A Logical Approach to Comparison of Music

Michael Lunøe

Kongens Lyngby 2010

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-BSc: ISSN 0909-3192

Abstract

The thesis establishes the theory of Constraint Programming (CP) and playlists. It applies techniques of CP, logic and functional programming with a similarity function between music pieces to build an Automatic Playlist Generator. The product of the thesis is a program in SML that generates playlists from a users query of suggestions and banning of songs. The similarity function is build solely on measures in tempo and key, which results in playlists that are somewhat useless. The program lack of measures in timbre, rhythm and melody, but is left open for the implementation of these. The thesis fianlly concludes that the techniques of CP and local search proves efficient for solving the problem.

Resumé

Projektet præsenterer den grundlæggende teori bag Constraint Programming (CP) og playlists. Strategien i arbejdet med constraints, logisk og funktionel programmering er anvendt til at opstille en Automatic Playlist Generator, som også benytter sig af en funktion til at sammenligne musiknumre. Produktet af projektet er et program i SML der genererer playlists ud fra forslag og forbud på sange fra et bibliotek. Funktionen til sammenligning er alene baseret på sammenligning imellem tempo og toner (key), som resulterer i ubruglige playlists. Programmet mangler sammenligninger imellem klangfarve (timbre), rytme og melodi, men den dynamiske tilgang gør at det let kan implementeres. Til sidst sluttes af den anvendte CP og local search tilgang viser sig effektiv til at løse problemet.

Preface

Time spent on assembeling playlists for certain purposes often exceeds the actual use. Some automatic playlist generation engines already exists, but they often lack the possibility of altering the generated playlist - or the option of specifying the length. The result is a repeated query for a playlist until one satisfies and another when it runs out of songs to play.

This thesis is the product of a personal need for an Automatic Playlist Generator.

The thesis is produced under the Department of Informatics and Mathematical Modelling at the Technical University of Denmark and it will presume some knowledge of logic programming and functional programming as it will include an implementation of a program in each language. The preconditions for the thesis are the courses 02156 Formal Logical Systems and 02157 Functional programming teached at the Technical University of Denmark.

Lyngby, June 2010

[illegible]

Michael Lunøe

Acknowledgements

I would like to thank my family and friends for support and review of the thesis. I would also like to thank my counsellor, Jørgen Villadsen from the Department of Informatics and Mathematical Modelling at the Technical University of Denmark, for giving me room for my own definition of the thesis. Also for review of the process and guidance in the right direction.

Contents

Chapter 1

Introduction

Constraint Programming have successfully been applied to many problems. Music similarity have within the area of Music Information Retrieval been discussed vastly. Applications of music similarity include many different areas. Playlist generation is one, that have recieved much attention. The reason could be the resemblance to puzzles, which makes it applicable in many approaches. This project will focus on the application of Automatic Playlists Generation (APG) using Constraint Programming.

> With the aid of cost affordable storage and greater device inter-connectivity, a listener's personal music collection is capable of growing at an extraordinary rate. When faced with such large music collections, listeners can often become frustrated when trying to select their music. Hence, it becomes increasingly difficult for a listener to find music suited for a particular occasion [Reynolds *et al.*, 2007].

The goal with this project is to see if Constraint Programming is applicable to the problem of APG and if this approach is efficient. The problem formulation of the project is stated in the next section.

1.1 Problem description

The need for an automated filtering of music has become bigger the resent years due to vast music libraries.

With focus on efficiency and logic a Constraint Programming (CP) approach to the problem of Automated Playlist Generation (APG) is adressed, thus the project will be implemented in a declarative programming language.

This APG will include a representation of the problem as a Constraint Satisfaction Problem (CSP) and an application using CP.

An objective of the project is to apply a similarity function to the program in order to improve the quality of the generated playlists.

The project will abstract from analysis of sound and will presume some information on music available.

1.2 Structure

After an introduction to the subject of APG the problem is approached. Basic playlist theory, problems that can be solved by CP, algorithms that implement CP. And the application to the problem will be presented in chapter 2. To utilise algorithmic choices an analysis of the strength and weaknesses of CP languages will be conducted in chapter 3. The use of constraints to represent the problem in a declarative domain places demands of decisions about the design and implementation of the system. The chapter will therefore also include a representation of the problem and further algorithmic choices tied to this decision along with the choice of a programming language. In chapter 4 the overall design and structure of the program will be presented along with a presentation of the specific functions in the program, thus forming the solution to the problem.

In chapter 5 test results and analysis of complexities is presented and evaluated. A discussion of the results is done in same chapter and finally the project is concluded and the future prospects are discussed in chapter 6.

1.3 Music theory

To be able to understand the background of choices in playlist generation the basic theory of music must be introduced. The section begins with an introduction to basic technical terms within the area of music and describes the features as a quality of the digital signal of music.

Rhythm Any regular recurring motion, symmetry [1].

Tempo The speed of a given piece of music [2].

Beat The basic time unit of music [3].

Tone A specific pitch that represents the perceived fundamental frequency of a sound [4].

Timbre The quality of a musical note or sound or tone that distinguishes different types of sound production, such as voices or musical instruments [5].

Chord Any set of harmonically-related notes that is heard as if sounding simultaneously[6].

Harmony Two or more notes played simultaneously to produce a chord [7].

Key A hierarchical scale of musical notes on which a composition is based[8].

Melody A linear succession of musical tones which is perceived as a single entity [9].

To understand music in terms of these descriptions further it is helpfull to study the digital understanding. As a digital signal a tone can be described by the frequency of waves, which can be seen as both a vertical and horisontical quality of the the digital signal. Rhythm is, on the other hand, a strictly horizontal quality of the digital signal, because it is described by when the tones occur rather than which. This is, however, not a sufficient description. Tones has to be regularly recurring, meaning that patterns of occurances is repeated. This vauge definition makes it very complex to deduce from a digital signal because every instrument (e.g. voice, guitar, drums, etc.) can follow its own pattern generating a new pattern combined. It is, though,

[1]`http://en.wikipedia.org/wiki/Rhythm`
[2]`http://en.wikipedia.org/wiki/Tempo`
[3]`http://en.wikipedia.org/wiki/Beat_(music)`
[4]`http://en.wikipedia.org/wiki/Pitch_(music)`
[5]`http://en.wikipedia.org/wiki/Timbre`
[6]`http://en.wikipedia.org/wiki/Chord_(music)`
[7]`http://en.wiktionary.org/wiki/harmony`
[8]`http://en.wiktionary.org/wiki/key`
[9]`http://en.wikipedia.org/wiki/Melody`

possible to deduce patterns from digital signal processing [Foote *et al.*, 2002]. The tempo in modern music is usually indicated in beats per minute (bpm), i.e. the speed of the music piece. As rhythm, tempo is a horisontical quality of the digital signal and can be deduced from digital signal processing as well [Foote *et al.*, 2002].

