size of subgraphs rises, and setting a maximum size on patterns, and thus putting an artificial upper bound on the running time of one SimFinder run, is sometimes a necessity. So there is also a practical reason to first locate and group together smaller similarities that are repeated inside larger patterns.

5.4.2 The SimSegmenter's use of the SimFinder

We see now why even the smallest overlap in similarity statements is annoying in the segmentation algorithm. The pairwise substitution of similar subgraphs with compound vertices cannot be done – or would need some special scheme to be carried through – if the two subgraphs overlapped. Having substituted one of the subgraphs, all of its vertices have in fact been removed from the

[41] Or rather, the current similarity measure, as it really is.

mother graph, so what do we do with the missing vertices (those that were in the overlap) of the next subgraph? As mentioned in section 4.3.4, this is the ultimate reason for simply discarding overlapping similarity statements in the SimFinder by setting their fitness rating to an astronomical number instead of the value given by applying the similarity measure in use.

As a GA, the SimFinder is a stochastic search method that does not give us an exhaustive search, so there is no guarantee that all pattern occurrences have been located when the SimFinder is stopped. This can be a problem. The practical solution has been to estimate the number of generations necessary to find a useful match and fix a maximum number of generations that allows the GA to find matches if there are any. If no matches with better similarity than the *limit* value are found after the maximum number of generations have elapsed, the search is considered to be a dead end, and the similarity measure is changed to a more general one. As described in the appendices on GA tuning, we have also tried tuning the mutation settings to allow the SimFinder to tumble around in the search space even while narrowing in on the best match it has found. Even so, the entire SimFinder population often does narrow in on a single similarity it has located and then exploits it by trying to extend the match to bigger subgraphs. But the GA is also an "anytime algorithm", able to deliver intermediate results whenever we need them. This could be exploited in an extended search program that controls a host of SimFinders, each searching its own corner of the search space. The search might benefit from a periodic exchange of genetic material from one SimFinder to another. Using multiple concurrent SimFinders would allow us to locate several similarities at once instead of narrowing in on only one, as the SimFinder does. From an implementation point of view, this would be very easy to test, because the SimFinder is described in its own class and can thus be instantiated in any number. But it would also be a little more expensive in computation time.

5.4.3 Discussion and relatives of the SimSegment algorithm

In section 3.2.2, we described a segmentation algorithm used by Smaill et al. (described in [SWH93]). Hopefully it is now clear that the SimSegment algorithm is a close relative; the segmentation is done also on the basis of found similarities. Similarities are located according to a number of similarity measures that are ranked from strong to weak. Strong similarities are found first, and weaker ones are labelled with the measure that located them.

Now the differences: while the Smaill et al. algorithm works on *streams*, i.e. sequences of notes, the SimFinder is able to locate both sequential and non-sequential similarities, in monophonic or non-monophonic music. The SimSegmenter does not simply remove occurrences of motives as they are found. It replaces them with compound vertices which may in turn be part of other similarities. This allows us to build a hierarchical structure instead of the flat segmentation output by the Smaill et al. algorithm. This could be remedied by running their algorithm again on its own output to produce another segmentation level, but in the article [SWH93] they have not described how to compare streams containing *constituents* - the structure used to group notes together.

We still have a few pending questions from Roger Dannenberg (see section 5). As to nesting, yes, the whole idea of compounds in the SimSegmenter algorithm is to construct a nested structure. Shallowness of compounds is more

dependent on implementation. In our implementation of the SimFinder system, each compound vertex contains a *deep clone* of the subgraph it is substituted for, i.e. a complete cloning of all objects in that structure, and all these clones are 'owned' by the compound. Thus the only thing that connects it to the other occurrences of the pattern it represents, is their common non-terminal in the graph grammar. Editing the subgraph of a compound would not change anything in the patterns, or subgraphs, of its sibling compounds, but it would change the relationships of the compound to its surroundings, possibly deleting or inserting new strong or weak edges to and from the compound. So the user cannot control (in the sense 'to manually edit') how close a connection the occurrence of the motif or pattern has to the global, idealised, description of the motive. There *is* no such global description of motives in a SimSegmented piece, only the individual occurrences. Anyway, the SimFinder system does not allow editing of the subgraphs of compound vertices. The analysis yielded by a SimSegmenter run points out similarities between different compounds in terms of their shared non-terminal *and*, more specifically, in terms of the similarity measure under which they were found to be similar. If subgraphs were edited by the user, the claimed similarity may no longer hold. So Dannenberg is right to point out that the 'one global pattern - many local occurrences' scheme "would quickly lead to many interesting problems" [Dan93, p.26].

5.4.4 Building a graph grammar

In grammatical terms, a compound substitution corresponds to a production rule. When the first pair of similar patterns of a motif is found, both patterns are substituted, giving two production rules:

$$c_1\ X\ c_2 \rightarrow c_1\ motif_1\ c_2$$
$$c_3\ X\ c_4 \rightarrow c_3\ motif_2\ c_4$$

where X is the non-terminal that links the motives and all their derivations together, and the c's are the context of each production. If other derivations of the two motives are located, each of them is also added as a production rule:

$$c_5\ X\ c_6 \rightarrow c_5\ derivation_i\ c_6$$

These examples belong in a context sensitive grammar. But we are building a context sensitive *graph grammar*. This means that the context can be any number of music vertices connected to the non-terminal by music edges.

An example of production rules in a graph grammar Let's look at the chorale that we know so well by now (BWV358). The following graph production rules are taken from a simple SimSegmenter test run using the `nonSeqVertex_PLIOS` measure on a partwise graph built from a midi file. The substitutions are not particularly smart, but they serve to illustrate the mechanism and the connection between compounds, non-terminals and graph production rules. The parameter settings of the SimSegmenter are shown in appendix C.1.

This particular test run resulted in 28 graph production rules concerning 13 non-terminals. Every match found by a SimFinder run inside the SimSegmenter yields two production rules[42], and additional occurrences of the motif are added

[42] One for each of the matching subgraphs, or motives.

as production rules for derivations. There were thus only $28 - (13 \times 2) = 2$ derivations – a little disappointing, but of course the `nonSeqVertex_PLIOS` measure finds only matches on absolute pitch and length – it does use relative views between vertices. All in all, the graph was not at all well segmented, and much more was left unsegmented in the mother graph. Please note that the "time rulers" are shown at the top of each subgraph only as a practical means to tell us where the fragments were found in the mother graph. The matching of subgraphs using `nonSeqVertex_PLIOS` is done *purely* using the pitch, length, and topological information present in each vertex and thus is indifferent to the absolute times indicated in the rulers – only the structure of the subgraph counts.

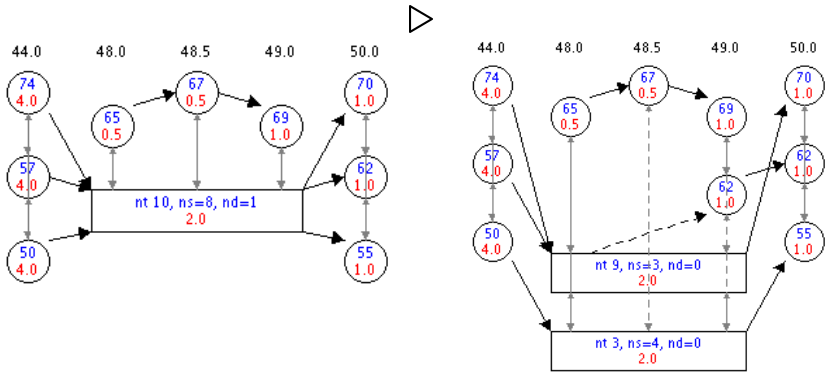


Figure 52: Production rule PR23 for non-terminal nt 10 (nonSeqVertex_PLIOS).

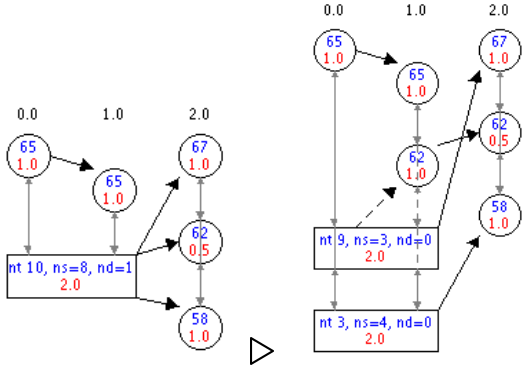


Figure 53: Production rule PR24 for non-terminal nt 10 (nonSeqVertex_PLIOS).

For non-terminal nt 10, no derivations were found, so we have only the two productions of the motif. These are shown in figures 52 and 53. The two rules tell us that when an nt 10 compound occurs in the context shown on the left-hand side, the context and the compound can be substituted by the right-hand side, which comprises the context *and* additionally two compound vertices of nt 9 and nt 3, and a simple vertex (62,1.0). What next happens

is determined by the production rules for nt 9 and nt 3. One rule for each is given in appendix C.1, and following these will produce a simple subgraph (i.e. containing no compounds) that is a subgraph of the original music graph created from the file. It is the rules that will apply to the situation resulting from PR24 that are given in the appendix.

Please notice that it is not necessary that our graph grammar be context sensitive. As we have implemented it, a compound vertex can be said to produce its inner subgraph irrespective of the context. But we would have to specify formally how substitution of edges between compound/subgraph and the surroundings should happen. We would have to formalise how the scheme described in section 5.1 should behave in all circumstances. In practice, in the algorithm that effects compound substitution in a mother graph, we have chosen to use anchors of edges to point to simple vertices inside compounds, thus facilitating the reverse substitution that a production rule corresponds to. But the reverse substitution could be effected simply from the knowledge of start times of all involved vertices. This has not been implemented. Instead, we have chosen to show the production rules as context sensitive, because the substituted edges can then be shown explicitly in the production, but we could have built a context free graph grammar. We discuss the generalisation of graph context below on p.133.

Building structure from simpler building blocks The following example is taken from a test run using the `nonSeqEdge_DAPAL`, `nonSeqEdge_DIAL`, and `nonSeqEdge_DFIAL` measures. The piece was the Bach fugue part of the *Prelude and Fugue in C minor* (BWV 847).

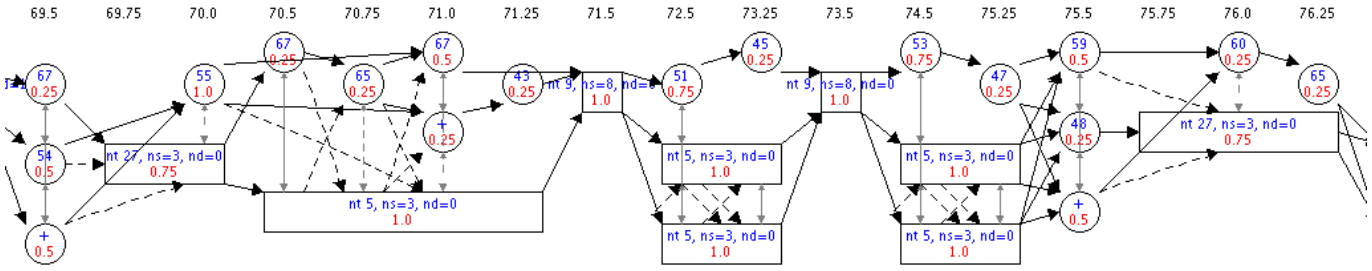


Figure 54: Example from the mothergraph after segmentation using `nonSeqEdge_DAPAL`, `nonSeqEdge_DIAL`, and `nonSeqEdge_DFIAL`.

This SimSegmenter run used a non-partwise graph of the fugue, which makes it more difficult to segment it in a meaningful way. But if we begin with some very small building blocks, we may find it easier. As seen in figure 54, the non-terminal nt 5 occurs in many places: twice on 72.5, twice on 74.5 and once on 70.5. These are all variations of the same three-note fragment, which starts the theme of this fugue. Figure 55 shows what the first occurrence on 70.5 contains. In fact, the sequence [(67,0.25), (65,0.25), (67,0.5)] beginning on 70.5 is also an instance of the nt 5 pattern (being equal to the nt 5 subgraph in figure 55 under the `nonSeqEdge_DIAL` measure), and it is clearly a mistake that the SimSegmenter hasn't located and substituted it along with all the other nt 5 compounds. Then we would have had three pairs of simultaneous nt 5 variations

separated by nt 9's in figure 54. In a more successful run, a new non-terminal nt x could have grouped the recurring [nt 9,<nt 5,nt 5>] structure.

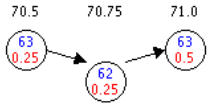


Figure 55: The subgraph contained in the nt 5 occurring at 70.5.

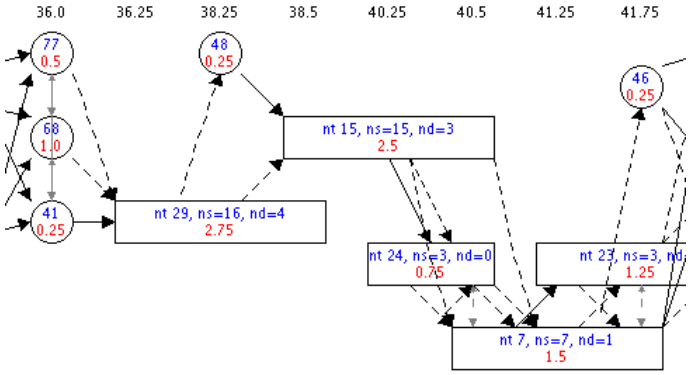


Figure 56: Part of the mothergraph after segmentation using edge comparison on DAP, DI and DFI.

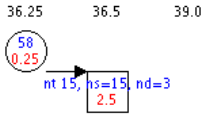


Figure 57: The subgraph contained in the nt 29 beginning at 36.25.

But there are other examples of nesting in this run. In figure 56, the recurring structures are not immediately visible. But upon closer inspection, we see that an nt 29 is actually an nt 15 with just one additional note (see figure 57). The nesting is even deeper: nt 15 contains an nt 14, which contains an nt 7, which contains the most recurrent pattern, nt 5, for this analysis. For the fragment shown in figure 56, we can therefore visualise the analysis as a derivation tree, shown in figure 58. The subgraphs inside the compounds contain simple vertices in addition to the compound vertices just mentioned, but we haven't drawn them in the derivation tree. The subgraphs for these compounds are given in appendix C.2.

These two examples should give an idea of the SimSegmenter's building of hierarchical structures, although the the derivation tree is more like a sequence of very deep nestings than a real tree structure combining more compounds on

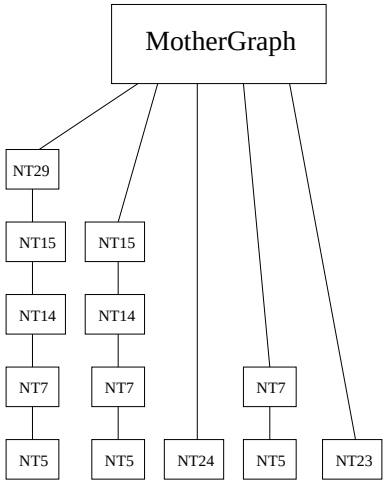


Figure 58: A derivation tree showing the nesting of compounds in the fugue fragment of figure 56. The individual notes (simple vertices) which constitute the musical surface are not shown here, only the nesting relationship between compounds. Each compound vertex also contains one or more simple vertices.

a single derivation level. As mentioned, this run used a non-partwise graph, and no grouping viewpoints that could help the segmentation process. We have had better results using partwise graphs and similarity measures with grouping viewpoints incorporated. This is presented in more detail in section 5.6.

Using a graph grammar generatively We have not yet experimented with the generative capabilities of the induced graph grammars. The purpose would of course be the sheer fun of seeing the mechanism in action, but more importantly, it would be an important step in the validation of the SimSegmenter. How well a grammar generalises to describe valid musical structures is a good indication of its quality.

How would we generate a piece using the set of production rules output by the SimSegmenter? If the rules were context free, we would need only search for non-terminals (compounds) and then pick a production involving that non-terminal. The substitution mechanism would be specified so that the edge substitution is feasible under all circumstances. As mentioned, we have not implemented this. Instead we have context sensitive rules; to apply a context sensitive rule, we have to locate a non-terminal with a certain context in the graph. Such a search can be accomplished using the SimFinder, since a context is just a subgraph. For a non-terminal that has m production rules associated with it, we would have to search for m different contexts. Here it could be useful to generalise the m different contexts, in order to be able to search for all variations of it in one search. The generalisation would consist in finding "the transformation under which the contexts are equal". Does this sound familiar? We happen to have a solution in our viewpoint system. It is possible that some of the contexts could be described as equal under some viewpoint, so that the search for matching contexts could be effected using that particular viewpoint

in the SimFinder search.

Generalisation of context also raises the question of how much context we really need to include. At present, the context of a compound is the set of all vertices connected with the compound by an edge. It might be that it is appropriate to include more than this, e.g. the set of all vertices whose distance to the compound (in the number of edges) is less than n. But we think it is more likely that context depends on musical constraints too. A highly relevant context could be the underlying harmonic context. Also, it seems likely that the size of the relevant context depends on the size of the compound. A sentence like:

> He looked away and said: Because I love you.

is much more sensible than a sentence where "Because I love you" is replaced by a 500 page speech. In the same way, a subgraph of three notes is probably not the best context to an entire movement, and conversely, an entire movement is too much of a context for three simple notes. If we have already abstracted a movement into a compound vertex, a suitable context would be building blocks at the same level: movements, or perhaps *expositions* and *developments* at the next lower level, but certainly not a subgraph of three simple vertices. Here is a genuine problem with our approach: we have based the segmentation almost purely on paradigmatic relationships of similarity, without considering syntagmatic relationships governing the token-to-token relationships in the grammar (the grouping viewpoint being the only exception). An explicit account of syntagmatic relationships could give us a better idea of the contexts in which compounds occur, e.g. that a compound C always starts on the tonic after a certain cadence.

An interesting experiment would be to analyse several pieces at once to build a common grammar. This would be easily implemented in the SimFinder: it amounts to loading two music files into a common graph, without connecting them with each other. The SimFinder would then search in an unconnected graph consisting of two internally normally connected subgraphs, each describing a piece. Similarities could be found in and among the pieces, and a grammar describing stylistic traits common to the pieces would be built. The last thing to do would be to separate the start production into several possibilities – the islands in the common graph representing each piece.

5.5 Running the SimSegmenter on two pieces of J. S. Bach

We have used two pieces of J. S. Bach as a test for our similarity based analysis: One homophonic piece – a chorale *Jesu meine Freude* (BWV 358) and one polyphonic piece – a fugue *Fugue in C minor* (BWV 847 from WTK 1). The pieces are thus of very different compositional natures. We have conducted three experiments:

- Segmenting *Jesu meine Freude* as a partwise graph in a sequential search.
- Segmenting *Fugue in C minor* as a partwise graph in a sequential search.
- Segmenting *Jesu meine Freude* as a non-partwise graph in a non-sequential search.

We have chosen to let the nature of the pieces inflict on the way we set up the experiment so that the fugue is analysed sequentially and, but the chorale also non-sequentially. Because of its polyphonic nature the fugue begs to be analysed in a sequential search in a partwise graph. When searching in non-partwise graphs, the computation task becomes rather heavy, so we were forced to impose a maximum size on the subgraphs handled by the SimFinder. The pieces we have chosen are otherwise as partwise graphs fairly computable on our computers. The SimFinder is however not able able to handle an entire movement from a symphony since it uses too much memory. We must optimise the system at a later time.