The timbre is characterised by many different qualities of the digital signal, e.g. the rise, duration, and decay of the sound, and it is therefore possible to deduce from digital signal processing. Global timbre refers to the timbre description covering the full duration of a composition [Reynolds *et al.*, 2007].

Because harmony operates on *which* tones that sound it can be viewed as a vertical quality of the digital signal. In figure 1.1 is specified which keys, and therefore also tones and chords, harmonise. The keys are in the figure represented by symbols because further explaination of keys is beyond the scope of this project [10].

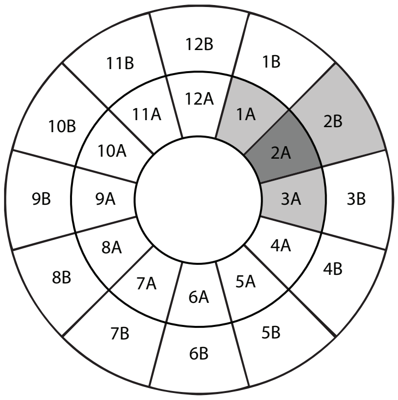


Figure 1.1: The figure shows the circle of fifths with symbols of 1A to 12B to represent the keys. An A means that it is a minor key and B major. For two keys to be harmonic they must be the same or right beside it, e.g. 2A is harmonic with 1A, 2A, 3A and 2B.

Because the key specifies which tones or chords that may be used in the composition of a music piece the digital signal processing can deduce the global key(s), i.e. the key of the whole composition [Anglade *et al.*, 2009].

Lastly the melody is described by both which tones and when they sound. This means that a melody contains a rhythm and for the melody to sound good, it must consist of harmonic succeeding tones or chords.

[10] Further explaination is given at Wikipedia, `http://en.wikipedia.org/wiki/Key_(music)`.

Chapter 2

Theory

This chapter establishes the basic playlist theory and the principles of Constraint Porgramming (CP). It will function as the knowledge base on which the project is buildt and introduce the methods and tools to solve the problem along with the discussion in chapter 3.

2.1 Playlist theory

To be able to compose a playlist of good quality it is necessary to study what is considered to be good charateristics of a playlist and which methods have proven effective in a sense of song selection. This will be presented in this section, but first a research on the use and purpose of playlists is conducted.

> Some playlists are created for personal use by oneself or a few close friends - primarily as background for another activity [...] A playlist may be created to reflect a particular mood or emotion in the creator [...] A playlist might also be shared as "party music", in this context mainly as background rather than as dance music or the center focus for the party [Cunningham *et al.*, 2006].

The assignment is to provide a series of songs, that go great together and consists of songs in the mood and/or activity of the users choice. To know what the user wants, it is crucial to gather information from the user. The task of conducting a suitable playlist is limited by manually gathered information and efficiency, because the user is interested in a fast solution, without much effort. Some automatised information gatherings go as far as measuring the tempature, weather and noise in the environment, where the playlist is needed [Reynolds *et al.*, 2007]. Others mainly rely on manually provided information [Pauws *et al.*, 2006].

A good balance between automatised information gathering and easy information retrieval from the user is needed to fulfil these demands and yield good playlist results. *Relevance feedback* is a technique to improve the quality of the returned playlist in reponse to a user's query by incorporating feedback from the user [Logan, 2002]. The use of relevance feedback in playlist generation has in many respects proven to be a good solution [Logan, 2002]
[Pampalk and Gasser, 2006][Pampalk *et al.*, 2005][Reynolds *et al.*, 2007]. A common approach to get immediate information of what the user wants is the use of *seed songs*, i.e. a song that represents the type of music the user wants, to listen to and constitutes the basis of the playlist.

Many different approaches have been researched with respect to meeting the requirement of good playlists, but they all use some sort of similarity function between songs.

> There are various measures for different aspects of similarity functions in the litterature [...]: melodic and timbral measures have generally received the most attention, but rhythmic and harmonic ones have also been considered, and metadata such as artist, lyrics, year of release, sales figures, chart position and label classification may also be examined [Allan *et al.*, 2007].

In addition to the, by [Allan *et al.*, 2007], suggested similarity measures in tempo, genre and duration has been suggested [Reynolds *et al.*, 2007][Pauws *et al.*, 2006]. To decrease and specify the music collection before engaging in an actual playlist generation *collaborate filtering* is commonly used. Collaborate filtering is a community process, as it employs a multi-user approach that uses explicit preference to match songs to a specific user [Reynolds *et al.*, 2007]. But for this to work the system has to have access to a music community.

The next section will present the theory of Constraint Programming.

2.2 Constraint Programming (CP)

This section introduces the concept of CP and Constraint Satisfaction Problems (CSP). It serves as background knowlegde for working with and understanding CP. The indtroduction to CSP will refer to [Russell and Norvig, 2003] and [Apt, 2006].

To classify if a problem is suited for CP a definition of a CSP is needed.

Definition 2.1 A Constraint Satisfaction Problem (or CSP) is defined by the 3-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is the set of variables, $\mathcal{D}$ is the corresponding set of domains and $\mathcal{C}$ is the set of constraints on the variables. Each variable, x_i, in $\mathcal{X}$ has a corresponding domain, D_i, of possible values, $v_i \in D_i$. A state of the problem is defined by the n-tuple of variables, $(x_1, \ldots, x_n)$, where some or all variables are assigned noted by $x_i = v_i$. A complete assignment is the assignment of every variable in the n-tuple, i.e. an element of the cartesian product between domains $D_1 \times \ldots \times D_n$. Let A be a complete assignment, $A = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ of n variables and c_i be one of k combinations of m variables from the assignment A, $c_i = (d_{i_1}, \ldots, d_{i_m}) \in D_{i_1} \times \ldots \times D_{i_m}$. Every element d_{i_j} from c_i is therefore contained in A. If all those k combinations of variables formed by c_i is contained in the constraint C, then A is a solution to C. C is said to be a constraint on the variables $x_{i_1}, \ldots, x_{i_m}$. This can be expressed by

$$\begin{aligned} A = (d_1, \ldots, d_n) &\in D_1 \times \ldots \times D_n \\ c_i = (d_{i_1}, \ldots, d_{i_m}) &\in D_{i_1} \times \ldots \times D_{i_m},\ where\ d_{i_j} \in A \\ c_i \in C &\subseteq D_{i_1} \times \ldots \times D_{i_m},\ where\ i \in [1;k],\ m \in [1;n] \end{aligned}$$

If the size of the tuple yielded by c_i is 1, i.e. $m = 1$ then C is said to be unary. If $m = 2$, C is said to be binary and global if it contains every element of the assignment, i.e $m = n$. An assignment, that fulfils the above for every $C \in \mathcal{C}$, i.e. does not violate any constraints, is said to be a solution to the CSP. If there exists a solution to the CSP, it is said to be consistent, otherwise inconsistent [Russell and Norvig, 2003, p. 137][Apt, 2006, p. 9].