Unfortunately we have not been experimenting with music in between the two types (homophonic and polyphonic), as for example a piano piece (which is hard to find in the MuseData format), where we could possibly benefit from the way our non-partwise graph representation handles both chords and melodies as the same thing.

The sequential similarity measures used in the two sequential experiments have been three of the five measures relying on the difference count (DC) view comparators and they all include the grouping preference rules. They are all based on the diatonic information: `seqDC_DAPALGrp` (uses diatonic absolute pitch, absolute length viewpoints and grouping), `seqDC_DIALGrp` (diatonic interval, absolute length and grouping), and `seqDC_DFIALGrp` (diatonic forward interval, absolute length and grouping) (see section 4.3.3).

Setting the stopping criteria The SimSegmenter is equipped with a limit value. All similarities found must have a fitness below this limit. When a SimFinder is not able to find anything below the limit, the best similarity statement found is not good enough and is discarded.

Since we are using the difference count approach in our search, we can see from the fitness of the similarity statement if a mismatch in pitch occur. By setting the limit of worst acceptable fitness to a value that allows zero errors, we can find only exact matches of pitches (according to the viewpoint used). Let's give an example: Remember that the seqDC are calculated like this (in the DI case):
$\texttt{seqDC\_DIALGrp}(s_1, s_2) = di + \frac{len}{1000} + (1 - \frac{size(s_1)}{100}) + group$. Setting the limit to 2.0 gives a rather good guarantee of finding sequences with no mismatches: the size value of the subgraphs $(1 - \frac{size(s_1)}{100})$ is in practice (in our experiments) roughly between 0.5 and close to 1.0. The difference in durations (variable len) are given almost no importance. The grouping value is in practice from γ^2 to 1.0. If γ is not too small, this will also give a value not too far from 1.0. So the overall fitness is with no mismatches ($di = 0$) between 1.0 and 2.0. The DI difference count will have to be 0 to be below the limit. This kind of reasoning is of course a little risky, but our experiments have shown that this value works in practice at least on our problems. The tuning of the variables in a GA is always connected with these kinds of choices.

Since we do not allow inexact matches (in pitch) in these three experiments, we can use any of the correct matched subgraphs to search for more occurrences, so we would like to save the time looking for occurrences of them both (since they are equal), but instead only search for occurrences of one of them.

The non-sequential similarity measures used in the non-sequential search are

based on the intersection ratio of vertices and edges (BagIR) idea: `nonSeqIR_DiatonicAbsPitch_Grp` and `nonSeqIR_DiatonicInt_Grp`(see section 4.4.4). They are thus also all based on the diatonic information.

We again set the limit to try to make sure that no mismatches occur. The fitness of the `nonSeqIR` suite of similarity measures depends on the sum of Intersection Ratios, the size of the subgraphs and the grouping fitness (`nonSeqIR_DiatonicInt_Grp` $= bagIRTrie_{DI}(\sigma_1, \sigma_2) + grouping(\sigma_1, \sigma_2) + (1 - \frac{Size(\sigma_1)}{100.0})$). So the size and grouping values are calculated in a way rather similar to the case of the sequential subgraphs, and thus yields the same kind of values.

This leaves a little slack to the error of the intersection ratios from the different kinds of encodings, so we cannot be as sure as in the sequential case, that no mismatches occur. But as one non-matching vertex in two subgraphs is likely to affect the value of more than one of the different encodings that the graphs are tested for intersection with, this mismatch will generate a not so insignificant error value, and will thus influence somewhat to the overall fitness. The limit value 1.7 has proven to be useful in the non-sequential case.

5.5.1 Partwise sequential segmentation of *Jesu meine Freude*

Segmenting the chorale should reveal that there is an exact repetition of the first three phrases. The chorale has the form: $ABCABCDEA'$. The A'-part has the same melody as the first two A sections, but some of the underlying harmonies have changed. So the best we can hope to find is to find three occurrences of the melody in the A parts. Furthermore we should find the exact repetition of the first 6 bars. It should be mentioned that the soprano and bass starts on the same notes in all three phrases, so it is possible to find similarities going across all three phrases.

We set up the SimSegmenter run for this task with these parameters:

Population size	100
Generations	500
Crossover	0.01
Mutation	0.5
Fresh Blood Chance	0.1
Initial size	5
Max size	N/A
Similarity measure	`SeqDC_DAPALGrp`
	`SeqDC_DIALGrp`
	`SeqDC_DFIALGrp`
Limit	2.0
γ	0.7

Figure 59 shows the beginning of the resulting graph (the full graph is printed on page 172). The beginning two repetitions have in each voice been reduced to one compound vertex each. Let's look inside the compounds. The score on page 171 in the appendix shows the nesting structure of the chorale. The SimSegmenter in this run split the chorale into 11 different kinds of similarities of varying sizes. We will in this section refer to compounds by their given nonterminal number (nt). Notice the nt 3. It is the six-note melodic phrase, which appears three times as mentioned. The repeated phrases were found as expected. Otherwise only small bits of similarities (nt 8 and nt 9) are found in the last 7 bars.

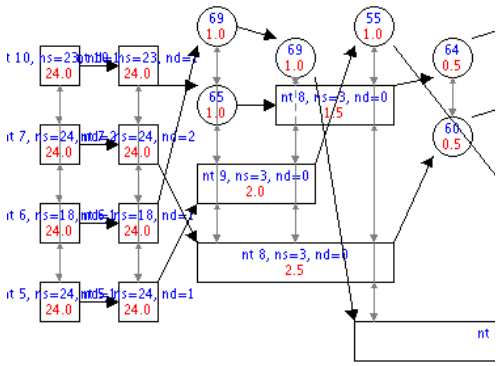


Figure 59: The start of the segmented chorale. Each voice starts with two similar compound vertices.

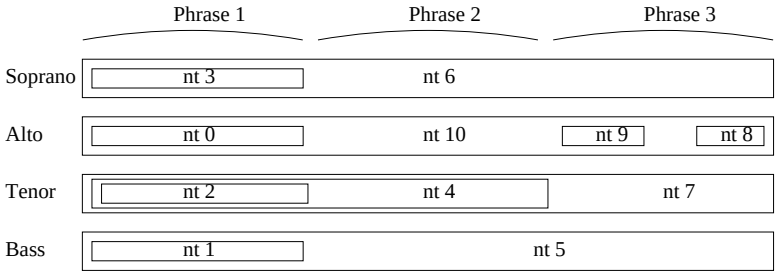


Figure 60: The first six bars and the repetition of them were segmented equally.

The grouping rules help the SimFinder to stretch the phrases from fermata to fermata. This has helped to find some of the phrases in the repeated part. Figure 60 shows the segmentation of the repeated 6 bars in a more simplified way. Most well grouped is the tenor voice, since all the natural phrases are suggested in it. Since the SimFinder prefers larger subgraphs to smaller, it has included the first compound found in the tenor (nt 2) into the next (nt 4) and again that one into the last (nt 7): (((2)4)7). A more equal partition of the phrases would have been ((2)(4)(7)). But the effect of the grouping rules is clearly present.

5.5.2 Partwise sequential segmentation of *Fugue in C minor*

We have run the SimSegmenter algorithm on the three part fugue in C minor from WTK 1. We have set the grouping value rather high, giving not so strict boundaries.

Population size	100
Generations	500
Crossover	0.05
Mutation	0.5
Fresh Blood Chance	0.1
Initial size	5
Max size	N/A
Similarity measure	`SeqDC_DAPALGrp`
	`SeqDC_DIALGrp`
	`SeqDC_DFIALGrp`
Limit	2.0
γ	0.97

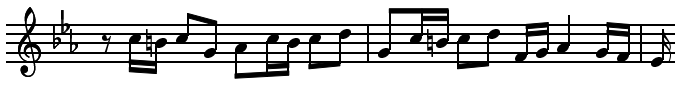
This results in larger sequences on the lowest level of the hierarchy, and thus a rather flat hierarchy since there are no larger scale repetitions in the piece. The themes however can sometimes be put together of smaller motifs or phrases, which we also will give an example of.

The run resulted in 25 different similarities found – from size 3 to 40 when counting simple vertices. We have decided to stop when reaching graphs of size 2. We will go through the main events found in the fugue here, but a full score with indications of all substituted compounds is found on page 173 and page 174 in the appendix. Unmatched notes are marked with an 'x'. There are not many of those. The resulting graph is found on page 175.

An error has unfortunately occurred in this run of the segmentation algorithm. This has the effect that the order in which similarity measures were used for occurrence search has not been as described. The result is that some similarities could have been categorised with other viewpoints than they actually are. For example: some DAP similarities are labelled instead as DFI, so the labelling we found is not the strongest possible. We will in this section denote the parallel passages with the labels the algorithm gave them. The matches are *not* incorrect, but some of them could have been labelled stronger.

The fugue consists basically of a sequence of expositions of a theme, interweaved with some other thematic material. The main theme comes in two variations: the first is traditionally called *dux*. This is the theme in its purest form. The second is a harmonic imitation of the dux and is called *comes*. It begins one fifth above dux, so in order establish the key of the dux, some of the jumps have changed in the comes and hence the term harmonic imitation. The rhythm of the two themes is equal, and so is the contour.

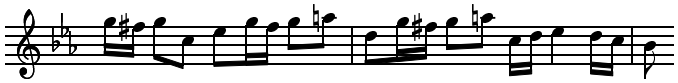
Our run of the SimSegmenter located the dux and comes in perfectly bounded subgraphs – no notes belonging to other phrases are included. The nonterminal number 2 (nt 2) is the dux, and nt 10 is comes. We show here the first occurrence of them: dux, nt 2, measure 1:



[43]

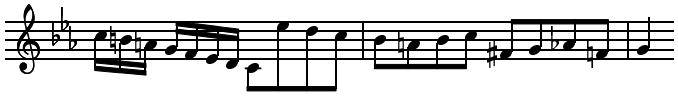
[43] There is only one occurrence of nt 2 labelled DAP. This is also because of the bug in this run of the segmentation algorithm. That occurrence was found as in an occurrence of another

comes, nt 10, measure 3:

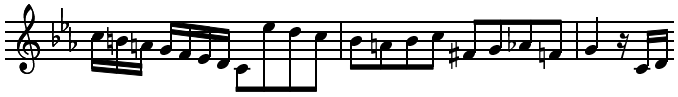


There are a total of 6 expositions of dux in the fugue spread out on the three different voices (5 starting on C and one on E♭) and 2 expositions of comes (both starting on G). The algorithm located them all as occurrences of nt 2 and nt 10 respectively. The nt 2's have however on two occasions been put inside a compound (nt 20) consisting of just one rest and then the nt 2. We will return to the themes below.

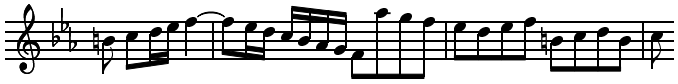
The theme (dux and comes) is often accompanied by a counterpoint which is a little thematic fragment which fits well to the theme. The counterpoint is therefore also an important fragment in the fugue. The SimSegmenter did not find the counterpoint in a so well grouped form as with the dux and comes. In the purest form, the counterpoint looks like this:



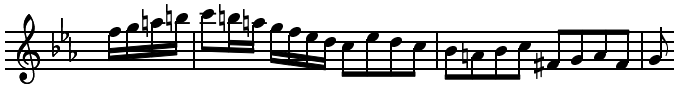
but what we found are these variants:
nt 4, measure 3 and 20, two extra notes (and one rest) in the end:



nt 1, measure 6, a too long beginning:



and its rhythmic variant nt 1, measure 14:

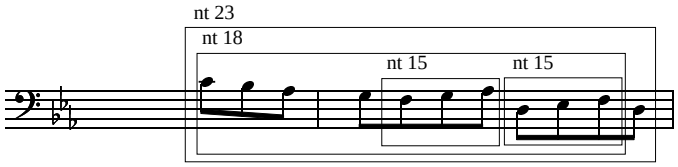


These two preceding figures are in fact a good example of what DFI was designed to do – notice the decim (octave + third) jump in the first figure in the first whole bar. It has been matched with a third jump in the next.

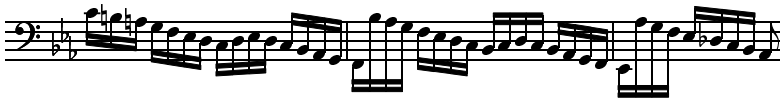
The nt 23's in measures 11 and 27 consist only of the last part of the counterpoint, and have been discovered as a nested structure. The first part of the "real" counterpoint – the descending 7 sixteenths were never connected to it as in this case. This has two explanations: in measure 27 the counterpoint has been split in two voices – the first part of the counterpoint is found in the top voice and the last part is found in the the middle voice. Since the graph is partwise, the SimFinder cannot connect the pieces, but there is also a difference in the jumps between the two parts, so it would demand another viewpoint to locate

nt 2 and with the measure `SeqDC_DAPALGrp`

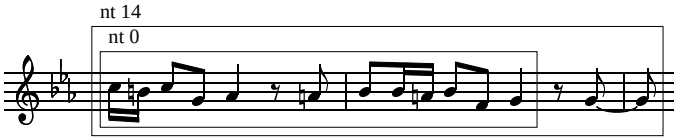
this connection. In measure 11 the first part of the counterpoint was connected to another very long repeated pattern (nt 3). This is nt 23 in measure 11:



The longest match found is nt 3 of size 40 which occur twice. The first in measure 9 (shown below) and the next in measure 22, this time starting on bes.



Other similarities of thematic origin occur, as for example nt 0 (inside nt 14), which can be found in imitation with itself (two beats displaced) twice in the score (starting in measure 9 and measure 22):



This should not be considered as a dux nor comes, but this is a little theme in its own right. The nt 14 could in two occasions with success have extended the phrase with one more note in the end, which would have completed the phrase, but the four occurrences do have different endings.

The score on page 173 and page 174 constitutes the final analysis from our program. We are not able to generate a more overall analysis of the piece – an analysis on a higher hierarchical level, where we abstract away the explicit information characterising the compound vertices. Such an analysis could reveal a structure in the structures we have found. Let us look at what there is to find. To give a short more overall and less detailed formal-founded analysis of this piece according to its similarities to itself, we have made figure 61. It shows the occurrences of dux, comes and counterpoints found in the piece. We see a pattern in the expositions of the themes in relation to the counterpoint. The counterpoint seems to occur alternately below and above each theme entry. This is a structural phenomenon, shaping the music according to the particular idea. Dux and comes are not represented with the same compound vertex. If we designed a viewpoint ignoring that differences of the two (and if we found the correct boundaries of the counterpoint), perhaps by running a non-sequential search on the resulting mothergraph, we could group some of the occurrences of theme and counterpoint together into larger compounds, and thus show a higher hierarchical level.

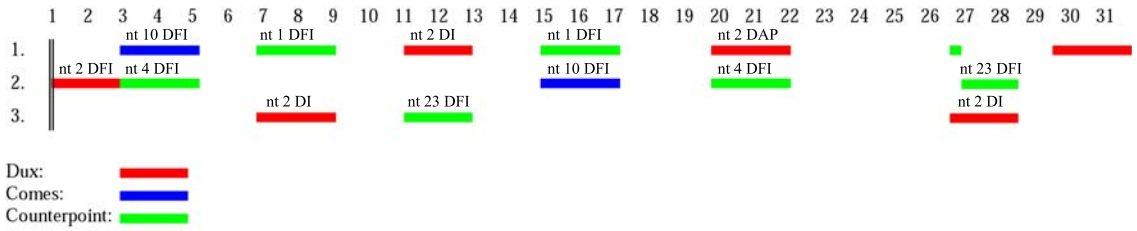
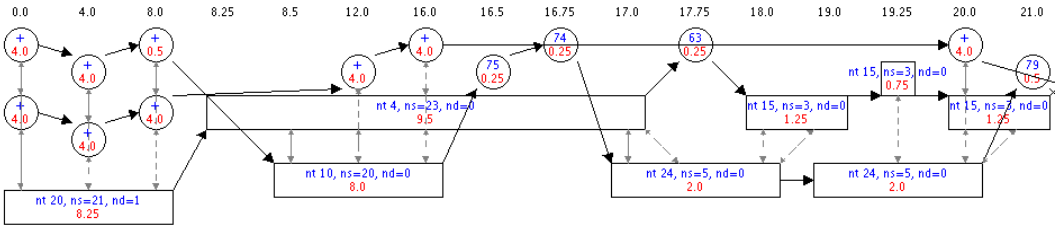


Figure 61: The overall structure of the fugue. The lines refer to the correct occurrences of the themes. The numbers refer to our analysis.

Another issue in using the SimSegmenter as a fugue analyser is that we are not able right now to determine the importance of the patterns found. By looking at the mothergraph, we cannot tell which compounds could be dux or comes or counterpoint, but some reasoning about the size of the similarities and the number of the occurrences would be needed to determine this.

Finally let's look at the beginning of the resulting mothergraph:



The nt 20 is the dux and the nt 10 is the comes. Above the comes is the counterpoint nt 4. Remember that the vertical position of the compound vertices do not reflect pitch. Also visible is three occurrences of nt 15 and two of nt 24. This analysis is based totally on the self-similarities of the piece. In a fugue, self-similarity is an extremely important feature. Not many notes are left unmatched. Only one pattern (nt 15) has the smallest allowed size of three simple vertices. We would like to conclude that to make an analysis totally based on self-similarity seems certainly to be one legal, though incomplete, way to gain insight into the structure of this kind of music.

5.5.3 Non-partwise non-sequential segmentation of *Jesu meine Freude*

This time we used only two similarity measures. Due to performance reasons we were forced to impose a maximum size of the subgraphs we used for searching. Otherwise the program would go slower and slower as the size of the match increased, ending in too long evaluation time. Here are the parameters:

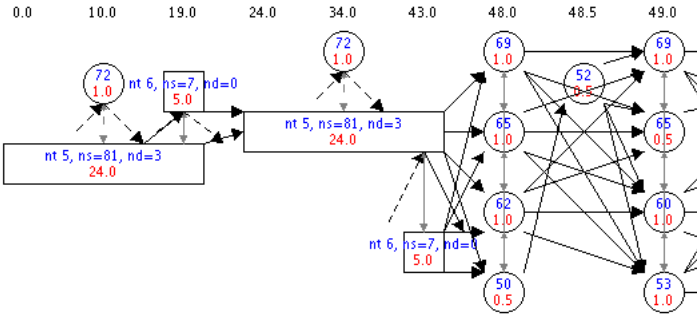


Figure 62: The start of the resulting mothergraph

Population size	120
Generations	250
Crossover	0.0
Mutation	0.4
Fresh Blood Chance	0.4
Initial size	10
Max size	16
Similarity measure	`nonSeqBagIR_DAP_Grp`
	`nonSeqBagIR_DI_Grp`
Limit	1.7
γ	0.5

We have chosen to forbid the subgraphs to grow to a size of more than 16 vertices. This restriction has some serious consequences for the result of the search. The SimFinder tries to make well bounded phrases, but the size restriction forces it to leave out some notes in each phrase. Another restriction in this search is that the minimum size of accepted subgraphs is 3. This is also highly visible in the result. But the SimFinder does what it can to meet all restrictions and preferences, so here we present the result. Again the similarities in the repetition are found – the possible overlap with the last phrase did not occur. Figure 62 shows the start of the resulting mothergraph. At first sight, the segmentation result does not look as beautiful as in the sequential case. The piece is segmented into seven different compound vertices. The limit value of 1.7 leaves no room for matches which are not totally correct, so the segmentation of the first six bars and the repetition of them is done in exactly the same way. The repeated six bars have been reduced to three vertices each: a compound (nt 5), a note and a compound (nt 6). Please notice that although our graph window decided to layout the 2×3 vertices in question in different ways, they are actually equal. With the minimum size of accepted subgraphs on 3, the SimFinder is not allowed to put the remaining six vertices into two final compounds, each containing six bars. This is a little unfortunate. Had there been four vertices left in the mothergraph, we are convinced that this would have happened. The same restriction means that no matches are found in the last seven bars of the chorale.

But let us look inside the compounds. Figure 63 shows the first six bars segmented. The nt 4 is a perfectly bounded entity consisting of the entire first

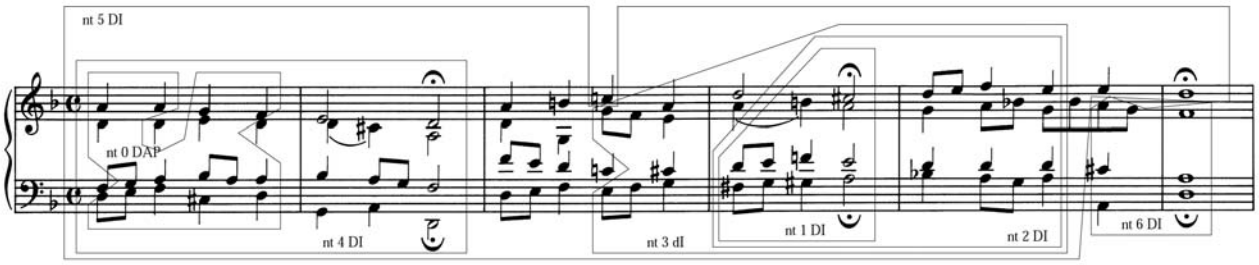


Figure 63: The segmentation of the repeated first six bars.

phrase of the chorale. Due to the maximum size of the subgraphs, the nt 0 is not able to span over the entire first measure (which contains more than 16 notes), but nt 0 also suggests that the first bar alone could be a subphrase. Also nt 1 suggests a subphrase. It is well bounded by ending on the fermata. But nt 2 and nt 3 are pulling the extension of the phrase in different directions. Instead, we end up not finding the boundaries of the last two phrases, but still the similarities found are correct.

For this experiment we also give the graphgrammar built when substituting subgraphs with compound vertices. The graph grammar is found in the appendix, beginning on page 178. It is a list of production rules, one for each compound vertex ever substituted in the run. This means a total of 14. The left side shows a compound (nonterminal) in a context – the vertices which have edges connecting to it, and the right side shows the result of the production (also including the context).

5.6 Summary

In this chapter, we have presented a segmentation algorithm based almost uniquely on musical parallelism. Building blocks for a graph grammar are similarities located by the SimFinder. We have introduced compound vertices in the graph and discussed problems with compound substitution and the resulting inconsistencies in the graph. The solution has been to distinguish between strong and weak edges, and compound substitution is reversible in the present implementation by virtue of the anchor resolution of edges[44].

The viewpoints, view comparators and similarity measures of the SimFinder have been extended to cope with compound vertices. The way we compare and handle compounds in the view comparators can be improved, as we discuss in section 6.1.3.

The SimSegmenter uses thresholds to control the SimFinder and uses the general-to-specific ordering of its viewpoints to find motives and derivations in a piece. These will constitute the building blocks for the grammar, and their internal relations are based on similarity. Musical similarity or parallelism thus becomes the basic relationship between tokens of the grammar, and the relationships can be labelled more specifically with the viewpoint under whose transformation they were found to be equal. The SimSegmenter resembles the algorithm of Smaill et al., but it improves it in two respects: the SimSegmenter

[44] Although the actual reversal mechanism has not been implemented yet.

handles non-sequential similarities, and it not only removes the found similarities but substitutes them with compound vertices, which allows a hierarchical structure to be built with successive runs of the SimFinder. The result is a set of production rules that are context-sensitive in this implementation but could have been context-free. We have discussed how to use such a graph grammar generatively, which poses questions on the size of the included context and on the generalisation of context.

Lastly, we have presented three test results on a chorale and a fugue, which have shown a segmentation based on similarities. The similarities found are correct. The SimSegmenter most often finds what we expect it to. The grouping boundary rules from GTTM have a tremendous effect compared to segmenting without them.

We have introduced almost no mechanisms to control syntagmatic relations between grammar tokens on the same level. The grouping viewpoint is the only exception; it is an attempt to ensure that similarities are located at (and chopped out of) appropriate places in the graph, according to phrasing and grouping constraints. We believe that a deeper study of syntagmatic relations must be included in order to improve the built grammars.

The SimSegment algorithm tries to build a hierarchic structure much in the sense of Herbert Simon (see section 2.2.1). It uses the SimFinder to locate stable intermediate forms, which can then be used as building blocks for further construction. The parallel drawn by Simon himself to evolving biological systems, finding more and more fit individuals, is the more striking because the SimFinder is actually a genetic algorithm, i.e. a search method based on the principle of evolution. The central question is that of 'selectivity' – the information feedback that gives an ability to discern more stable subsystems (subgraphs) from less stable subsystems in order to abstract them away into components (compound vertices) that may be regarded as units in themselves. The SimSegmenter selectivity lies in a thresholded use of the SimFinder, the latter being almost solely based on similarity, although a few grouping preferences have been thrown in as well. Simon's two kinds of selectivity, the trial and error method, and learning from previous experience, are not implemented in the SimSegmenter. Trial and error could be built into the algorithm if a reasonable measure could be given of whether an analysis was coming to a dead end or proceeding in an acceptable way. The SimSegmenter could then backtrace and undo the compound substitutions that were unbeneficial and try another choice of stable intermediate forms. Learning from previous experience could be introduced through a number of ingenious methods. A simple way would be to remember often seen patterns (i.e. subgraphs) and allow the SimFinder to additionally rate the fitness of similarity statements according to their resemblance to statements that were previously found to be good intermediate forms.

6 Conclusion

Let's evaluate the three main ideas we have described.

Graph representation for non-monophonic music

We think there are many possibilities in the music graph. It is a very flexible and extensible representation. But this also means that we need very powerful analysis tools to exploit it. Searching for similarities in the graph is certainly possible, but locating the right fragments for segmentation purposes does not happen automatically. Particularly, the non-sequential subgraphs would benefit from additional guiding heuristics to ensure that they are sensible subparts of the piece.

Both sequential and non-sequential search in the partwise graphs have proven more successful than the respective searches in non-partwise graphs. This is almost certainly due to the added complexity of graphs that include inter-part edges.

Search for musical parallelism

We find that he use of the multiple viewpoint system is the most successful of our three ideas. This idea is also very extensible; ideas for new and better viewpoints keep popping up, so there are still many to try out. Even though we have restricted ourselves to repetition and simple transformation, we have succeeded e.g. in finding similarities in the fugue that we hadn't noticed before the SimFinder pointed them out. An important ingredient in our success with finding melodic similarities have been the MuseData encoding of diatonic information and our diatonic viewpoints. The addition of elaborating/simplifying viewpoints would be a great improvement.

Segmentation using a graph grammar

We regret that we haven't been able to present more results, both in the way of analyses of other pieces than the chorale and the fugue, but also in the way of validation of test results we have. The best validation we have of the graph grammar found is the analyses given in section 5.5. Using the grammar generatively would probably point out a number of weaknesses present in the current approach.

Even so, we have built a system that is able to segment a piece of music to some degree. We have tried it on a chorale and a fugue, which represent two extremes in compositional technique. We count it as a strength that we have been able to treat both with the SimFinder system and the graph representation. As discussed in section 5.4.4, p. 133 and under 'improvements and extensions' below, there are a number of loose ends that would improve the SimSegmenter. E.g. a more systematic and concrete examination of what constitutes syntagmatic relations in a musical grammar. The introduction of the grouping 'viewpoint' has improved the SimSegmenter results significantly. The similarities found in the SimFinder are usually good similarities, but the boundaries are the problem; we have tried a few of the grouping rules from the GTTM and seen that they do actually improve grouping significantly in spite of their small number.

Different analyses

The GTTM proposed four ways of making a preferred analysis: grouping structure, metric structure, time-span reduction and prolongational reduction. Our analysis approach was based solely on parallelism. This approach was not intended to be the only way to understand music, but a way of illuminating the structure of the music. No doubt we could have made a more complete analysis had we been able to perform other types of analysis – making multiple hierarchies to contribute to the final analysis.

Meter could be another viewpoint usable when searching for similarities. The metric accentuation emphasises passages in the music. For example on the measure level, in a 4/4 meter measure, the first and third beat are the most metrically accentuated – the first is the most important. A viewpoint picking out notes on metrically accented beats would be a natural one. In this way we are likely to find only the most important notes and thus discard all other differences.

We have not dealt with information regarding tension and relaxation (prolongational reduction). This is where the meaning or the logic of the music can be found, or at least a meaningful relation between the musical fragments in each hierarchical level (syntagmatic relations). Important on this issue is the harmonic progression of the piece. From a MuseData score we should be able to quite easily extract harmonic information. The graph representation allows us to express each 'underlying chord' as a set of notes, ranging in a given time span. A harmonic analysis could contribute to extending the vertex versus vertex based search we are depending on. Harmony could have been a basis for finding similarities in differently structured musical objects.

6.1 Improvements and extensions of the SimFinder system

6.1.1 The GA and the definition of musical similarity

A systematic tuning of the GA is still to be done. On the other hand, there are a number of general questions on musical similarity that are probably too loosely answered here to justify spending large amounts of time on fine-tuning the GA. It may not make sense to begin a thorough tuning before we are sufficiently confident that the defined similarity measures actually capture what we want to find. What fragments people judge as similar (when not depending on the most basic melodic alikeness) is a question we have completely skipped over, for fear of getting lost in the jungle of music psychology. For example a similarity measured in pitch contour is a hard one to control. If two passages have similar contour, they might be heard as such, but if one of them furthermore forms a harmony, and the other does not, or forms another harmony, we are likely to say that they do not sound alike. It would, however, bring substantial weight to a claim of finding similarities in music, if the similarity measures were backed with empirical evidence that they do really describe similarities that *perceptible* and thus relevant to human listeners. In the early versions of the SimFinder, we made a couple of informal listening experiments to judge the quality of the first similarity measures we had constructed. The impression was that there is a large deviation both among ourselves but also between our subjective

similarity ratings and the ratings computed by the similarity measures. If a sufficient number of human similarity ratings were produced in a listening study, machine learning techniques could be used to tune the numerical combinations of viewpoints that happens in the similarity measures.

There is ample room for improvement in the use of the GA. We have used a plain standard algorithm, but the search could be improved with several more advanced techniques from the evolutionary toolbox. Below we suggest co-evolution of similarity statements and similarity measures (section 6.1.8). As discussed on p. 128, the search could also be improved by using multiple SimFinder populations evolving concurrently and periodically exchanging genetic material between them.

There is room for creative improvement of the non-sequential mutation operations. E.g., using the best match found by the edge comparison could be used to mutate intelligently as we do for the vertex comparison (see p. 98). In general, mutations that are more 'musically aware' could improve the SimFinder search.

Although the GA that is at the heart of the SimFinder is a search algorithm like many others, the choice is not arbitrary. The music graph represents an enormous search space of subgraphs; particularly when inter-part edges are added (to create a non-partwise graph) and the search is extended to non-sequential subgraphs, the number of possible subgraphs explodes. Genetic algorithms are useful to search such enormous search spaces. But the price of using a stochastic search method, as we have discussed in section 5.4.2, is a non-deterministic analysis, where additional work is needed e.g. to ensure that all occurrences are found. It is also necessary to introduce heuristics to help the GA find stable intermediate forms of the appropriate sizes. In Jesu Meine Freude (BWV358), we would rather first find similarities among the three phrases constituting the repetition and then find them to constitute a larger repetition, than locating the entire repetition in the first run. This is necessary to build a good hierarchical description, and the grouping viewpoint and the maximum size parameter are examples of such heuristics. But the use of viewpoints is applicable to other search methods, as e.g. the work of Conklin has shown [CW95], and the music graph is a general and flexible representation. Despite our reasons to choose a GA, there could be a sensible extension of this project in trying out other viewpoint-enabled search methods on the music graph.

6.1.2 The music graph

As described in section 4.2.2, there are many possible extensions to the graph. We think it would be beneficial to include other symbols in a score such as slurs, beams, fermata, bar lines, either in vertices or as additional edges. The whole mechanism of viewpoints, view comparators and similarity measures would need to be revised to either take account of the new graph elements or to just ignore them if they are irrelevant to the particular viewpoint and keep on comparing graphs as though the new elements did not exist.

Non-sequential graph matching can probably be done a lot smarter and faster than we do. Edge comparison can be easily improved by comparing separately `SIMULTANEOUS` and `FOLLOW` edges of the two subgraphs, since comparing a `SIMULTANEOUS` edge with a `FOLLOW` edge will give a rating of 1.0 (i.e. 'unsimilar') anyway.

6.1.3 The SimSegmenter

The vertex comparison of compounds should do something smarter than simply compare the non-terminals of the two compounds. After all, they may be very different kinds of derivations of the same motif (e.g. a variation that is equal to the motif in absolute length and another variation found under 'diatonic interval').

It would have been nice to have more evaluation of grouping structure, in order always to find the smallest entities first, and thereafter using them for larger compounds. The maximum size imposed on the subgraphs (for better performance) does too much damage when segmenting.

We feel that there is a lot more to be done in the structuring of known transformations of the building blocks located in the graph, and probably letting this knowledge influence the SimFinder during segmentation. At present we store only the paradigmatic relation between a derivation and its original motif, but all the derivations are also in some kind of paradigmatic relationship. We may also wish to find variations that are related by the consecutive application of several different transformations, such that $fragment_1 = t_1(t_2(\ldots t_n(fragment_2)))$. We might use such a web of paradigmatic relationships to infer which is the real original version of the motive – not just the first located version, which is what we call 'the original' at present, but the purest and musically clearest version of it. This may in turn be used to guide the segmentation process to a more reasonable result.

In section 5.6, we considered briefly how learning mechanisms might be introduced in the SimSegmenter to improve segmentation through experience.

As explained in section 5.4.4 (p.131 and p.133), we could have made the SimSegmenter output context free graph grammar rules, or we could have worked more on generalising the context of production rules for the same non-terminal.

6.1.4 Multiple hierarchies

Perhaps a more stable hierarchical ordering could be obtained by searching separately for different basic kinds of compound vertices? We might build a separate harmonic grammar of the piece, where compound vertices are always simultaneities, i.e. chords, as well as a melodic grammar, where compounds are always sequences or near-sequences of notes, and then combine the two grammars. Other hierarchies describing meter, prolongation, etc. could be of use.

6.1.5 Elaboration, simplification, reduction

In section 3.2.2, we distinguished between two basic types of parallelism: repetition/transformation, and elaboration/simplification. The latter are more complicated, since they are transformations that insert and/or delete events in a musical fragment. E.g. a melody can be repeated in an embellished form, where grace notes are added. To see the exact similarity between the embellished version and the original, we would have to determine which are the grace notes and then remove them before comparing with the original. In other words, the viewpoint applied to the embellished version should be selective, picking out some notes that are not necessarily connected in the graph.

We have no way of locating patterns in notes that are not connected by the immediate precedence relations (`FOLLOW`, `WEAKFOLLOW`), or simultaneity (`SIMULTANEOUS`, `WEAKSIMULTANEOUS`). For example, in the GTTM, Lerdahl and Jackendoff base the time-span reduction on a reduction hypothesis: "The listener attempts to organise all the pitch-events of a piece into a single coherent structure, such that they are heard in a hierarchy of relative importance" [LJ83, p.106]. The time-span reduction locates the most important pitch-events and thus, when moving to a higher reductional level, leaves out less important pitch-events in between the important ones. The reductionally important events therefore are not connected directly to each other by `FOLLOW` edges in our graph. It is probable that certain interesting patterns recur on higher reductional levels, but we are not able to locate these, because the details of the less important pitch-events must be included in the subgraphs compared in the SimFinder.

Again this could be remedied by the introduction of *selective*, or *reducing* viewpoints that omit some of the information in a view. Such a viewpoint is basically a small reduction mechanism that picks out the most important events. We believe this would be an extraordinary improvement to the SimFinder.

6.1.6 Structure in structures

When trying to segment music, at some point we have to abstract away the information from the compound and simple vertices in a graph. We would like to be able to compare structures in the music with each other – disregarding the explicit note information revealed in the vertices below. For example to see that two pieces have similar forms (for example a sonata form), we must be able to compare the structures of both pieces to each other. The sonatas have most probably no similar significant melodic material in common, but their structure – the overall form is similar. We could call this a *higher-order structural similarity* because the resemblance pertains to a similarity in the structures of the building blocks, rather than to the correlation between the building blocks.

We are yet unable to find this kind of similarity with the SimFinder. Let us give a smaller example: the pattern AAB is similar to the pattern CCD, even though A and C (and B and D, respectively) could be very different. The SimFinder would not find such a similarity, because the view comparison requires A and C (and B and D, respectively) to be directly similar according to some viewpoint. This could be a viewpoint that compares the 'type interval' between two objects. In the present SimSegmenter, the idea closest to a 'type of compounds' is the non-terminal. We have not experimented with a viewpoint able to do this. For example, if we let the viewpoint denote type equality between successive elements, the sequence [A,A,B] gives [0,1] and [C,C,D] also gives [0,1]. From this viewpoint, we could discover the similar structure of the two sequences.

Another solution could be to introduce viewpoints which are more intelligent in comparing compounds and their relations, than the ones we have used, only comparing on the parallelism type. We could attempt to locate similar structures on the basis of relations between compounds. E.g. if B is a transposition of A, and also D is a transposition of C, the similarity can be located by looking at the paradigmatic motif-derivation relationships in and among the patterns. This presupposes, of course, that the A, B, C, and D patterns are correctly lo-

cated in the first place. This example is a sequential one, but the paradigmatic relationships must be comparable along edges in the graph in order to apply to non-monophonic music. This is a feasible extension of the SimFinder.

But this raises another question. Is it interesting to compare a sequence of three 'small' compound vertices (when counting in nested size) with a sequence of three much 'larger' compound vertices just because both passages contains repetitions? Sometimes yes, but the point here really is to be able to keep track of which abstraction level the two sequences are located – can we compare a top layer with a lowest layer? In the way we build our hierarchy we have very little control over which elements could be classified as belonging to the same abstraction level. A schenkerian division of the abstraction levels into exactly three reduction levels could be a way to deal with this.