This definition seems to leave the problem very open, so that many problems can be modelled to a CSP, but it is not all for which it is effective to solve this way. Also there are different approaches to solve the problem, but they all rely on the same structure - they are general to solving CSPs. In operating with CSPs some problems needs an optimal solution, for this are the formulation of Constraint Optimisation Problems.

Definition 2.2 A Constraint Optimisation Problem (COP) is a subset of CSPs and is defined by the 4-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{O})$, where the three first elements are defined as in a CSP in definition 2.1 and $\mathcal{O}$ is an objective (or cost) function, that determines the quality of a current state.

$$S \rightarrow \mathbb{R} \in \mathcal{O}$$

S is the set of solutions to the COP and $\mathbb{R}$ is the set of real numbers [Apt, 2006, p. 43].

A CSP can be visualised by a constraint graph where vertexes in the graph corresponds to variables of the CSP and edges corresponds to constraint relations. In the following are two examples of CSPs that should help to get a notion of working with CSPs. The two examples each have properties that can relate to the problem of Automatic Playlist Generation. Constraint graphs are explained for them both.

2.2.1 Cryptarithmetic puzzle

The classic example of an CSP is the cryptarithmetic puzzle, where symbols are replaced with digits for an equation to make sense. When these problems deals with valid sums it is referred to as a alphametic problem.

In the following letters from the alphabet are the symbols to replace with digits, so it satisfies the sum.

$$\begin{array}{r} SEND \\ \underline{+MORE} \\ MONEY \end{array}$$

The variables are $\mathcal{X} = \{S, E, N, D, M, O, R, Y\}$ and the corresponding domains are seen below:

$$\mathcal{D} = \left\{ \begin{array}{cc} S \in [1;9], & M \in [1;9], \\ E \in [0;9], & O \in [0;9], \\ N \in [0;9], & R \in [0;9], \\ D \in [0;9], & Y \in [0;9] \end{array} \right\}$$

The values of S and M are leading digits and are therefore further restricted to being a non-zero integer. The problem is formulated as an equality constraint where every $x \neq y$ for $x, y \in \{S, E, N, D, M, O, R, Y\}$ or as it is often represented $\texttt{all\_diff}(S, E, N, D,$-$M, O, R, Y)$. The constraints are formulated as the following.

$$\mathcal{C} = \left\{ \begin{array}{r} 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ +1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y, \\ \texttt{all\_diff}(S, E, N, D, M, O, R, Y) \end{array} \right\}$$

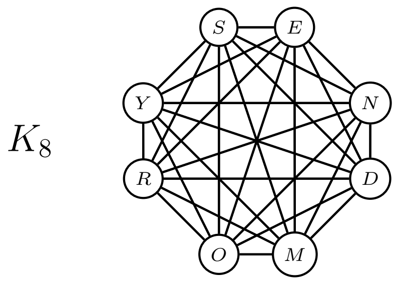


Figure 2.1: The figure shows the constraint graph for the *SEND MORE MONEY*-problem. The number of constraint bindings or edges in the complete graph is $|E_{K_n}| = \frac{n(n-1)}{2} \Rightarrow |E_{K_8}| = \frac{8(8-1)}{2} = 28$ [Nielsen, 1995].

The corresponding constraint graph is in figure 2.1.

In stead of one big equality the problem can be divided into smaller subproblems, which simplifies the process of solving it. The division introduces however new variables, that carries a value from one column to the next, and with that altered and additional constraints. In fact, every high-order constraint with a finite domain can be reduced to binary constraints if enough auxiliary variables are introduced [Russell and Norvig, 2003].

$$\mathcal{C} = \left\{ \begin{array}{rcl} D + E & = & Y + \alpha_1 \cdot 10, \\ \alpha_1 + N + R & = & E + \alpha_2 \cdot 10, \\ \alpha_2 + E + O & = & N + \alpha_3 \cdot 10, \\ \alpha_3 + S + M & = & O + \alpha_4 \cdot 10, \\ \alpha_4 & = & M \end{array} \right\}$$

The domain of each letter remains and the domain of every carry is, $\alpha_i \in [0; 1]$. This formulation of the problem is equivalent to the calculation method of sum by hand. The constraint graph for this problem is much more complex and a constraint hyper graph aids the visualisation, see figure 2.2. Each constraint is in the hyper graph a square box connected to the varables that it constrains.

Because every constraint does not include every variable it can be viewed as a composite problem, see figure 2.3.

The tree decomposition of the constraint graph can be helpful to get an overview of the problem. Each subproblem can be solved individually and the consequence of this can be transferred to the next subproblem until the whole problem has been solved. It is a good visiualisation of where lazy evaluation can be used, because subproblems can be ”left open” until a solution of these is needed.

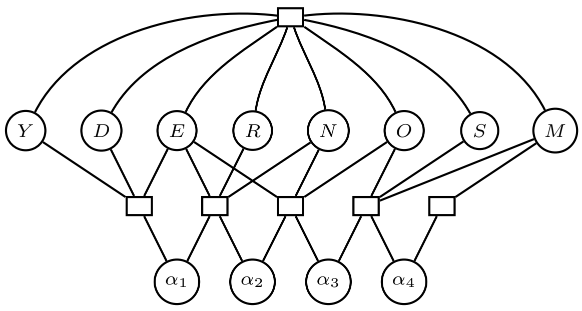


Figure 2.2: Hyper graph for the carry representation of the *SEND MORE MONEY*-problem. Each square is a constraint and each edge from it a relation to a variable.

The solution to the *SEND MORE MONEY*-problem is

$$\begin{array}{r} 9567 \\ \underline{+1085} \\ 10652 \end{array}$$

There exists only one solution to the problem, which can be shown by a search tree [Apt, 2006, p. 11][Russell and Norvig, 2003, p. 140].