6.1.7 Ambiguity

We have ignored altogether the issue of *ambiguity*. Sometimes a musical fragment may be interpreted in different ways and no interpretation is more prominent than another. A small example is *elisions*, where a single note can be construed equally well as ending one phrase or beginning the next. On a higher level, there might be different structural interpretations of an entire phrase, section, or movement. In terms of generative grammars, we can insert another production for a given non-terminal to produce alternative renderings of the same fragment. For purposes of analysis, it could be useful to be able to insert, in a derivation tree for a given piece, nodes containing several alternative derivations for a fragment.

6.1.8 Co-evolution of similarity measures and similarity statements

A more drastic change to the SimFinder would be to drop the idea of producing weighted evaluations of similarity statements through the combination of viewpoints and view comparators in similarity measures. Instead, each transformation function describing a musical similarity we desire to locate could be implemented in an *evaluator* object that 'clicks', or responds, when the two subgraphs are exactly equal under its transformation[45]. It would have a boolean output that was much easier to interpret and combine with other outputs. The SimFinder could then be adaptive in the sense that we don't tell it what kind of similarity to look for, it adapts to the similarities it finds. If a similarity statement is found, where the PitchInterval evaluator 'clicks', a natural mutation of the similarity statement would be, not to move the subgraphs, but to try out the AbsolutePitch evaluator. The fitness of a stronger viewpoint that clicks should of course be better than the fitness of a weaker viewpoint that clicks. This is an example of co-evolution of the similarity statement and its similarity measure, where the viewpoints applied are selected through evolution.

The whole notion of *similarity statements* could be elaborated more. A statement can be true or false in itself. But it could also be a conjunction of other similarity statements. This way, if the subgraphs contained in a conjunction of similarity statements were required to be related to each other, a similarity statement could describe not only the similarity of two subgraphs but of sets of

[45] The transformation could be applied to only one or to both of the subgraphs, according to the kind of transformation we are talking about.

subgraphs that are alike. The occurrence search, which the SimSegmenter performs now, would be built into the SimFinder, weighting similarity statements that combine many subgraphs stronger than those combining only two. This idea, especially in conjunction with the previous one of 'evaluator objects', we find very promising and indeed would have implemented if we had had another month.

6.1.9 Applying SimFinder to audio arrangements

In her attempt to define an abstract "Music-piece" (see section 3.1.3 and [Bal92]), Balaban defines an abstract framework of "Music Structures" whose elementary components are not necessarily confined to notes and rests, as found in a traditional staff notation of Western tonal music. The elementary components might as well be recorded audio sounds.

What would it take to extend the SimFinder to handle loop-based audio structures such as programmed drum tracks? Such compositions are typically constructed using a sequencer and a sampler and consist of a temporal arrangement (in the sequencer) of occurrences of pre-recorded digital audio representations (played back by the sampler) of individual drum sounds. Some occurrences are specified to be played back faster or slower than others, or to be transformed through other kinds of digital signal processing (DSP), e.g. equalisation, filtering, reverb, delay, compressor, chorus, distortion, etc. If this structural information is available for analysis (as it is from the sequencer file), we could imagine defining music objects not as individual notes or rests, but as occurrences of the basic pre-recorded audio sounds. Properties of such objects would include which basic sound is used, the playback speed, loudness, and applied DSP effects, and the viewpoints used when comparing objects would be constructed either with reference to their basic sounds and other properties, or using common DSP analysis tools, e.g. cross correlation, Fourier transforms, wavelets, etc. The introduction of audio comparison capabilities in the SimFinder system amounts to the following:

- subclassing the MusicVertex class with an audio-specific vertex class that has properties as mentioned above,
- imbuing the compound vertex class with abilities to envelop groups of music objects in DSP effects,
- writing audio-specific Viewpoints, ViewComparators, and SimilarityMeasures

The temporal structure in a programmed drum track differs from the temporal structure in a staff notation of tonal music in that the music objects in the sound track most often will not be placed in relations of immediate precedence.

The obvious solution to this is to insert rest objects spanning from the end time of the preceding object to the start time of the following object. There still might be issues with the inferred graph structure, depending on the level of quantisation imposed on the structure by the sequencer. In a quantised structure, the start times of all music objects have been discretised to some extent, or "snapped" to a temporal grid of beats imposed on the structure, just like notes and rests in a staff notation occur at an integer multiple of half

the smallest note length[46] occurring in the piece. In a non-quantised structure, however, the basic sounds might be placed on a much more fine-grained time scale, e.g. down to the sample length for the highest sampling rate occurring in the piece. It is common to have different attack lengths on different drum sounds, which means that the audible onset of the sound occurs at different intervals from the start time of the sounds. This means that a very precise placement of the audible onset times on a grid of beats requires the start times of different sounds to be aligned at different (although small) distances before the exact beat. It should be possible, though, to solve such problems by inserting an intermediate analysis that guesses at the structural starting points intended in the actual placement of sounds.

If no rest objects are inserted, the MusicGraph could be built using, say, `FOLLOW2` edges, that do not represent *immediate* precedence, only precedence. Vertices would then have `FOLLOW2` edges connecting them to the next occurring vertex – or vertices, if there are more than one occurring on that beat.

Analysing a drum track would probably yield many more exact repetitions of patterns than the analysis of scores of classical music. It is difficult to predict if this will increase the complexity of the SimFinder results so much that it will need remedy in some form of alteration of the search scheme, to work. Although the example given here focuses on drum tracks, any loop-based music arranged in sequencer files would be analysable in this way.

6.1.10 Applying SimFinder to non-western music

We have not tried our program on non-western music. But our system should be able to represent any kind of music which has a discrete division of pitches. Idiom-specific knowledge such as knowledge of diatonic scales is encapsulated in the viewpoint mechanism and should be easily replaceable.

6.1.11 Applying SimFinder to other areas than music

The SimFinder locates similar patterns in temporal configurations of attributed events. In theory, this should be applicable to other areas than music, where data may be configured as attributed events occurring over time. For example (although there are much more developed systems that specialise in this), if there are any recurrent patterns in the fluctuations of stock market prices, it should be possible to find them using a SimFinder. Imagine following the stock values of n companies. The continuous flow of data describing the stock price of $company_i$ must be reduced to discrete events such as "stock i begins rising", "stock i begins stagnating", and "stock i begins falling". These events will then be the vertices in the graph that the SimFinder searches for similarities, and they will be connected by edges signifying the temporal relations among them. Viewpoints could include e.g. the mean increase/decrease over the period that the event spans.

[46] There might be dotted notes, or double dotted notes, in which case start times would be integer multiples of a quarter of the smallest note length.

A SimFinder design and implementation

A.1 UML diagram

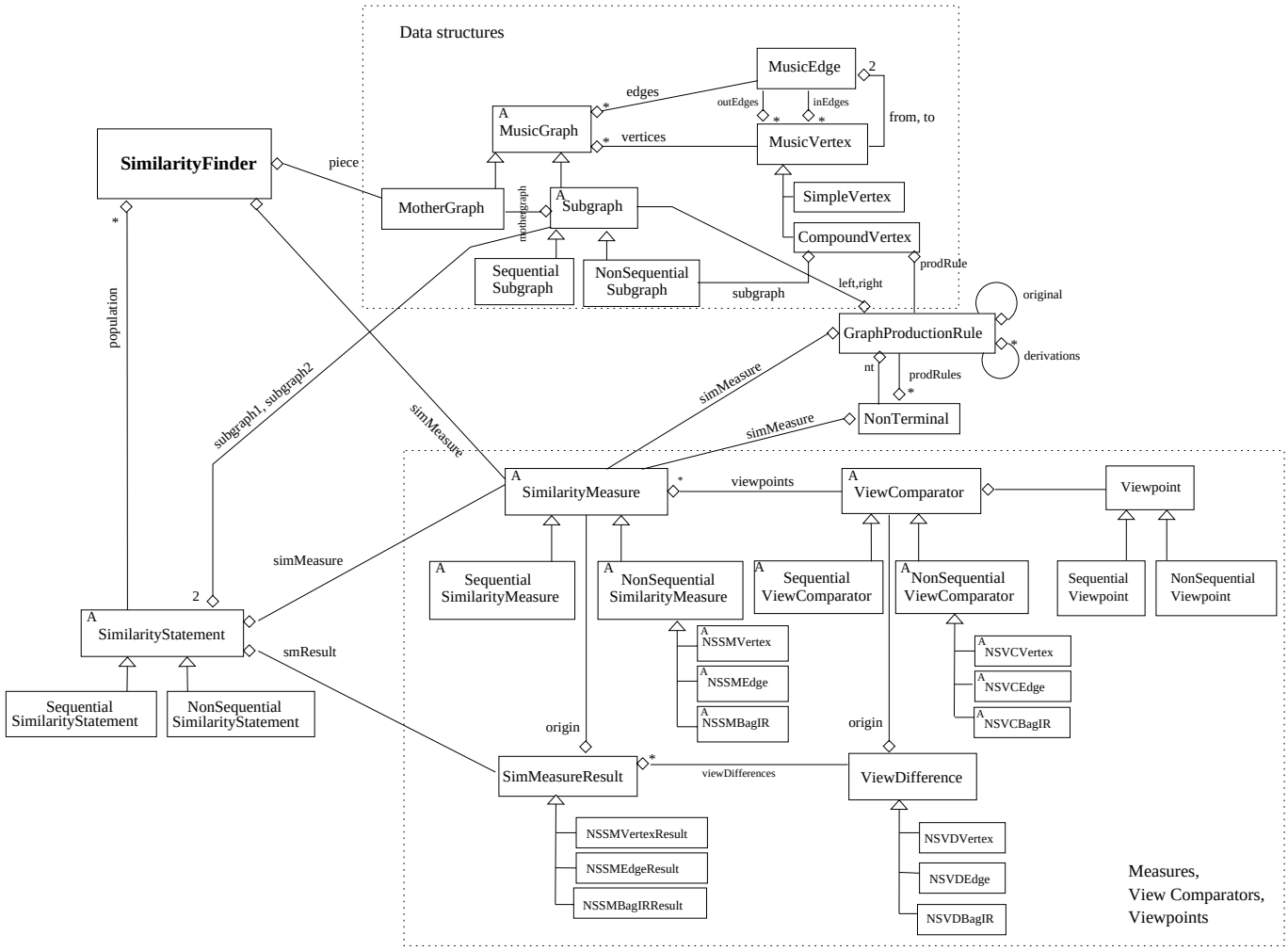


Figure 64: UML diagram of the SimFinder system

The classes of the graph structure are grouped in the dotted box 'Data structures'. The 'Measures, View Comparators, Viewpoints' box groups the classes of the viewpoint system, and to the left is shown the SimFinder class and the similarity statements that are individuals in the GA population. The two remaining classes 'GraphProductionRule' and 'NonTerminal' really belong to the sphere of the SimSegmenter, which controls the use of the SimFinder. We haven't shown the SimSegmenter in the UML diagram.

The `seqDC` and `seqMD` sequential similarity measures are static instantiations of anonymous subclasses of the abstract class *SequentialSimilarityMeasure*. The sequential view comparators are likewise to the *SequentialViewCompara-*

tor class. Non-sequential similarity measures are subclasses of *NSSMVertex*, *NSSMEdge*, or *NonSequentialSimilarityMeasure*; non-sequential view comparators are likewise with the subclasses of *NonSequentialViewComparator*. The following acronyms are used:
NSSM = Non-Sequential Similarity Measure
NSVC = Non-Sequential View Comparator
NSVD = Non-Sequential View Difference

A.2 Implementation

We have implemented the SimFinder system in Java. MuseData files are read using our own file input Java methods, but to read midi files, we use the jMusic package. jMusic discretises the midi events and interprets them as notes in a score notation. jMusic is available at *http://jmusic.ci.qut.edu.au/* .

The implementation of view comparators differs a little from the description in the SimFinder section (4) and from the design shown in figure 64. In the implementation, a SimilarityMeasure object does not point to a set of (ViewComparator, Viewpoint) pairs, instead it points to a set of ViewComparator objects. Each ViewComparator object then does the actual computation of its viewpoint on the subgraphs, because the implementation wouldn't be prettier if the viewpoint job was abstracted into a separate class. Therefore there exists no Viewpoint class. The NSSMBagIR class doesn't exist in the implementation either, because all the BagIR measures subclass *NonSequentialSimilarityMeasure* directly.

B SimFinder performance and tuning

We have not found time to tune the GA thoroughly, so the following sections are mostly presented in order to give an impression of the GA mechanism. Parameters are only varied one at a time. Also, there are a number of general questions on musical similarity that are probably too loosely answered here to justify spending substantial amounts of time on fine-tuning the GA.

Only the `seqMD_APAL` and `seqMD_PIAL` similarity measures have been tested, without any grouping structure evaluation. Parameter values are different for the non-sequential similarity measures, but we expect that the values be much the same for the `seqDC` measures. It seems that the *crossover*, the *fresh blood chance* and to some extent the *mutation* parameter do affect the mean size of the best found similarity after a fixed number of generations. The number of mutations applied in a mutate operation also influences the mean size. The *bonus value* and the *flattening constant* φ are not so important.

B.1 Varying the flattening constant φ (`seqMD`)

The following table shows how varying φ affects the mean size of the best similarity statement in a population of size 100. The mean size was calculated over 30 test runs of 100 generations each.

SimFinder parameters:

Population size	100
Generations	100
Crossover	0.0
Mutation	0.5
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`
μ	0.000001
Bonus value	0.9

Results:

φ	Mean size of best (30 × 100 gens.)
0.1	9.8
0.2	11.3
0.3	8.3
0.4	9.8
0.5	9.4
0.6	10.0
0.7	9.7
0.8	10.3
0.9	9.0
1.0	9.4
5.0	8.9
9.0	10.8
11.0	10.2
13.0	9.7
17.0	11.3
21.0	8.4
25.0	10.5
31.0	11.6
41.0	9.7
51.0	9.6

We do not find any significant development in the size of the best found similarity statements over φ. We have therefore chosen a high value of $\varphi = 40$ because it does not decrease the size modifier as close to 0 as lower values of φ do.

B.2 Varying the bonusValue parameter (`seqMD`)

The following table shows how varying the $bonusValue$ parameter affects the mean size of the best similarity statement in a population of size 100. The mean size was calculated over 30 test runs of 100 generations each.

SimFinder parameters:

Population size	100
Generations	100
Crossover	0.0
Mutation	0.3
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`/`SeqMD_PIAL`
μ	0.000001
φ	40

Results:

bonusValue	APAL Mean size of best (30 × 100 gens.)	PIAL Mean size of best (30 × 100 gens.)
0.1	9.4	7.1
0.2	7.8	7.3
0.3	10.3	7.5
0.4	9.5	8.6
0.5	9.8	8.2
0.6	9.0	9.2
0.7	8.9	10.0
0.8	9.4	8.2
0.9	10.8	9.8
1.0	8.6	7.5

It is difficult to judge if there is any development in these numbers without resorting to statistics. We won't. It looks like if some bonusValue is better than the others, it may be located in the high end. But this is only speculation. We set the bonusValue to 0.9.

B.3 Varying the crossover parameter

The following table shows how varying the *crossover* parameter affects the mean size of the best similarity statement in a population of size 100. The mean size was calculated over 30 test runs of 50 generations each.

SimFinder parameters:

Population size	100
Generations	50
Mutation	0.3
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`/`SeqMD_PIAL`
μ	0.000001
Bonus value	0.9
φ	40

Results:

crossover	APAL Mean size of best (30 × 50 gens.)	PIAL Mean size of best (30 × 50 gens.)
0.0	6.2	5.4
0.1	5.5	5.6
0.2	5.5	4.9
0.3	4.0	3.4
0.4	4.2	3.0
0.5	3.4	2.9
0.6	3.4	2.8

We find that there is a visible degradation in the size of the best match when the crossover rate is increased. We therefore keep it close to 0.

B.4 Varying the mutation parameter

The following table shows how varying the *mutation* parameter affects the mean size of the best similarity statement in a population of size 100. The mean size was calculated over 30 test runs of 100 generations each.

SimFinder parameters:

Population size	100
Generations	100
Crossover	0.0
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`/`SeqMD_PIAL`
μ	0.000001
Bonus value	0.9
φ	40

Results:

mutation	APAL Mean size of best (30×100 gens.)	PIAL Mean size of best (30×100 gens.)
0.0	3.0	3.0
0.1	6.4	5.3
0.2	8.7	7.8
0.3	9.5	9.6
0.4	10.7	11.5
0.5	10.7	9.0
0.6	9.4	8.8
0.7	10.2	8.1
0.8	10.2	10.0
0.9	9.3	9.9

The tendency here is that small values (0.0, 0.1, and 0.2) of the mutation parameter are not enough. We obtain better results when a certain amount of mutation allows the SimFinder to tumble about and explore more possibilities. Since there is no crossover used in these results, what is not created by mutation in the new population, is chosen through selection. Selection and mutation must work together, both to explore and to exploit the search space. We have to balance the exploitation-exploration goals to be able to use the SimFinder in the segmentation algorithm. Values between 0.3 and 0.5 seem to give such a balance.

B.5 Varying the 'fresh blood chance' parameter

The following table shows how varying the $freshBloodChance$ parameter affects the mean size of the best similarity statement in a population of size 100. The mean size was calculated over 20 test runs of 100 generations each.

SimFinder parameters:

Population size	100
Generations	100
Crossover	0.0
Mutation	0.3
Initial size	3
Max size	N/A
Similarity measure	`SeqMD_APAL`/`SeqMD_PIAL`
μ	0.000001
Bonus value	0.9
φ	40

Results:

$freshBloodChance$	APAL Mean size of best (20×100 gens.)	PIAL Mean size of best (20×100 gens.)
0.0	7.9	7.2
0.1	11.6	9.3
0.2	10.3	10.4
0.3	12.5	8.2
0.4	8.4	7.7
0.5	8.0	6.9
0.6	7.5	6.7
0.7	7.0	5.7
0.8	5.4	4.6
0.9	3.7	4.1
1.0	3.0	3.0

We find that the size of the best match rises when $freshBloodChance$ is increased until 0.2 or 0.3. Then it keeps falling when $freshBloodChance$ is further increased. We keep this parameter close to 0.3.

B.6 Varying the number of mutate operations per mutation

The following table shows how the mean size of the best similarity statement in a population of size 100 is affected when we vary the number of randomly chosen mutations applied each time the GA calls a mutation operation. The mean size was calculated over 30 test runs of 100 generations each.