2.2.2 The n-Queens problem

The goal of the n-Queens problem is to place n queens on a n-size chessboard so that no queen attacks another, i.e. the diagonals, rows and columns are free for every queen. The problem can be represented as a CSP by the following.

$$\begin{aligned} \mathcal{X} &= \left\{x_1, \ldots, x_n\right\}, \\ \mathcal{D} &= \left\{x_1 \in [1;n], \ldots, x_n \in [1;n]\right\}, \\ \mathcal{C} &= \left\{ \begin{array}{c} \texttt{all\_diff}(x_1, \ldots, x_n), \\ x_i - x_j \neq i - j \ for \ i \in [1;n-1] \ and \ j \in [i+1,n], \\ x_i - x_j \neq j - i \ for \ i \in [1;n-1] \ and \ j \in [i+1,n] \end{array} \right\}. \end{aligned}$$

The variabels, $\mathcal{X}$, consists of a list of variables of numbers. Each position in the list represents the horisontical position and the value represents the vertical position, hence the domain of each variable x_i is $[1;n]$. The vertical alignment is an implicit constraint of the position in the variable list, the `all_diff`() constraint constrains the horisontical alignment and the two last constraints handles the diagonals. The

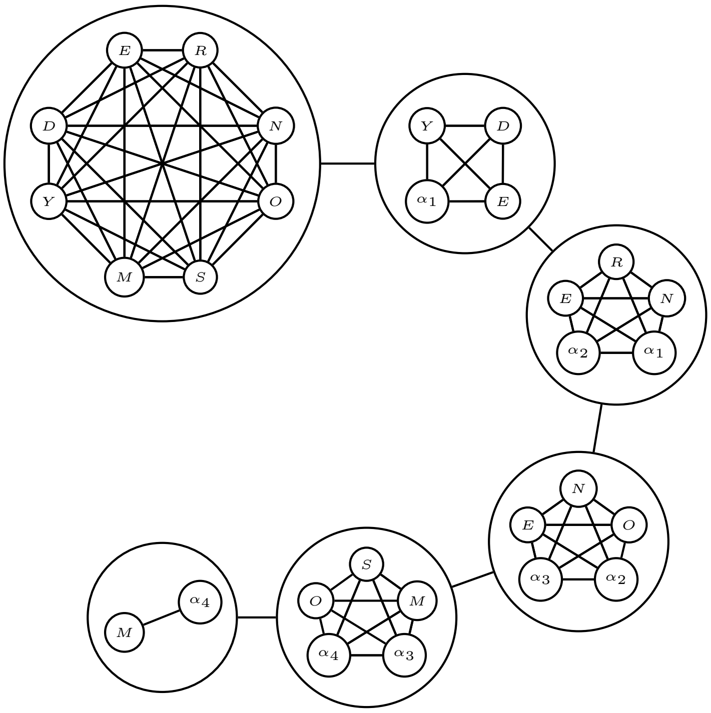


Figure 2.3: A tree decomposition of the constraint graph for the carry representation of the *SEND MORE MONEY*-problem. Each vertex consists either of a subproblem or a variable and each edge a constraint. The constraints that connect subproblems constrains mutual variables of the subproblems [Russell and Norvig, 2003, p. 154].

constraint graph is the complete graph, where each variable is constrained by the others in three different ways, i.e. every constraint in the CSP constrains all variables.

A solution to the 4-Queens problem is given in figure 2.4. The domain for each value is listed and the underlined values are chosen values.

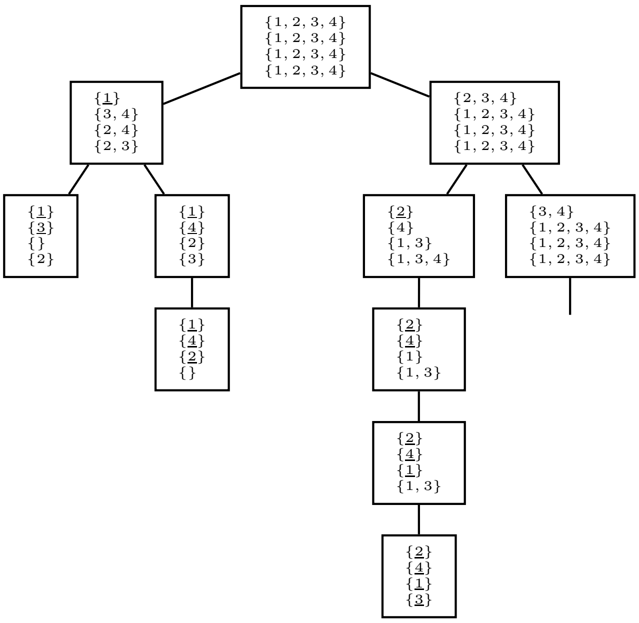


Figure 2.4: The search tree for solution of the 4-Queens problem with domains for each variable. The underlined values are chosen values for the variables. Each value corresponds to a vertical position and each position of domains consists of the horisontal position. The search is stopped with the first discovery of a solution. If all solutions is sought the search would continue with the leftmost node indicated by the unended edge.

There exists only solutions for $n = 1$ or $n \geq 4$, but no expression for the number of solutions to a given n. The n-Queens problem can also be formulated as a COP, where a simple objective function could be the number of constraint violations in the current state [Apt, 2006, p. 13].

2.3 CP heuristics

A heuristic function is in CP defined as a function that answers one or more of the following questions.

1. Which variable to select.
2. Which value to select.
3. Which constraint to split.
4. What are the implications of the current variable assignments for the other unassigned variables.
5. When a path fails - that is, a state is reached in which a domain is empty - can the search avoid this failure in subsequent paths
[Apt, 2006, p. 64][Russell and Norvig, 2003, p. 143].

Number 3 and 4 is the result of *Split* and *Constraint Propagation*, respectively. A Split is a division of a CSP into two or more CSPs over constraints or domains. A property of this split is that the union of the new CSPs has to be equivalent to the original problem. An example of a Split over constraints is given in section 2.2.1, by the decomposition of the constraint graph of the *SEND MORE MONEY*-problem and over domain the so called *enumeration* of variables in the solution to the 4-Queens problem in figure 2.4. The enumeration of variables splits the CSP into two equivalent problems over a variable's domain. The current variable is in the first case given a possible value from the domain and removed from the domain in the other. Continuing this proces, until the domain is empty, generating a new CSP for every value before proceeding, is called *labeling.* The branching of the tree would then be the number of possible values in the domain of the current variable [Apt, 2006, p. 62][Russell and Norvig, 2003, p. 146].

Constraint Propagation is also used in the 4-Queens solution. Constraint Propagation on a variable is the appliance of knowledge from the assignment of another variable onto the domain of this. This means that if the assignment of a variable removes the possibility of some values to others in the solution, the domains of these variables can be reduced. An example of this is *arc consistency.* If every constraint is represented by an arc, as it is done in the constraint graph, just directed, arc consistency is when there for every value exists some value that it is consistent with. This property is used in figure 2.4 [Apt, 2006, p. 66 and 138].

2.4 Generic solve procedure for CP

To solve a CSP a generic procedure, determining the processes a CSP shall undergo for it to be solved, can be helpful. Such a procedure is presented in this section.

Heuristics that can be applied to a CSP was presented in the previous section, where two processes were specific to CP and is computed directly in the generic procedure, see figure 2.5.

```
SOLVE(csp) returns a solution or failure
    inputs:
        csp, a constraint satisfaction problem

    continue ← TRUE
    current ← csp
    while continue and not SOLUTION(current) do
        PREPROCESS(current)
        CONSTRAINT PROPAGATION(current)
        if not SOLUTION(current)
            if ATOMIC
                continue ← FALSE
            else PROCEED BY CASES(SPLIT(current))
        else return current
        return failure
```

Figure 2.5: The generic procedure SOLVE() for solving a CSP. The SOLUTION() is a goal test function and the PREPROCESS() is a procedure to bring the CSP to a desired form. The CONSTRAINT PROPAGATION() uses information of the current assignments to restrict values of others and ATOMIC() is a test if search does not need to proceed, that is the outcome of the CSP (or sub-CSP) can be directly computed. The SPLIT() divides the CSP into one or more sub-CSPs and PROCEED BY CASES() deals with the order of which sub-CSP to handle next [Apt, 2006, p. 59].

All processes in the generic procedure can be defined for every CSP. Not all are necessarily used, but can be considered when implementing the program.

Analysis of CP will after this chapter aid the algorithmic choices and choice of data-structure.

Chapter 3

Analysis

In this chapter an analysis of CP and a discussion of the appliance on Automatic Playlist Generation (APG) is conducted. A discussion of the implications by a representation of the problem in declarative languages will aid the choices in datastructures. Finally, a discussion on algorithms and heuristic functions to solve the problem will aid the algorithmic choices.

3.1 Algorithms for CSPs

When working with CSPs several specific algorithms exists to solve the problem. Backtracking search is a depth-first search that backtracks to the parent node, whenever there exists an empty domain for a variable. It then continues with the next descendant spanning the whole tree. Every node in the tree is a CSP and the algorithm stop with the first assignment that satisfies the CSP if a single solution or inconsistancy is sought. If every solution is wanted the algorithm continues untill every leaf has been generated. Because the search tree is generated without further information about the problem, than that given in the problem formulation, it is a uniformed search and therefore not expected to perform very well [Russell and Norvig, 2003, p. 73].

If, on the other hand, information is gathered and used while searching, the search can be improved. An example of this is the BRANCH AND BOUND search. The branch and bound search uses a heuristic function to bound the search. The branch and bound heuristic presumes an objective function, where it in each step, holds the current best assignment. This allows the search to prune the search tree, or bound the search, whenever the objective function yields a worse result than the current best. Another example of a heuristic function is the most constrained variable (MCV) heuristic. This function always selects the variable that appears in the largest number of constraints, to be assigned next. When used with backtracking, the performance can be greatly improved [Apt, 2006, pp. 337].

The enumeration and Constraint Propagation heuristics are both direct properties of the FORWARD CHECKING search. The algorithm used in figure 2.4 is forward checking. Forward checking uses propagation to detect inconsistency whenever a variable is assigned to a value, but it does not detect if the removal of values generates new inconsistancies. A stronger propagation is to repeatedly applying arc consistency until no inconsistancy is left. This is the property of the MAC (Maintaining Arc Consistency) algorithm. The worst-case time is $O(n^2d^3)$ for the total number of arcs n and the total number of values d in a problem [Russell and Norvig, 2003, p. 146]. Of cause arc consistency , or MAC, does not find every consistency. The arc consistency checks for inconsistency in any two adjacent variables. If the check is expanded to include the second adjacent variable it is referred to as *path consistency* [Apt, 2006, p. 150][Russell and Norvig, 2003, p. 147].

Finally we have the MIN-CONFLICTS algorithm. It is the result of the application of *local search* to CSPs. It uses the min-conflicts heuristic, i.e. choosing the value that results in a minimum number of conflicts. Its algorithm is shown in figure 3.1.

Local search are algorithms that do not care about which path they take to a solution, just that they find one. The MIN-CONFLICTS algorithm operates by moving to neighbors from a current state. It can be applied to both CSPs and COPs. In the latter the techniques of *hill climbing* and *simulated annealing* can be used to improve the search. Local search has proven very effective in solving many CSPs and COPs. For the MIN-CONFLICTS algorithm to work an initial complete assignment is needed, so it can operate on a current state. The efficiency is, of cause, depending on the initial state [Russell and Norvig, 2003].

In [Russell and Norvig, 2003, p. 143] a thorough survey on commonly used algorithms efficiency on commonly known CSPs is conducted and the results from the n-Queens problem is listed in table 3.1.

Table 3.1 shows that the min-conflicts local search is by far the most efficient algorithm for solving the n-Queens problem. The initial assignment could, with this problem, be randomly chosen or a greedy algorithm and the neighbor states could be

```
MIN-CONFLICTS(csp, max_steps) returns a solution or failure
    inputs:
        csp, a constraint satisfaction problem
        max_steps, the number of steps allowed before giving up
    current ← an initial complete assignment for csp
    for $i = 1$ to max_steps do
        if current is not a solution for csp
            return current
        var ← a randomly chosen,
              conflicted variable from VARIABLES[csp]
        value ← the value $v$,
                that minimises CONFLICTS(var, $v$, current, csp)
        set var = value in current
    return failure
```

Figure 3.1: The Min-Conflicts algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that choses a minimal conflict value for each variable in turn. The Conflicts function counts the number of constraints violated by a particular value, given the rest of the current assignment [Russell and Norvig, 2003, p. 151].

Problem	Backtracking	BT+MCV	Forward Checking	FC+MCV	Min-Conflicts
n-Queens	(> 40,000K)	13,500K	(>40,000K)	817K	4K

Table 3.1: Comparison of various CSP algorithms on the n-Queens problem. The algorithms from left to right, are simple backtracking, backtracking with most constrained variable (MCV) heuristic, forward checking, forward checking with MCV, and min-conflicts local search. Listed in each cell is the median number of consistency checks (over five runs), required to solve all n-Queens problems for n from 2 to 50; note that all entries are in thousands (K). Numbers in parentheses mean that no answer was found in allotted number of checks [Russell and Norvig, 2003, p. 143].

generated by moving a queen or swapping two queens. The conflicts function could return the number constraints violated by the assignment of a variable to a given value [Apt, 2006, pp. 372][Russell and Norvig, 2003, p. 150].