SimFinder parameters:

Population size	100
Generations	100
Crossover	0.0
Mutation	0.3
Initial size	3
Max size	N/A
Similarity measure	`SeqMD_APAL`/`SeqMD_PIAL`
μ	0.000001
Bonus value	0.9
φ	40

Results:

#mutations	APAL Mean size of best (30 × 100 gens.)	PIAL Mean size of best (30 × 100 gens.)
1	9.7	8.7
2	6.8	5.7
3	4.3	4.6
4	4.6	4.2
5	3.9	3.9
6	3.6	3.5
7	3.1	3.1
8	3.1	2.9
9	2.8	2.9

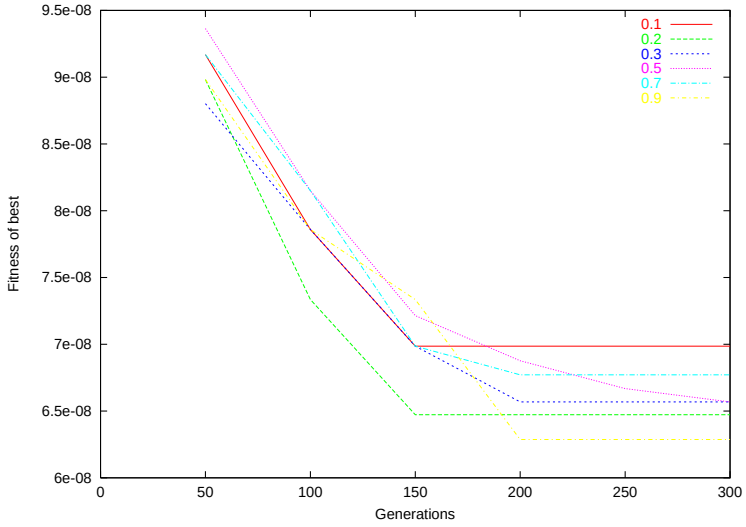
The time needed to locate larger matches clearly increases when the number of applied mutation operations in each mutation is increased. We conject that the exploitation ability of the GA is worsened. The fitness landscape is very spiky, so when applying more than one mutation operation to a reasonably good match, there is a greater risk of 'overshooting' and mutating to something worse than if we apply only one mutation at a time. We stick to a single mutation per mutate operation.

B.7 Size and fitness of the best individual over 300 generations

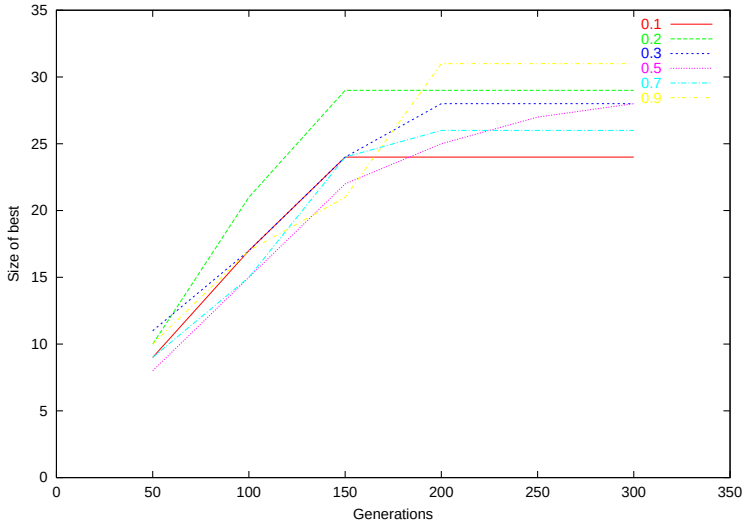
In section 4.3.4 the vertex usage was shown for six different settings of the mutation parameter: 0.1, 0.2, 0.3, 0.5, 0.7, and 0.9 (see figure 32, p.88). Below is shown the size and the fitness of the best individual from the 300 generation SimFinder runs that generated each of the vertex usage graphics.

SimFinder settings:

Population size	100
Generations	300
Crossover	0.0
Mutation	variable
Initial size	3
Max size	N/A
Fresh blood chance	0.3
Similarity measure	`SeqMD_APAL`
μ	0.000001
φ	40
Bonus value	0.9



↑ Best Fitness. Size of most fit ↓



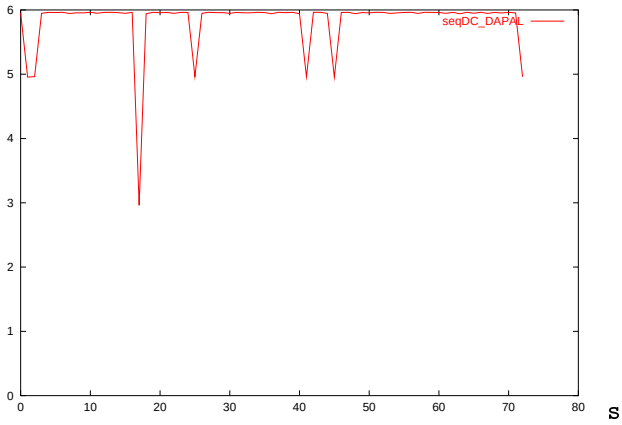
The two graphs are inverses of each other because the best similarity statement in all runs has been a perfect match, so the decrease in fitness value depends solely on an increase in size. The size modifier is thus shown in effect

here. For all mutation setting cases, the best fitness in the population converges to a value over the 300 generations.

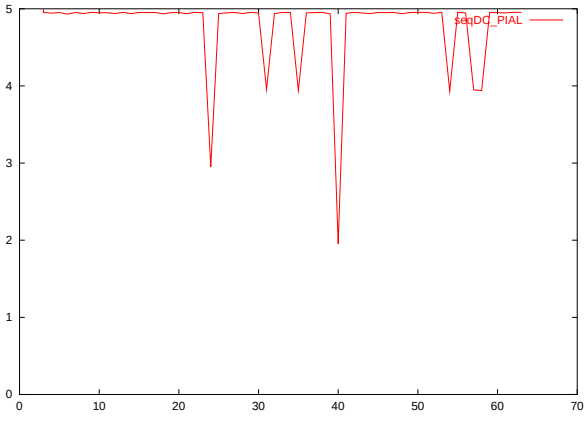
B.8 The fitness landscape

The following figures show the values of a sequential similarity statement while sliding one of its subgraph through the graph. The graph was built from the Bach chorale BWV358, and the plots are intended to show how spiky the fitness landscape is, in terms of how much the fitness of a similarity statement may be changed by a single slide operation. Subgraph s_1 of the similarity statement is picked randomly and stays fixed while subgraph s_2 is also initially randomly picked in the beginning of the graph and then slid from left to right through the graph. After each slide, the fitness value of the similarity statement is re-evaluated; this value is what is shown on the plot as a function of the slide number. Thus sharp points show places in the fitness landscape where the fitness rapidly changes – making it more difficult to locate a good match using slide operations.

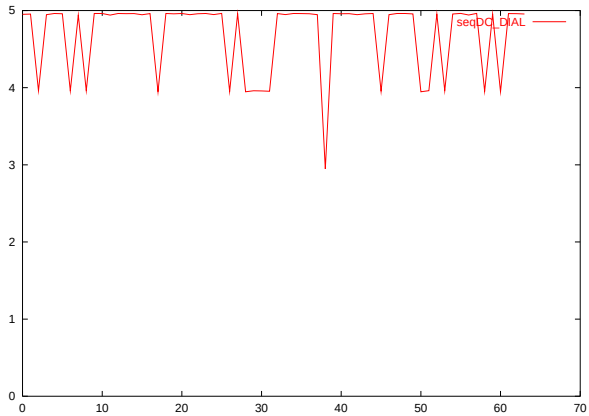
These plots are only intended to give a vague idea of the fitness landscape, since the subgraphs are completely randomly picked. We have made a slide plot for each of the sequential similarity measures. In addition to other differences, the `seqDC` measures incorporate grouping structure while the `seqMD` measures do not. Please note that the y-range may change.



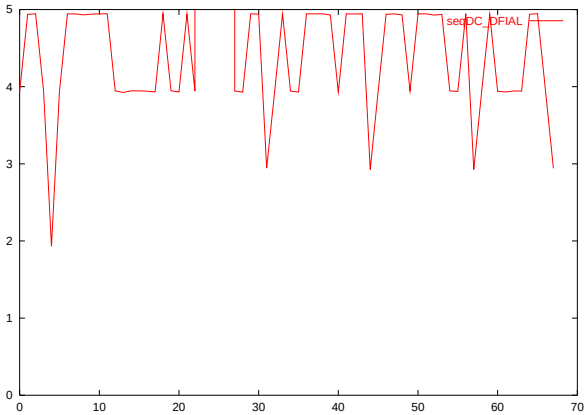
`seqDC_DAPAL`



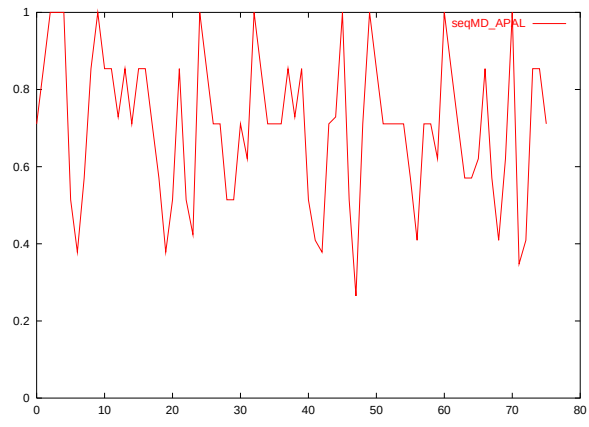
`seqDC_PIAL`



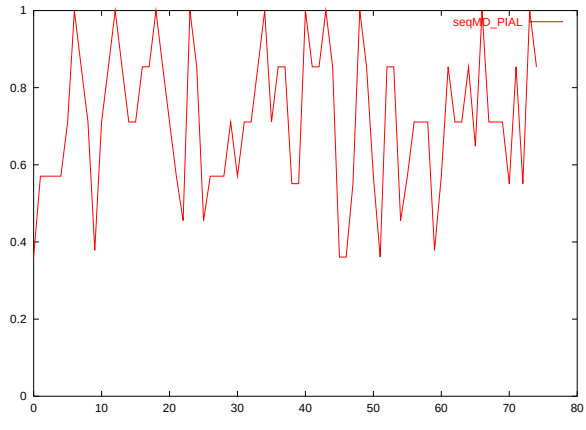
seqDC_DIAL



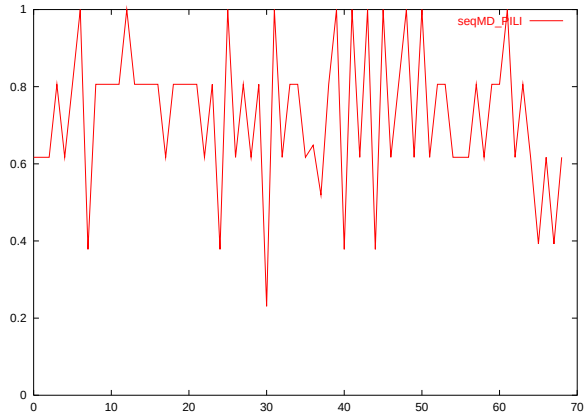
seqDC_DFIAL



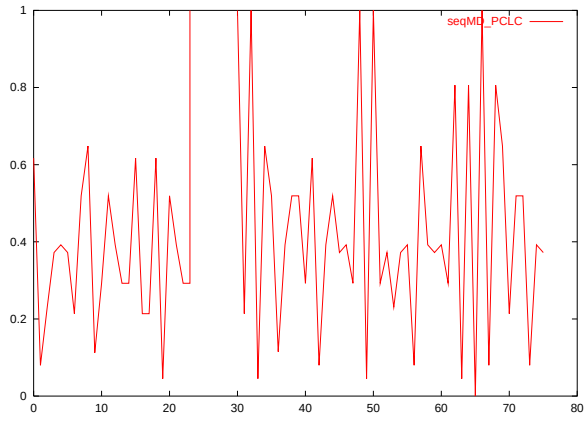
seqMD_APAL



seqMD_PIAL



seqMD_PILI



seqMD_PCLC

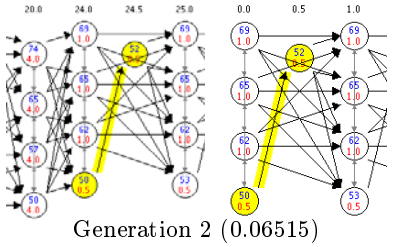
B.9 An evolving non-sequential similarity statement

The following is an example showing the evolution of similarity statements from a non-sequential SimFinder run. The SimFinder parameters were:

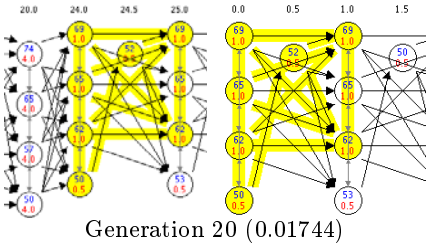
Population size	50
Generations	100
Crossover	0.0
Mutation	0.3
Initial size	3
Max size	N/A
Fresh blood chance	0.2
Similarity measure	`nonSeqVertex_PLIOS`
φ	40

The shown subgraphs belong to the *best* similarity statement of each generation in question. This is not the same statement from generation to generation, but a series of descendants of each other, each having a new mutation that enlarges the subgraphs. The population also contains a lot of less good similarity statements mutating aimlessly about. But the development of the best individual shows that the GA is able to exploit a found match.

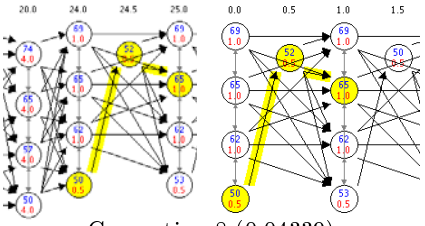
Only generations in which a change in the best similarity statement has occurred are shown. At first (generations 8-12) the GA tries out different matches of the same length. Then (gens. 13-41) it extends the match rather quickly, and finally (gens. 42-100) the extension slows down a little. At generation 100, the match was the same as at generation 80. The values of `nonSeqVertex_PLIOS` are shown in parentheses. Note that the best match is always a perfect match. Therefore, the only difference in the similarity measure value is the one caused by the size modifier.



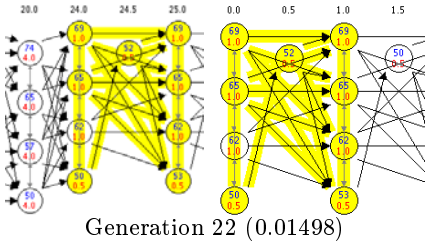
Generation 2 (0.06515)



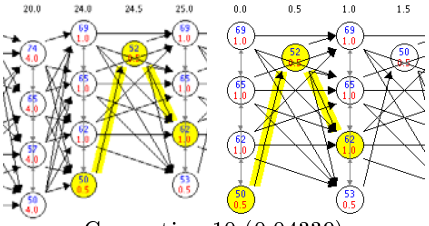
Generation 20 (0.01744)



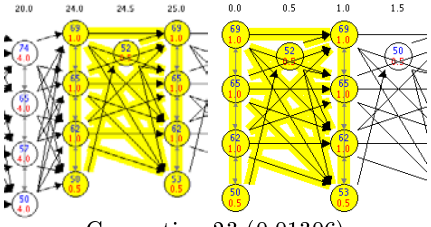
Generation 8 (0.04339)



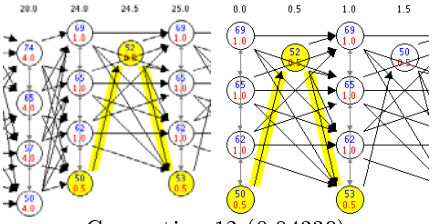
Generation 22 (0.01498)



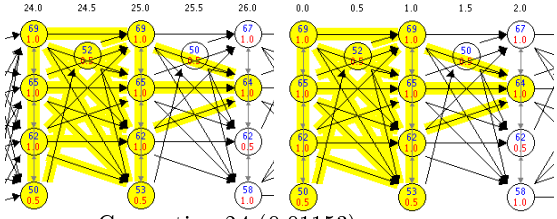
Generation 10 (0.04339)



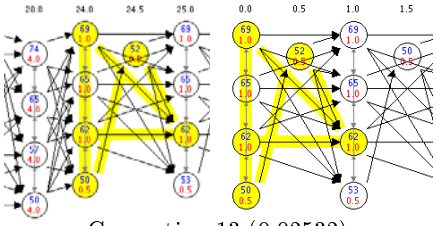
Generation 23 (0.01306)



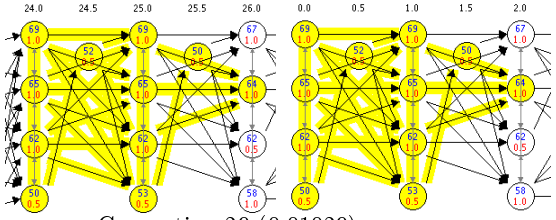
Generation 12 (0.04339)



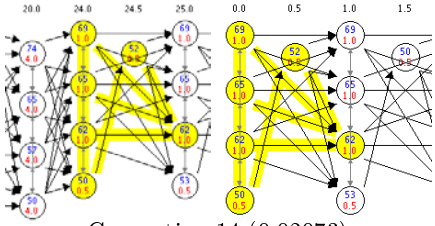
Generation 24 (0.01153)



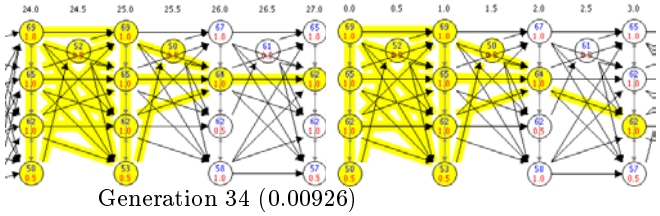
Generation 13 (0.02532)



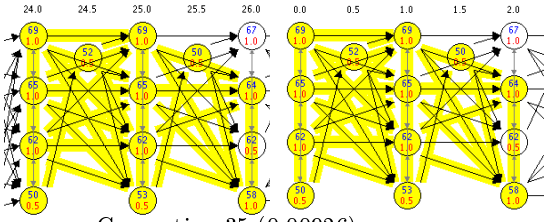
Generation 29 (0.01029)



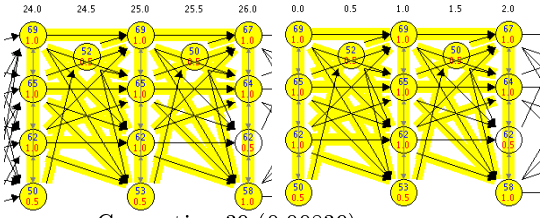
Generation 14 (0.02073)



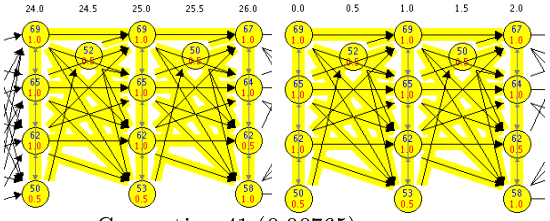
Generation 34 (0.00926)



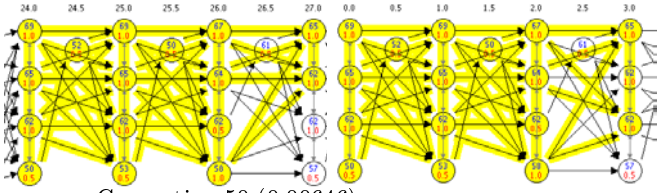
Generation 35 (0.00926)



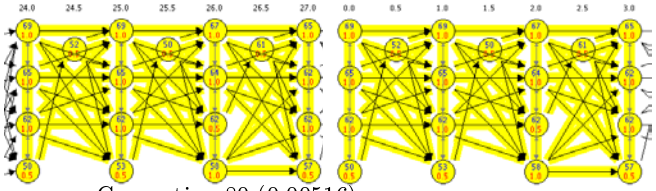
Generation 39 (0.00839)



Generation 41 (0.00765)



Generation 50 (0.00646)



Generation 80 (0.00516)

Generation 100: same as generation 80.