Before continuing the discussion of algorithmic choices a representation of the problem is needed.

3.2 Problem representation

On the basis of the theory presented in section 2.2 the problem of APG is formulated formally as a CSP. Because a song consists of a list of specific information each variable, s_i, consists of a vector where $s_{i,k}$ is k'th attribute. In addition the domain, d_i, is also represented by a vector specifying the domain of every attribute in s_i, $d_{i,k}$. On the other hand a constraint functions is defined as a relationship between attributes of the same type in variables, see equation 3.1.

$$\begin{aligned}
\mathcal{X} &= \left\{ s_0 = \begin{pmatrix} s_{0,0} \\ \vdots \\ s_{0,m} \end{pmatrix}, \dots, s_n = \begin{pmatrix} s_{n,0} \\ \vdots \\ s_{n,m} \end{pmatrix} \right\} \\
\mathcal{D} &= \left\{ s_0 \in d_0 = \begin{pmatrix} s_{0,0} \in d_{0,0} \\ \vdots \\ s_{0,m} \in d_{0,m} \end{pmatrix}, \dots, s_n \in d_n = \begin{pmatrix} s_{n,0} \in d_{n,0} \\ \vdots \\ s_{n,m} \in d_{n,m} \end{pmatrix} \right\} \\
\mathcal{C} &= \left\{ \begin{matrix} c_u = (s_{i,x}), \\ c_b = (s_{i,x}, s_{i+1,x}), \\ c_g = (s_{i_1,x}, \dots, s_{i_n,x}) \end{matrix} \right\} \\
\mathcal{O} &= \left\{ o_i = (s_i \to \mathbb{B}/\mathbb{R}) \right\}
\end{aligned} \tag{3.1}$$

In the equation $\mathbb{B}/\mathbb{R}$ is either a `boolean` or `real` depending on whether the problem is a COP or not, respectively. c_u is the set of unary constraints, c_b is the set of binary constraints and c_g is the set of global constraints. The unary and global constraints are trivial since they only constrains a single variable and the whole set, respectively. The unary constraints can be any of the variables in the playlist, but this does not tie the problem closer, since the problem can be handled by it self before considering any other problems. The binary constraints, on the other hand, ties the problem in a linked list because it is defined for every succeeding variable. The tree decomposition of the constriant graph of the APG problem is shown in figure 3.2.

This representation of the problem seems to support the local search algorithms if the global constraints can be propagated to the other domains. If this is done after

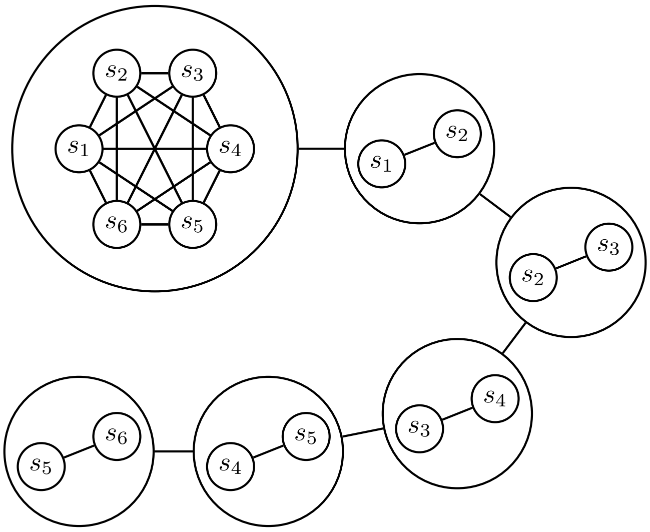


Figure 3.2: A tree decomposition of the constraint graph for the automatic playlist generation problem. Each vertex consists either of a subproblem or a variable. The complete graph of six vertexes models the global constraint sub-problem and the rest models the linked list of binary constraints.

assigning the first variable only the binary constraints needs to be fulfilled, which can be done step by step.

A resemblance of this problem to the n-Queens problem can be seen in the formulation of variables and domains. Each variable has a place in the list generating the implicit constraint that no other song can have the same position. Furthermore every variable has a domain consisting of the set of possible values and there is no difference between allowed values when no variable have been assigned jet, which is also equal to the n-Queens problem. The difference between the formulation of variables and domains of this problem and the n-Queens problem is that this contains one more dimension.

3.3 Algorithmic choices and datastructures

At this point the main algorithm of min-conflicts local search is chosen. This section will adress the subjects of intial assignment, the conflicts function and heuristics specific to the problem.

In choosing a beneficial initial assignment one must consider the structure of the constraints tied to the problem. As described above the problem is tied mostly by the binary constraints. This generates challenges to the possiblities of an initial assignment because every unary constraint only has effect on the other attributes of the same variable. This is because the unary constraint is very specific to a single attribute and thereby only very few variables can fulfil this demand generating implicit constraints on the other attributes. After the unary constraints have been applied, the globals can be applied, setting the boundaries for the rest of the variables.

The global constraints cannot be applied before the unary because they are defined as the overall spectrum of values allowed in a playlist and therefore needs a point of reference.

After this the binary constraints can be applied setting further boundaries for the variable in the list.

Inspiration from the n-Queens problem aids the choosing of the initial assignment. A random assignment and the greedy algorithm have been mentioned as suitors [Russell and Norvig, 2003, 151], but a genetic algorithm could also be an answer. The Genetic Algorithm works by matching and mutating an initial set of random assignments. It is inspired by the process of natural selection in genetics, hence the name. The initial population is matched by a fitness function that determines the pairwise compatibility - in the n-Queens problem this could be the number of nonattacking pairs queens. A crossover point is chosen randomly and “children” are produced by assembeling each “parent” parts (each parent can produce either one or two children, in the first case the best can be chosen). A small probability determines if a state is mutated. The mutations are again chosen randomly, in this case, by moving a queen to a random new place. The process is continued until a satisfactory state has been reached or enough time have elapsed. The process frequently takes large steps in the state space early in the process and smaller later, due to similarity in the state space. Because of the early large steps the algorithm can quickly produce a good proposition for the main algorithm [Russell and Norvig, 2003, pp. 116]. There is no guarantee that it performs well and the iterations needed multiplied by the cost of the functions together with the initial random assignment is not free.

In the case of a random initial assignment, that will be somewhat fast, the performance all depends on the main algorithm, that has to work harder to produce a solution to the problem.

In between these is the case of the greedy algorithm, that performs well and pull some of the weight from the main algorithm. There is just one problem that makes it complicated for the greedy algorithm. The greedy algorithm will built on problems, if not every assignment is a solution, which would be much to ask. Because the rest of the list is built on an erroneous state it will attract other errors. So the lists could very well, after one problem is missed, be assigned to wrong values.