C SimSegmenter examples

C.1 SimSegmenter run with nonSeqVertex_PLIOS

The following settings were used in the simple test run described in 5.4.4, p.129.

SimSegmenter settings	
File format	midi
Partwise graph	yes
Similarity measures	`nonSeqVertex_PLIOS`
Population size	50
Motif search gens.	50
Occurrence search gens.	50
Gens. per limit check	20
Gens. after limit reached	30
Limit	0.05
Initial size	3
Max size	5
Crossover	0.0
Mutation	0.5
Fresh blood chance	0.4

The production rules for NT10 are shown in section 5.4.4. These NT10 rules produce two compound vertices and a simple vertex, but in a different context for each rule. The production rule for NT3 involves only simples vertices. When the production rule is created as the result of a compound substitution, every vertex connected to the inserted compound is included as context on the left-hand side of the production. On the right hand side, in addition to the context, NT3 produces four simple vertices: (50,0.5),(52,0.5),(53,0.5),(50,0.5). The rule for NT9 includes an NT3 compound as context for the production.

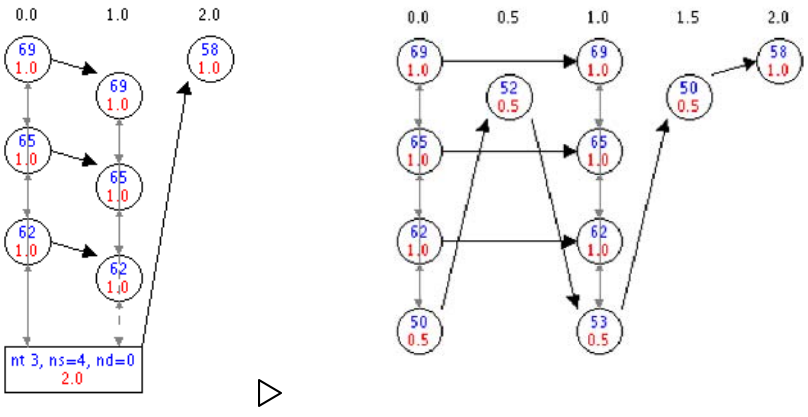


Figure 65: Production rule PR7 for non-terminal NT3 (nonSeqVertex_PLIOS).

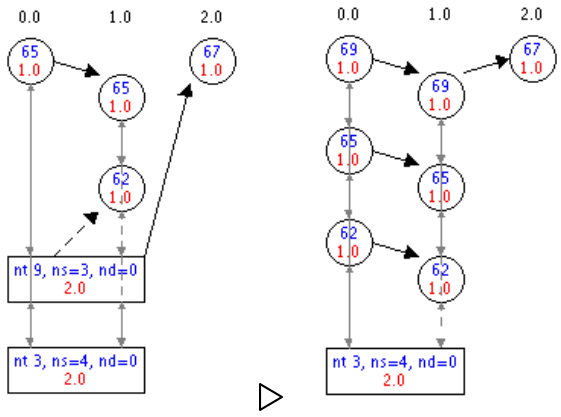


Figure 66: Production rule PR21 for non-terminal NT9 (nonSeqVertex_PLIOS).

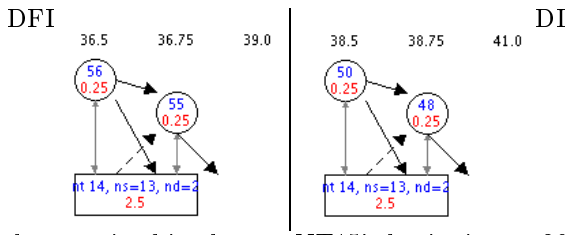
C.2 SimSegmenter run with nonSeqEdge_DAPAL, -DIAL, and -DFIAL

The following settings were used in the simple test run described in 5.4.4, p.131.

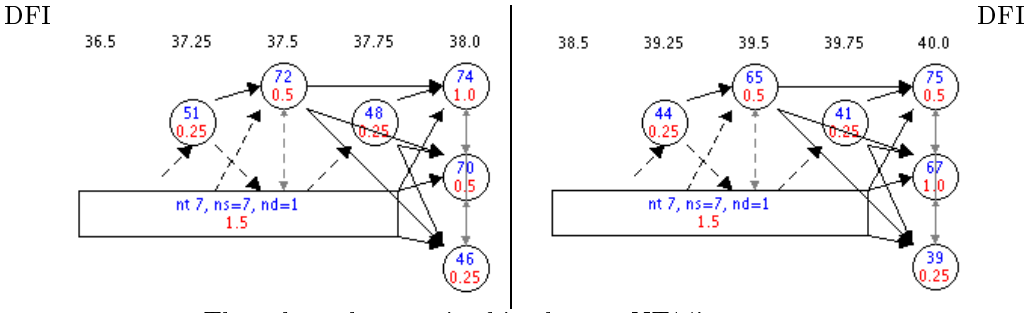
SimSegmenter settings

File format	MuseData
Partwise graph	no
Similarity measures	`nonSeqEdge_DAPAL`, `nonSeqEdge_DIAL`, `nonSeqEdge_DFIAL`
Population size	50
Motif search gens.	100
Occurrence search gens.	100
Gens. per limit check	20
Gens. after limit reached	50
Limit	0.06
Initial size	3
Max size	N/A
Crossover	0.0
Mutation	0.5
Fresh blood chance	0.4

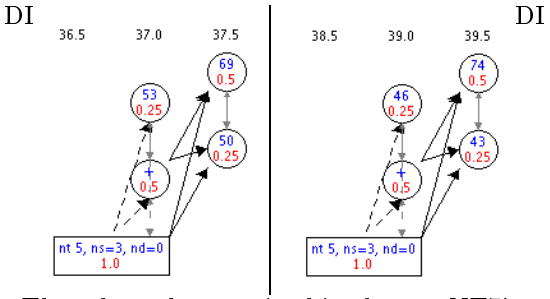
The following pairs of subgraphs need not have been substituted simultaneously, i.e. they are derivations of some third and/or fourth pattern, which was judged equal to these patterns under DI or DFI. The tonal viewpoint used is notated to the left and right of the subgraphs.



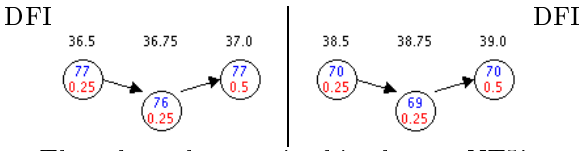
The subgraphs contained in the two NT15's beginning at 36.5 and 38.5.



The subgraphs contained in the two NT14's.



The subgraphs contained in the two NT7's.

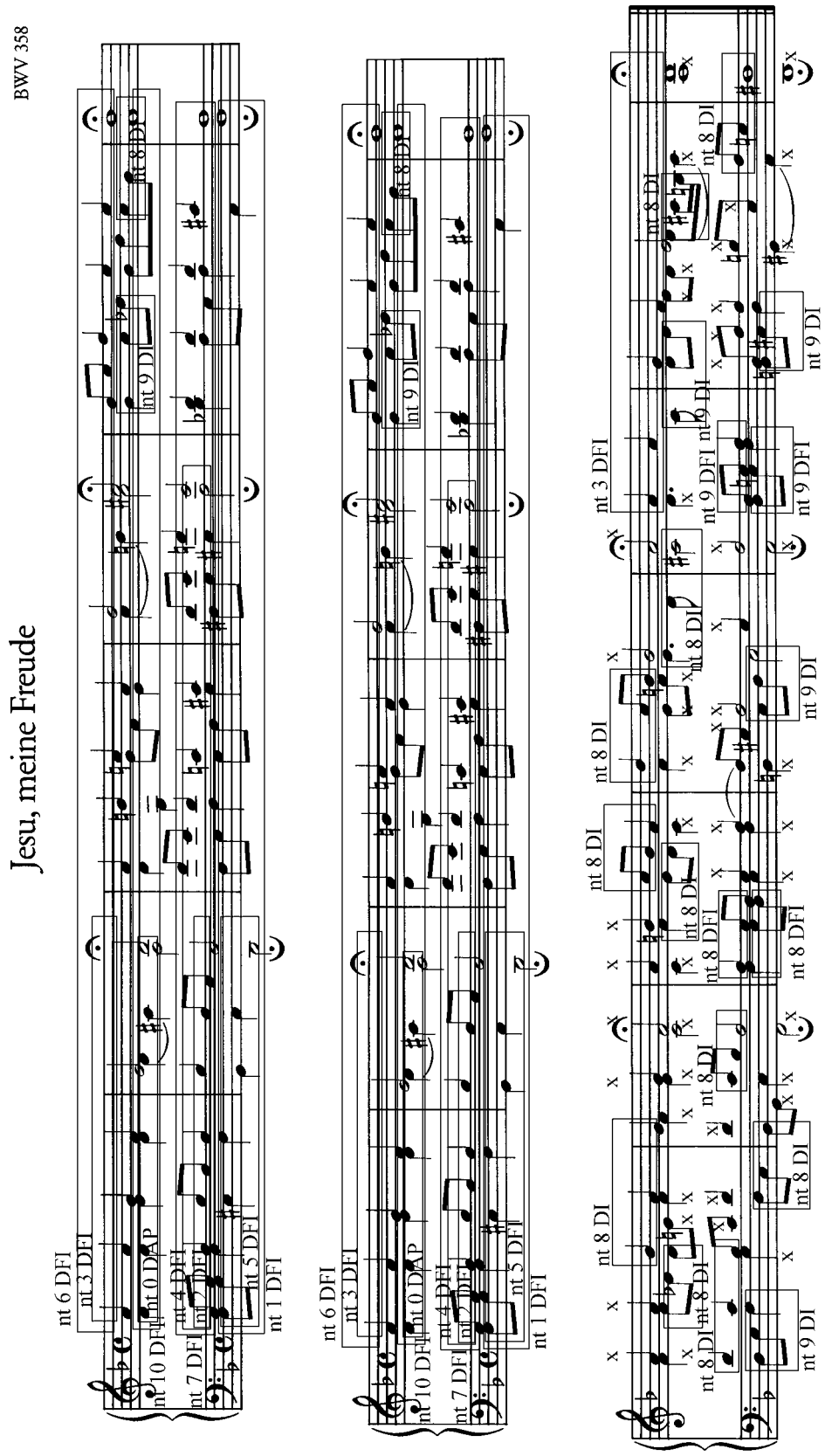


The subgraphs contained in the two NT5's.