The conflicts function could be consisting of the number of constraints violated, if the problem is represented as a CSP. On the other hand, a calculation of violation of each attribute represented in a variable, would give a very accurate match percentage if the problem is represented as a COP. See the problem representation, equation 3.1.

The choice of heuristic to choose value is obviously the min-conflicts heuristic, but as already mentioned the hill climbing and simulated annealing heuristics can be used, if the problem is chosen to be represented as a COP. Hill climbing algorithms comes in many different variants, but the basic hill climbing algorithm, often referred to as the greedy local search, operates by taking the best neighbor without further concern. A variant that in particular have is proven effective for the n-Queens problem is the Random-restart hill climbing. It operates with restarting a hill climbing at a random generated state, until a solution is reached, and can do so in approximately 25 steps. It solves the 8-Queens problem in less than a minute [Russell and Norvig, 2003, p. 114]. Another example is simulated annealing which uses a combination of hill climbing and random walk. A research on the performance in this matter would be profitable, but is beyond the boundaries of this project.

3.4 CP languages

In this section properties of programming languages that are fit for working with CSPs will be discussed, but only Standard Meta Language (SML) and Prolog with the bounds library, will be adressed due to boundaries of the project.

The Prolog programming language has a very effective and useful library, `bounds`, that can be used to work with Constraint Logic Programming (CLP). In this library features for working with variables, domains of values and constraints are built-in. Domains are defined by ranges of integers or as a union of these. Constraint can be computed using the built-in arithmetic operators, implications and refined constraints in the bounds library. The solutions are found by assigning variables in the desired order, e.g. the MCV heuristic is already implemented in the language and can be used upon request. Also propagation is built-in and is invoked by default when working with CLP. This makes the solving of problems very effective and all the other features of the Prolog engine is still available, which opens the possibilities, but as mentioned with the domains, the CLP is bound to work with integers [1].

SML's strengths are in the high-order functional programming possibilities. Types and datatypes are open for definition, making it easy to work with compound variables. Also concurrency of the solving is an obvious advantage when working with SML. Finally there exists functional languages that work with constraints based on

[1]For more information see `http://www.swi-prolog.org/man/clpfd.html`

the SML, Alice ML is one. This language utilise concurrency as well, but suffers as the Prolog bounds library from being tied to work with integers [2].

Prolog and SML are both declarative languages and the usage of logic is therefore implicit in both cases. Logic is however used exclusively in Prolog.

A small experiment was conducted in Prolog, where variables available from the music library consisted of facts and functions were used to implement constraints, see figure 3.3.

Three things comes to mind when analysing the program. The only constraint that is actually used from the bounds library is the `all_different`(+*Vars*) constraint, no domain is defined for any variable and finally that the library is a constraint itself. When studying each constraint it is realised that only keys and tempo are numerical constraints, making the numerical domains for variables useless. To link our variables (songs) to the library a relation between variables (attributes) have to be defined. The `tuples_in`(+*Tuples*, +*Extension*) constraint at first seems to be helpful in constraining our variables to the library. But is determined less useful after closer inspection, because the relation (*Extension*) has to be tuples of integers. This constraint is the only implemented constraint that links a relation between variables and if the songs cannot be taken from the library the program will solve the problem and then see if the solution exists in the library afterwards. This approach is very inefficient because the program can come up with many solutions to the playlist before finding a full solution, that can be found in the library. Converting every attribute in a song to a numerical value would do the trick, but would use a lot of space having the library listed as a look-up table with the corresponding hash-values. The program in figure 3.3 solves the problem using Prolog's own library and with the help of only the `all_different`(+*Vars*) constraint, but does not do so very effectively. A test shows that the program can find a list of 7 songs from a library of 10 songs, that can be combined without violating any constraints in less than a second, but uses more than 9 minutes! to find a combination of 9 songs.

The full source code for a test and the program with comments in Prolog is found in appendix A.2.

After the discussion on the subject of using CP in APG a specific solution to the problem will be presented, first the overall design in the next chapter and the implementation in the following.

[2]For more information see `http://www.ps.uni-saarland.de/alice/manual/`

```
/* Returns a subset of Y */
subset([A|X], Y) :- member(A, Y), subset(X, Y).
subset([], _).  % The empty set is a subset of every set.

/* Returns two succeeding elements of Pl */
succ(X, Y, Pl) :-
 append(_, [X, Y|_], Pl).

/* Returns if two succeeding artist lists contains equal artists */
neq_art(Ar1, Ar2) :-
 intersection(Ar1, Ar2, []).

/* Returns if two succeeding tempi are corresponding */
in_temp(T1, T2) :-
 T1 =< T2 + 10,
 T1 >= T2 - 10.

/* Returns if two succeeding lists of keys are harmonic */
harm(Ks1, Ks2) :-
 member(K1, Ks1),
 member(K2, Ks2),
 ((K1 =< K2 + 1,
   K1 >= K2 - 1);
 K1 =:= K2 + 12; % from minor to major
 K1 =:= K2 - 12), !. % from major to minor

/** Automatic Playlist Generator:                    *
  * Computes a playlist, Pl, from library of songs, *
  * a given seed song and length                    */
apg((N, Ar, Al, T, Ks), L, Pl) :-
 length(Pl, L),
 s(N, Ar, Al, T, Ks), % seed in library
 findall((X1, X2, X3, X4, X5),
  s(X1, X2, X3, X4, X5),
  Library), % compute library
 append([(N, Ar, Al, T, Ks)], Tl, Pl), % seed first in list
 subset(Tl, Library), % rest a subset of the library
 all_different(Pl),
 forall(succ((_, Ar1, Al1, T1 ,Ks1),
      (_, Ar2, Al2, T2, Ks2), Pl), % two successors
        (neq_art(Ar1, Ar2), % different artists
  Al1 \= Al2, % different albums
  in_temp(T1, T2), % corresponding tempo
  harm(Ks1, Ks2))), % harmonic keys
 !. % cut after finding a solution
```

Figure 3.3: `succ()` is a function to get two succeeding variables in a list, `neq_art()` is constraint for every songs artists to be different from its succeeding, `in_temp()` is constraint for every song to be within $[-10; 10]$ in bpm of a succeeding and `harm()` is constraint for every song to be in harmony with its succeeding. `apg()` is the main function to compose a playlist og length L, given a seed song. A song consists of $s(\#N, \#Ar, \#Al, \#T, \#Ks)$, where $\#N$ is the title, $\#Ar$ the list of artists, $\#Al$ the name of the album, $\#T$ the tempo in bpm and $\#Ks$ a list of the keys. The keys are here converted to be numerical values from $1 - 24$ in stead of $1A - 12B$.