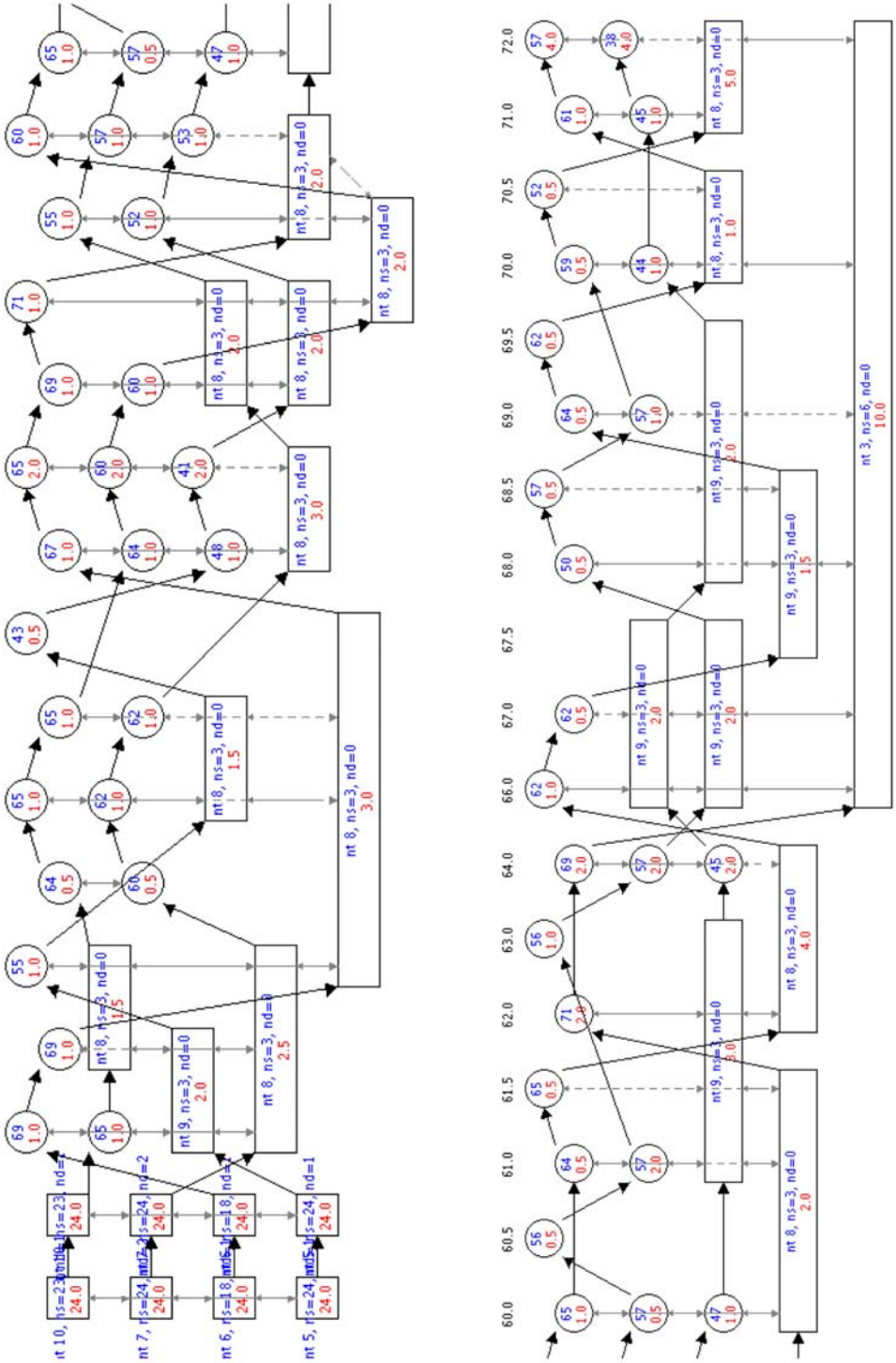
D Results from Simsegmenting

D.1 Segmenting a chorale partwise sequentially

D.1.1 The segmented score

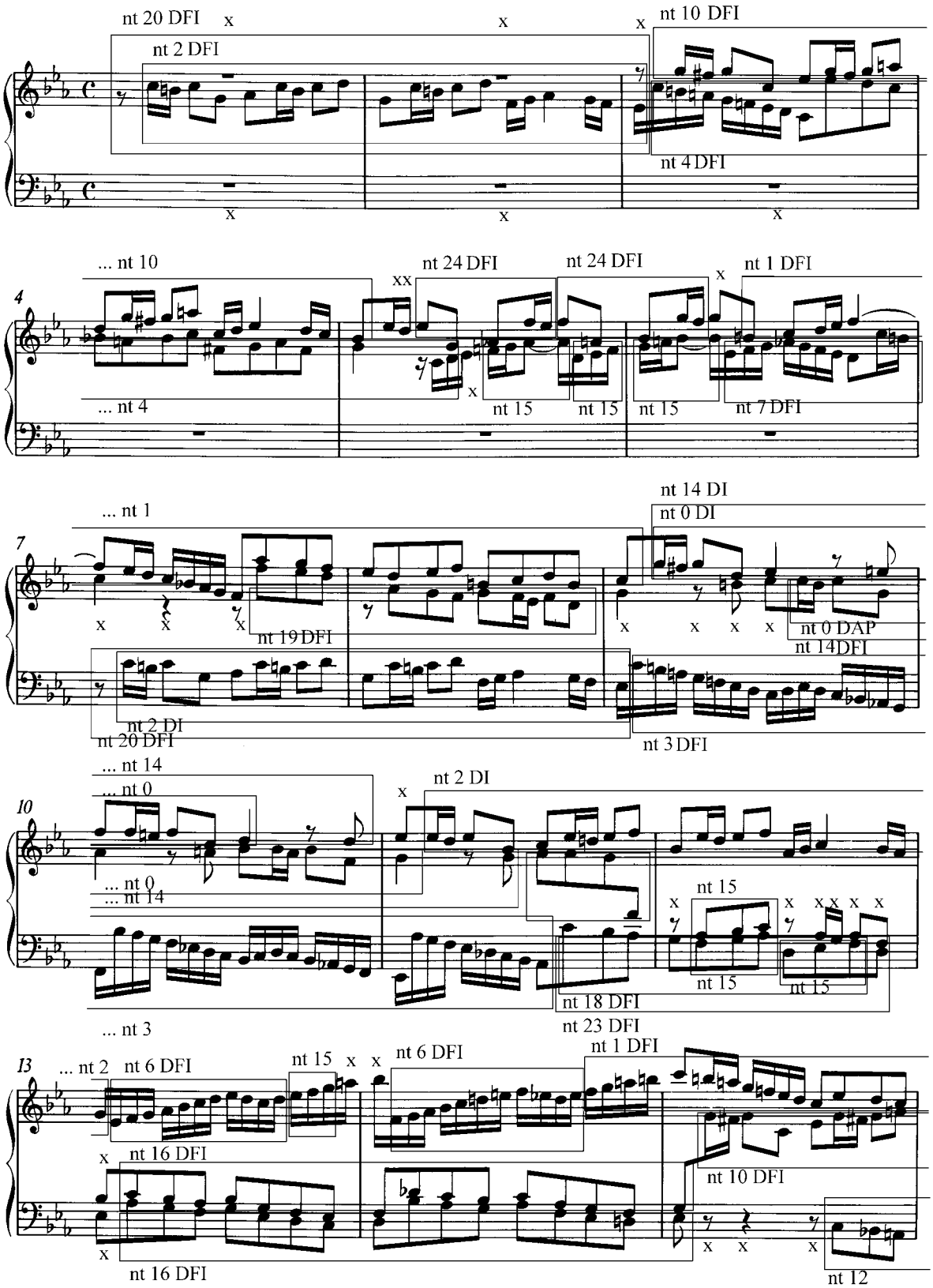


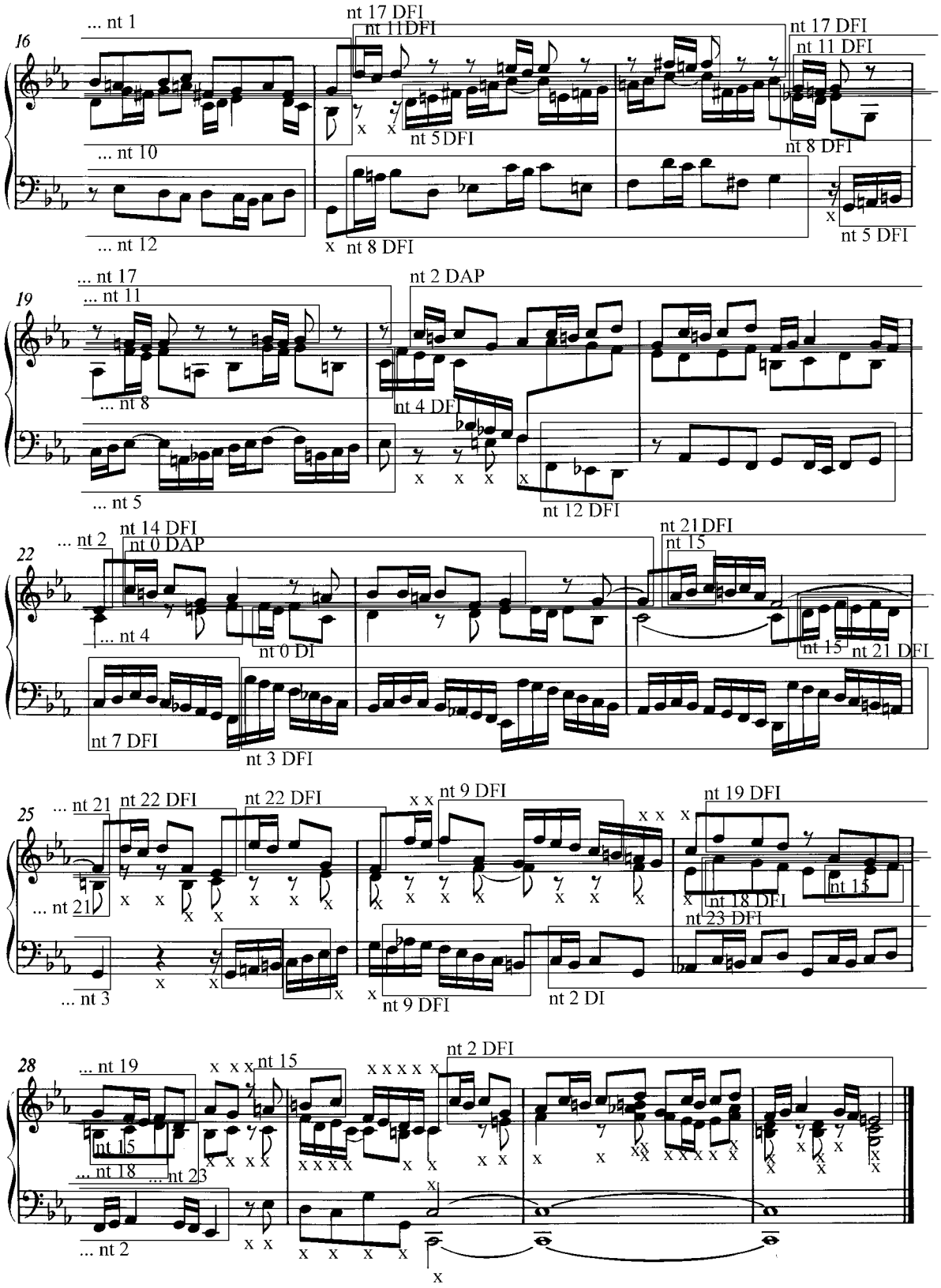
D.1.2 The final graph



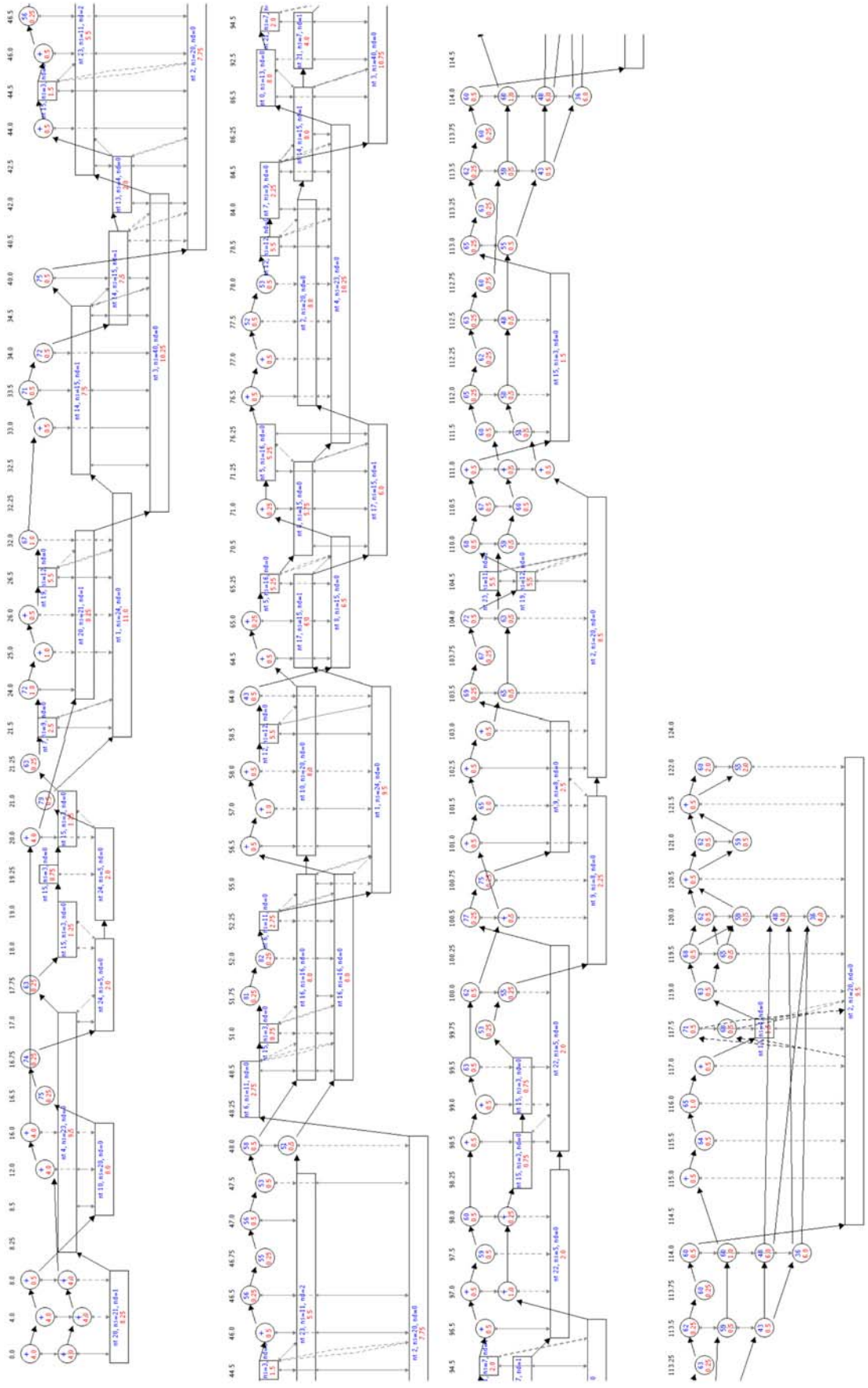
D.2 Simsegmenting a fugue partwise sequentially

D.2.1 The segmented score



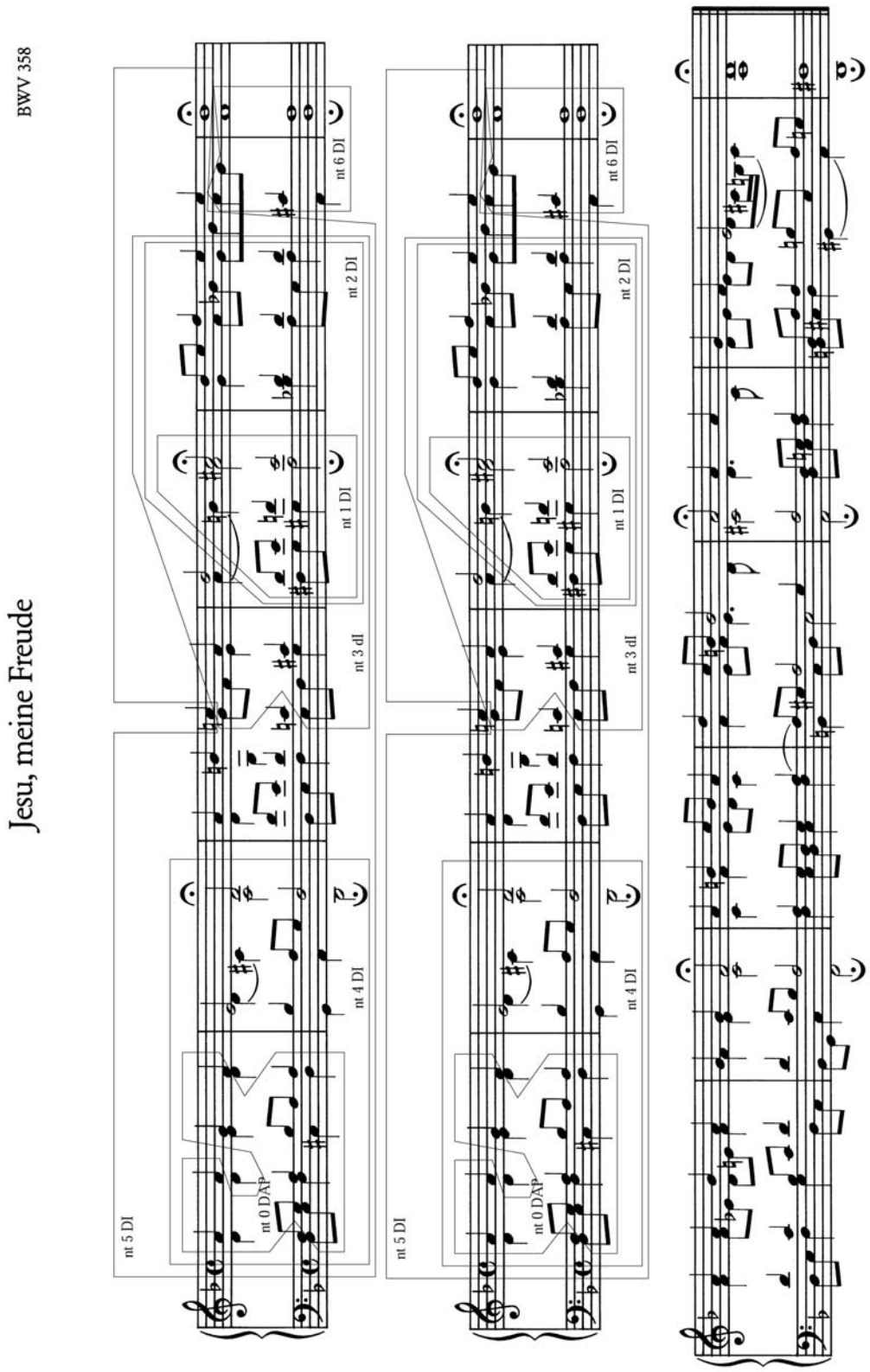

16
... nt 1
nt 17 DFI
nt 11DFI
nt 17 DFI
nt 11 DFI
nt 5DFI
... nt 10
nt 8 DFI
... nt 12
nt 8 DFI
nt 5 DFI
19
... nt 17
... nt 11
nt 2 DAP
... nt 8
nt 4 DFI
... nt 5
nt 12 DFI
22
... nt 2
nt 14 DFI
nt 0 DAP
nt 21DFI
nt 15
... nt 4
nt 0 DI
nt 15
nt 21 DFI
nt 7 DFI
nt 3 DFI
25
... nt 21
nt 22 DFI
nt 22 DFI
nt 9 DFI
nt 19 DFI
... nt 21
nt 18 DFI
nt 15
nt 23 DFI
... nt 3
nt 9 DFI
nt 2 DI
28
... nt 19
nt 15
nt 2 DFI
nt 15
... nt 18
... nt 23
... nt 2

D.2.2 The final graph

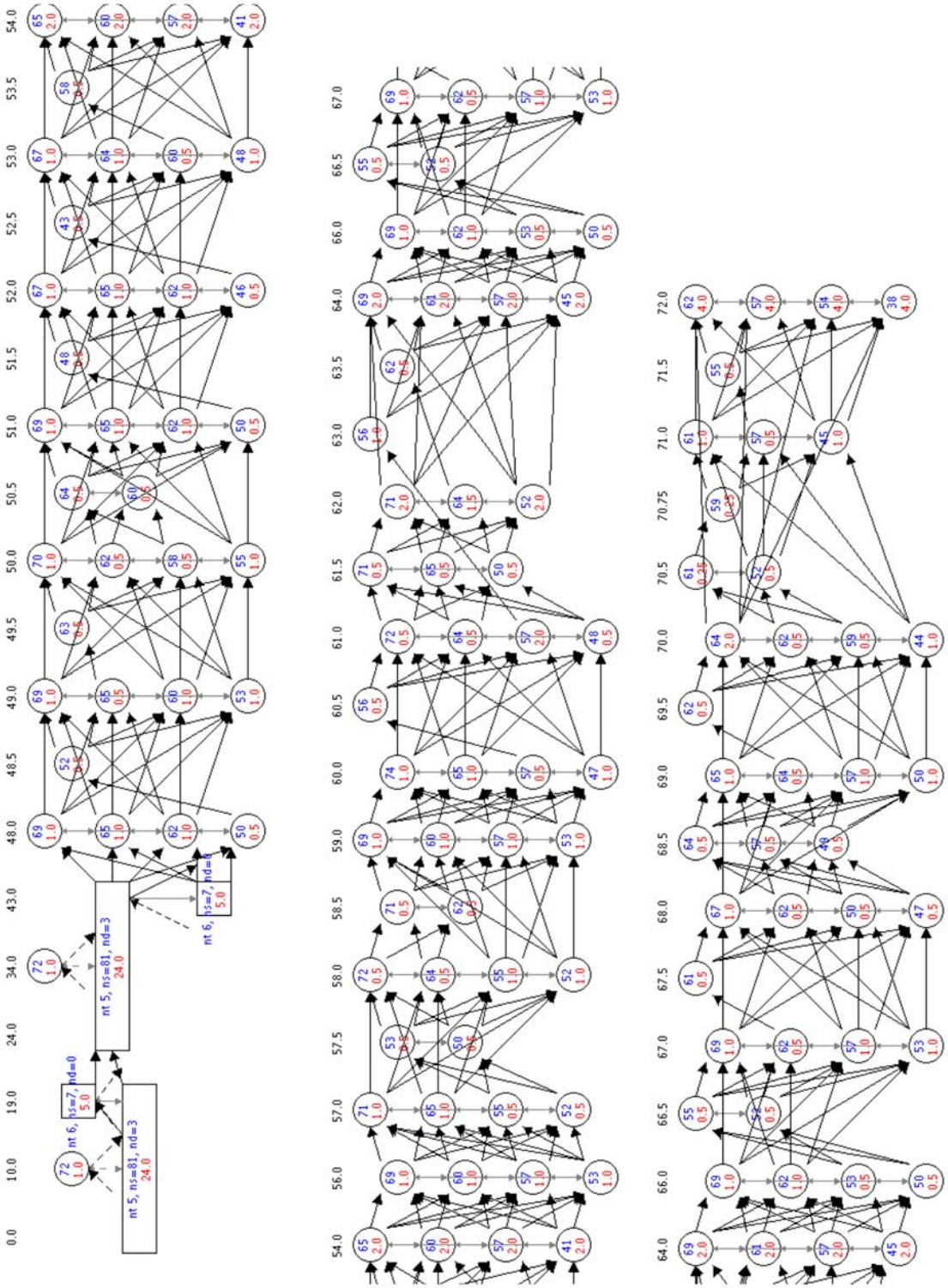


D.3 Simsegmenting a chorale non-partwise non-sequentially

D.3.1 The score when segmented



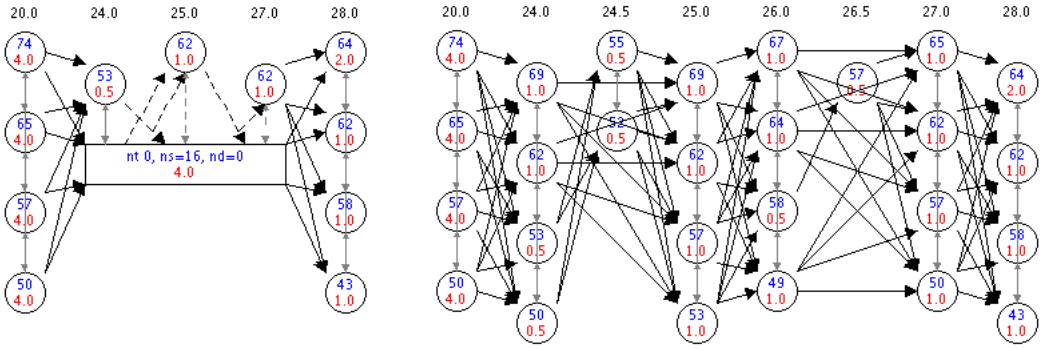
D.3.2 The final graph



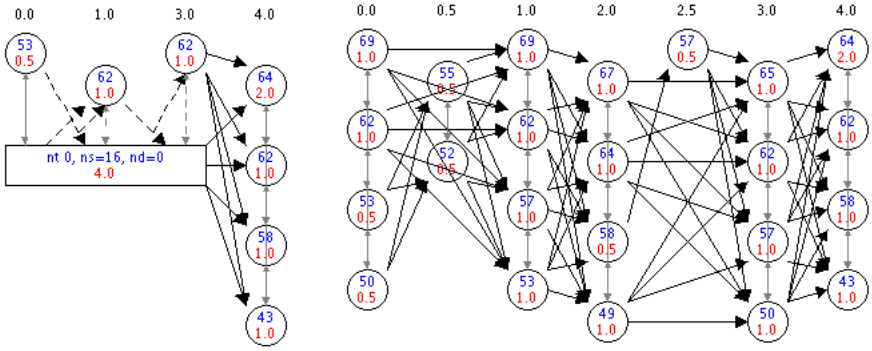
D.3.3 The graph grammar

The graph grammar is a list of production rules – one for each compound vertex ever substituted in the mothergraph. Read about the experiment on page 141.

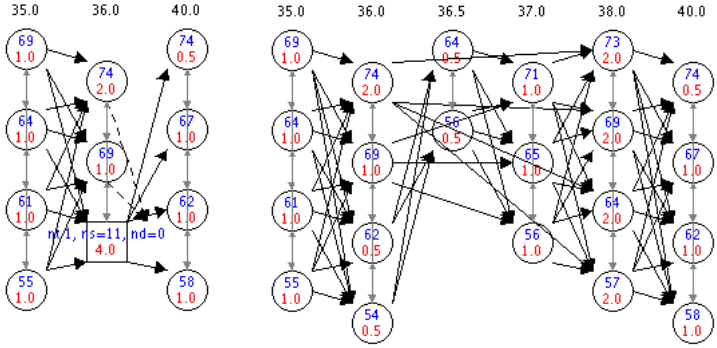
Production rule 1: PR1 nt 0 nonSeqBagIRTrie_DiatonicAbsPitch_Grp Orig(0)



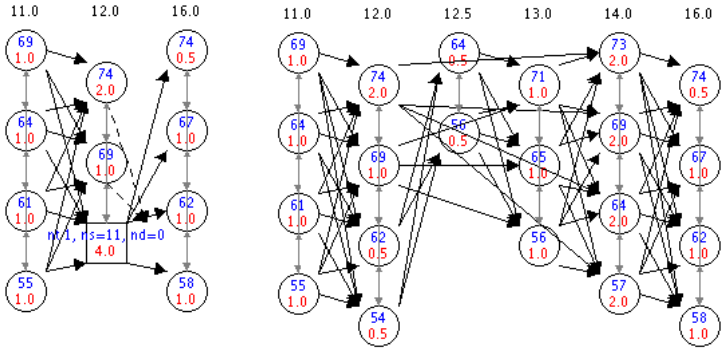
Production rule 2: PR2 nt 0 nonSeqBagIRTrie_DiatonicAbsPitch_Grp Orig(0)



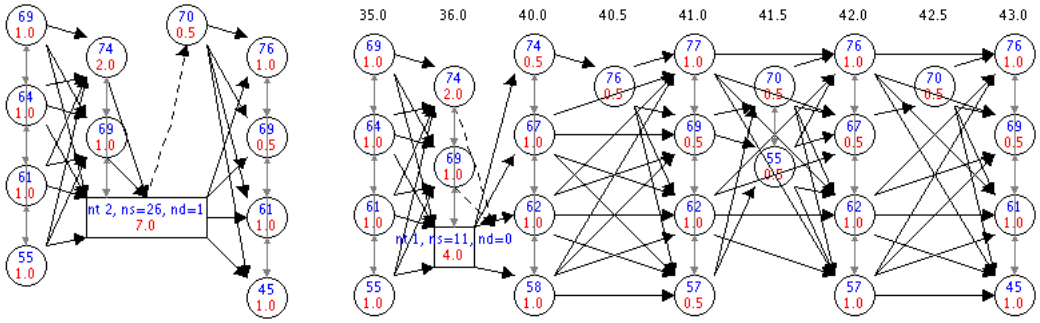
Production rule 3: PR3 nt 1 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



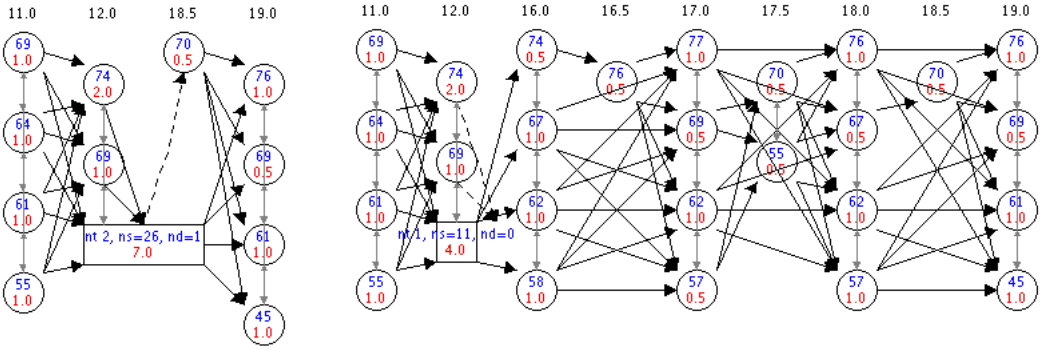
Production rule 4: PR4 nt 1 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



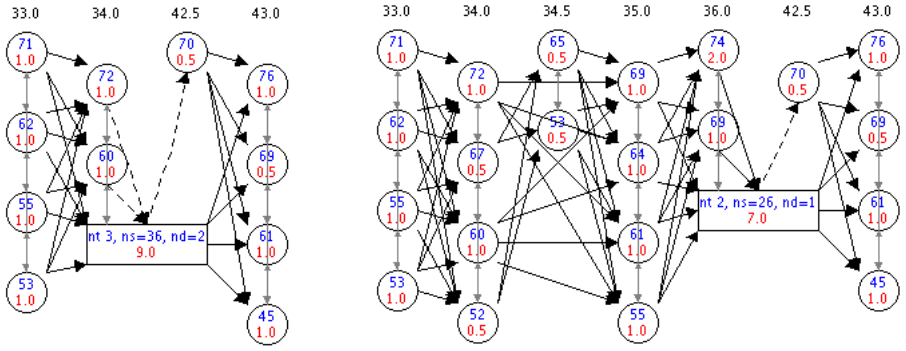
Production rule 5: PR5 nt 2 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



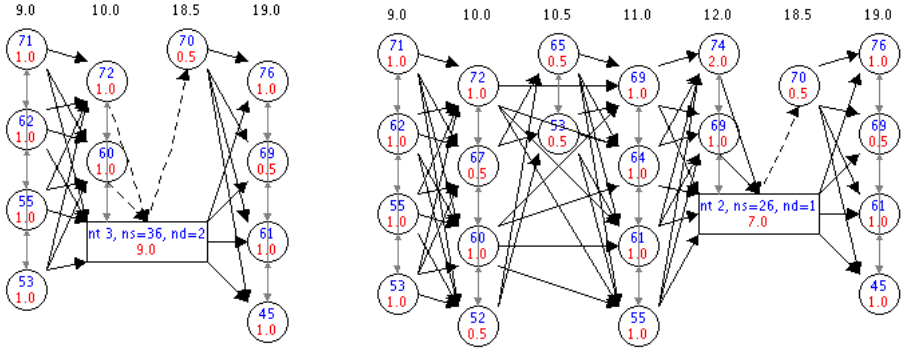
Production rule 6: PR6 nt 2 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



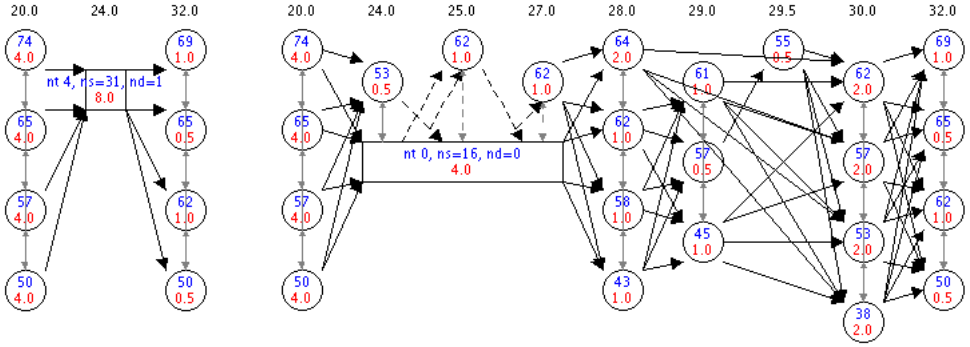
Production rule 7: PR7 nt 3 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



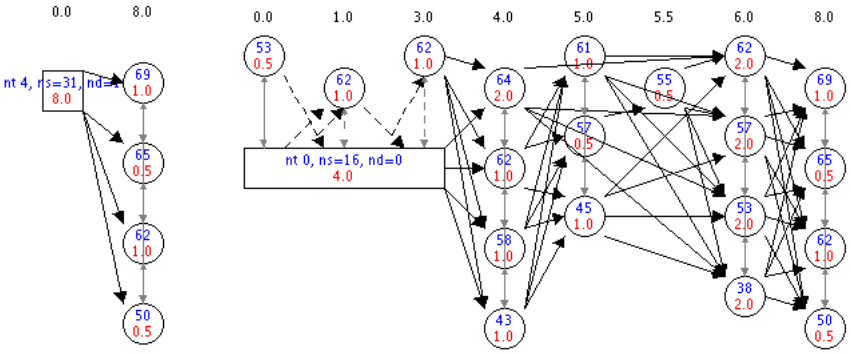
Production rule 8: PR8 nt 3 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



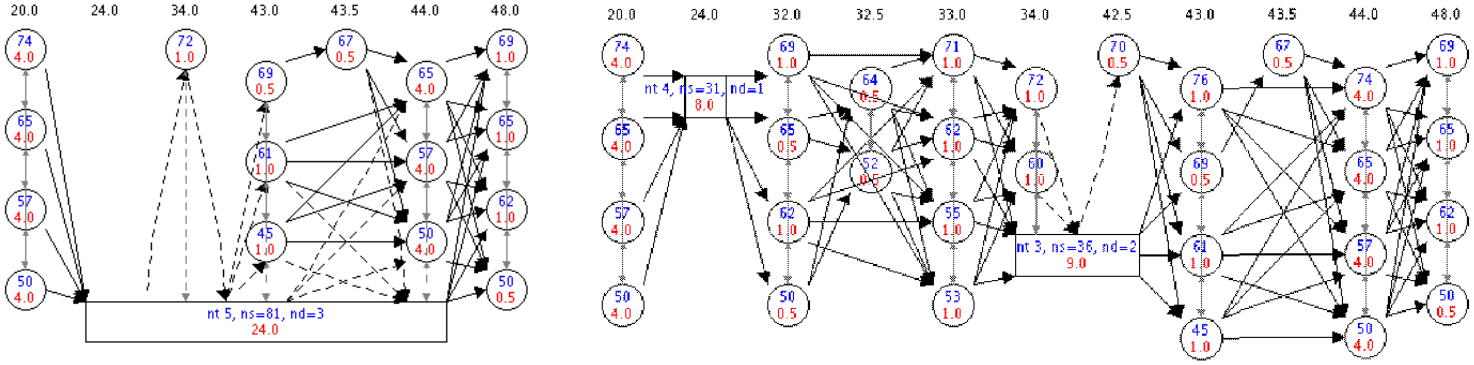
Production rule 9: PR9 nt 4 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



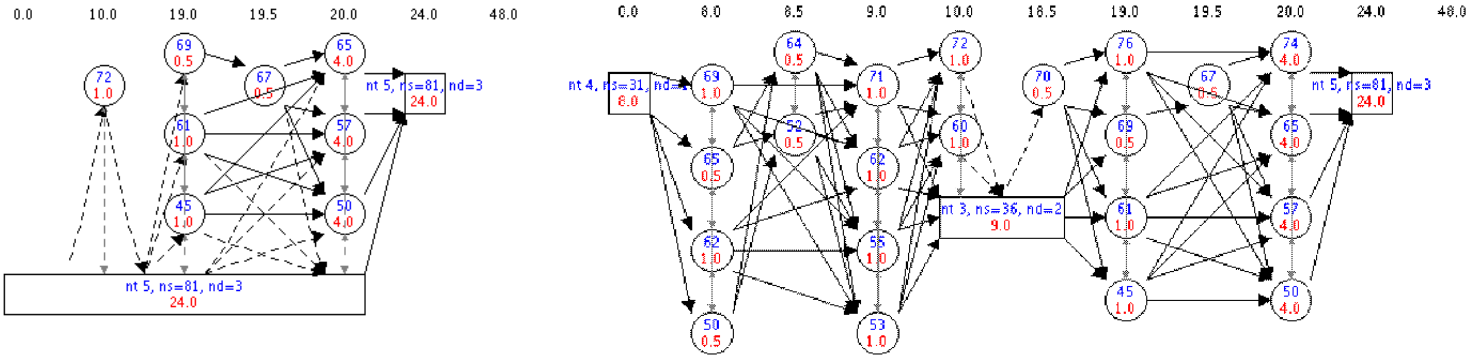
Production rule 10: PR10 nt 4 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



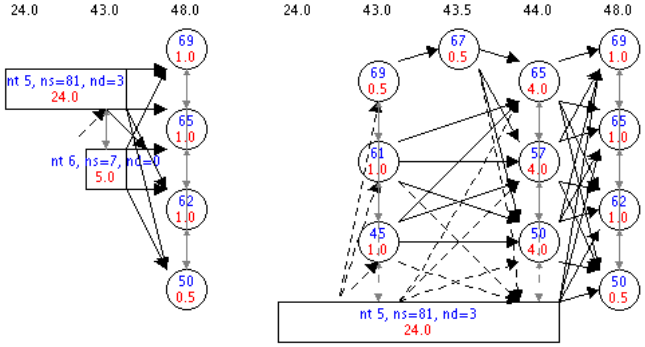
Production rule 11: PR11 nt 5 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



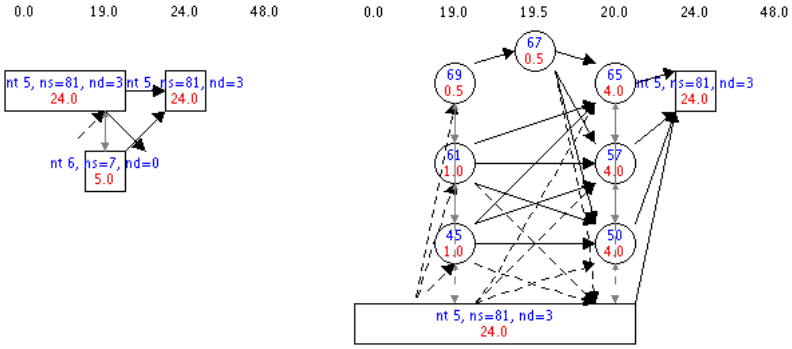
Production rule 12: PR12 nt 5 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



Production rule 13: PR13 nt 6 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



Production rule 14: PR14 nt 6 nonSeqBagIRTrie_DiatonicInt_Grp Orig(0)



References

[Bal92] Mira Balaban. *Understanding Music with AI: Perspectives on Music Cognition, Eds.: M.Balaban, K.Ebcioglu, O.Laske*, chapter Music structures: interleaving the temporal and hierarchical aspects in music, pages 110–38. Cambridge Mass., Menlo Park & London: AAAI Press & MIT Press, 1992.

[Bel92] Bernard Bel. *Understanding Music with AI: Perspectives on Music Cognition, Eds.: M.Balaban, K.Ebcioglu, O.Laske*, chapter Symbolic and Sonic Representations of Sound-Object Structures, pages 64–109. Cambridge Mass., Menlo Park & London: AAAI Press & MIT Press, 1992.

[BK92] Bernard Bel and Jim Kippen. *Understanding Music with AI: Perspectives on Music Cognition, Eds.: M.Balaban, K.Ebcioglu, O.Laske*, chapter Bol processor grammars, pages 366–413. Cambridge Mass., Menlo Park & London: AAAI Press & MIT Press, 1992.

[BRBM78] William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, 2(4):10–20, 1978.

[Bri90] Alexander R. Brinkman. *Pascal Programming for Music Research*. The University of Chicago Press, Chicago, 1990.

[Cam00] Emilios Cambouropoulos. Extracting 'significant' patterns from musical strings. *Invited Talk presented at London String Days*, 2000.

[Che02] Marc Chemillier. *Informatique musicale, Traité IC2, Eds.: J.-P. Briot, F. Pachet*, chapter Grammaires, automates et musique. Hermès, Paris, 2002.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[Con02] Darrell Conklin. Representation and discovery of vertical patterns in music. In Smaill Anagnostopoulou, Ferrand, editor, *Proceedings of Second International Conference on Music and Artificial Intelligence, ICMAI 2002.*, Edinburgh, Scotland, UK, September 2002. Springer.

[Cop91] David Cope. *Computers and Musical Style*. Oxford University Press, 1991.

[Cop01] David Cope. *Virtual Music: Computer synthesis of musical style*. MIT Press, 2001.

[CW95] Darrell Conklin and Ian Witten. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24:51–73, 1995.

[Dan93] Roger Dannenberg. A brief survey of music representation issues, techniques, and systems. *Computer Music Journal*, 17(3):20–30, Fall 1993.

[FB98] Hoda Fahmy and Dorothea Blostein. A graph-rewriting paradigm for discrete relaxation: Application to sheet-music recognition. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 12(6):763–799, 1998.

[HM87] Lilia Hess and Brian H. Mayoh. Graphics and their grammars. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*, volume 291 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1987.

[Hol80] Steven R. Holtzman. A generative grammar definition language for music. *Interface*, 9:1–48, 1980.

[KH02] Tatsuya Aoyagi Keiji Hirata. Representation method and primitive operations for a polyphony based on music theory gttm. *IPSJ Journal*, 2002.

[LJ83] Fred Lerdahl and Ray Jackendoff. *A Generative Theory of Tonal Music*. MIT Press, 1983.

[Mar97] John C. Martin. *Introduction to languages and the theory of computation*. McGraw-Hill International Editions, 2nd edition edition, 1997.

[Mar00] Alan Marsden. *Representing Musical Time - A Temporal-Logic Approach*. Studies on new new music research (Ed. Marc Leman). Swets & Zeitlinger, Lisse, The Netherlands, 2000.

[Mit01] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 2001.

[Moo72] James A. Moorer. Music and computer composition. *Communications of the ACM*, 15(2):104–113, February 1972.

[Moz70] Wolfgang A. Mozart. *Zwölf Variationen in C, KV 265, "Ah, vous dirai-je maman"*. Nagels Verlag, Kassel, 1970.

[MSJ92] Benjamin O. Miller, Don L. Scarborough, and Jacqueline A. Jones. *Understanding Music with AI: Perspectives on Music Cognition, Eds.: M.Balaban, K.Ebcioglu, O.Laske*, chapter On the Perception of Meter, pages 428–447. Cambridge Mass., Menlo Park & London: AAAI Press & MIT Press, 1992.

[Ock91] Adam Ockelford. *Representing Musical Structure, Eds.: Peter Howell, Robert West and Ian Cross*, chapter 4: The Role of Repetitions in Percieved Musical Structures, pages 129–160. Academic Press, 1991.

[Roa79] Curtis Roads. Grammars as representations for music. *Computer Music Journal*, 3(1):48–55, 1979.

[SF97] Eleanor Selfridge-Field, editor. *Beyond MIDI, The Handbook of Musical Codes*. The MIT Press, 1997.

[Sim62] Herbert A. Simon. The architecture of complexity. In *Proceedings of the American Philosophical Society, Vol. 106*, pages 467–482, Philadelphia, Pa, 1962.

[Sis01] Elaine Sisman. *'Variations' In The New Grove Dictionary of Music and Musicians, Ed. Stanley Sadie.* Macmillan Publishers Limited, London, 2001.

[Slo85] John A. Sloboda. *The Musical Mind. The cognitive psychology of music.* Clarendon Press, Oxford, 1985.

[SS68] Herbert Simon and Richard Sumner. *Formal Representation of Human Judgment, Ed.: Benjamin Kleinmuntz*, chapter 8. Pattern in music, pages 219–251. John Wiley & Sons, 1968.

[Ste84] Mark J. Steedman. A generative grammar for jazz chord sequences. *Music Perception*, 2(1):52–77, Fall 1984.

[SWH93] Alan Smaill, Geraint Wiggins, and Mitch Harris. Hierarchical music representation for analysis and composition. *Computers and the Humanities*, 27:7–17, 1993.

[Tem01] David Temperley. *The Cognition of Basic Musical Structures.* MIT Press, Cambridge, Massachusetts, 2001.

[TW99] Peter M. Todd and Gregory M. Werner. *Musical Networks: Parallel Distributed Perception and Performance, Eds. N.Griffith and P.M.Todd*, chapter Frankensteinian Methods for Music Composition, pages 313–339. Bradford Books, MIT Press, 1999.

[WHC91] Robert West, Peter Howell, and Ian Cross. *Representing Musical Structure, Eds.: Peter Howell, Robert West and Ian Cross*, chapter 1: Musical Structure and Knowledge Representation, pages 1–30. Academic Press, 1991.

[Whi01] Arnold Whittall. *'Form' in The New Grove Dictionary of Music and Musicians, Ed. Stanley Sadie.* Macmillan Publishers Limited, London, 2001.

[Win68] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, Vol.12:2–49, Spring 1968.

[ZM02] David B. Vogel Zbigniew Michalewicz. *How to Solve It: Modern Heuristics.* Springer, 2002.