CHAPTER 4

Implementation

The program that have been implemented on the basis of the established theory and analysis is prestented in this chapter. It has been implemented in Moscow ML, which is a light-weight implementation of SML[1].

4.1 Datastructure

To work with the problem specificly choices on attributes in the variables and thereby their domains has to be made. Below is a representation of variables in the program.

$$\mathcal{X} = \left\{ s_i = \begin{pmatrix} place_i : \texttt{int} \\ name_i : \texttt{string} \\ artists_i : \texttt{string list} \\ album_i : \texttt{string} \\ tempo_i : \texttt{real} \\ keys_i : \texttt{string list} \end{pmatrix} \right\}$$

$keys_i$ refers to the global key of a song from the circle of fifths presented in figure 1.1. Due to delimitation of the project it has only been possible to get information on

[1]For more information see `http://www.itu.dk/~sestoft/mosml.html`.

tempo (bpm) and global key for every song to base the similarity function between songs upon. Based on the theory conducted in section 2.1 constraints on the variables are listed below.

$$\mathcal{C} = \left\{ \begin{array}{c} \texttt{all\_different}(s_i), \\ artists_i \neq artists_{i-1} \wedge artists_i \neq artists_{i+1}, \\ album_i \neq album_{i-1} \wedge album_i \neq album_{i+1}, \\ tempo_i < tempo_{i+1} + 10 \wedge tempo_i < tempo_{i-1} + 10 \wedge \\ tempo_i > tempo_{i-1} - 10 \wedge tempo_i > tempo_{i+1} - 10, \\ tempo_i < +5 \cdot length \cdot tempo_j \wedge \\ tempo_i > -5 \cdot length \cdot tempo_j \textbf{ for } i \neq j, \\ keys_i = keys_i \vee keys_i = keys_i \pm 1 \vee keys_i = keys_i \pm 12 \end{array} \right\}$$

The constraints are from above: every song should be different, every succeeding artist must be different, every song must be in tempo with its succeeding and lastly the songs must harmonise with its succeeding. Number 1 and 5 are global constraints and the rest are binary. The place of a song is not constrained because it is merely for convinience and always corresponds to the position in the variable list. Unary constraints are generated implicitly by the interface functions. The domain is defined to meet the constraints and to make it as easy to compute the wanted result as possible.

$$\mathcal{D} = \left\{ s_i \in d_i = \begin{pmatrix} s_i \notin banned\_songs_i \\ artists_i \notin illegal\_artists_i \\ album_i \notin illegal\_albums_i \\ tempo_i < lower\_limit_i \\ tempo_i > upper\_limit_i \\ keys_i \in legal\_keys_i \end{pmatrix} \right\}$$

The domain does not entirely follow the problem description in equation 3.1, but the idea of defining a domain for every attribute remains, except for the already mentioned place.

The playlist generator consists of three interface functions. One to compose a playlist of a given length from a seed song, and two, to model the proposed playlist into a satisfying result. This will give the user the opportunity to change the playlist if it does not fulfil the expectations by adding more and more information to the playlist until it does. The two modelling functions consists of a suggest song function to add a desired song to the playlist and a ban song function to remove an unwanted.

On the basis of the above stated descriptions the datastructure is defined below.

```
(* place, name, artist(s), album, tempo (bpm), key(s) *)
type song =
    int * string * string list * string * real * string list;

(* library of songs *)
type library =
    song list;

(* playlist *)
type playlist =
    song list;

(* place, banned songs, illegal artists, illegal albums
   lower limit tempo, upper limit tempo, legal keys *)
type domain =
    int * song list * string list * string list *
    real * real * string list;

(* function that constrains domains *)
type constraint =
    song * domain list -> domain list;

(* argument for solution calculation *)
datatype arg = InsertA of song
             | ListA of playlist;
```

The types `song` and `domain` follows the definition above directly. Furthermore the type `constraint` consists of a function from a `song` and `domain` list to a new `domain` list because it is defined as a propagation function. Lastly the datatype `arg` provides the possibility of defining a function generally to handle both a song and a playlist. This is useful when writing the function SOLUTION(), that determines whether a song or a playlist is a solution given a corresponding domain.

4.2 Functions

To easily access and model information of songs, domains, playlists and domain lists a series of basic functions are defined after the datastructure. The code of these basic functions are kept out of the report for the purpose of minimising space. All source code of the program can be found in appendix A.1. Among the basic functions some more critical functions exists. These are SOLUTION(), OBJ() and CONSTRAINT-PROP(). SOLUTION() is, as described earlier, defined for both a single song and a whole playlist. It simply consists of boolean expressions that express if every attribute obeys its corresponding domain and a recursive call if the argument is a list. The

OBJ() function simply returns a number for how much a domain is open, i.e. how possible it is for song from the library to fulfil the demands. CONSTRAINTPROP() is constraint propagation function that runs through the list of constraints and applies them to a given domain. The main functions consists of RANDOMASSIGN(), MCV(), CONFLICTS() and MINCONFLICTS(). The RANDOMASSIGN() assigns, as the name suggests, every variable in the playlist to randomly chosen values from the library. The MCV() is an implementation of the MCV heuristic for variable selection based on the objective function and is used in the main algorithm. The CONFLICTS() function follows the function of the same name in the main algorithm, but instead of returning a variable that minimises the problem it returns a minimised CSP, see figure 4.1.

```
CONFLICTS(var, csp, max_steps) returns a possibly modified CSP
    inputs:
        var, a conflicted variable
        csp, a constraint satisfaction problem
        max_steps, the number of steps allowed
                    to minimise the problem
    for $i = 1$ to max_steps do
        current ← get random song from LIBRARY[csp]
        if SOLUTION(current, csp)
            csp' ← CONSTRAINTPROP(current, csp)
            if OBJ(csp') < OBJ(csp)
                var ← current
                csp ← csp'
    return csp
```

Figure 4.1: LIBRARY[] gets the library of songs from *csp* and the SOLUTION() function returns whether *current* satisfies *csp*. CONSTRAINPROP() propagates the knowledge from assigning the variable to *current* in *csp* and the OBJ() function returns a number for how closed the domains are in a CSP.

It uses constraint progation and the objective function to determine whether the CSP is minimised. The main function MINCONFLICTS() follows the algorithm showed in figure 3.1 except that it does not do the initial assignment, it does not choose a random conflicted variable and it contains a check for impossible domains in the for-loop. In stead of a random variable it uses MCV() to find the most constrainted variable. If it finds an impossible domain it returns failure as it is also does if all steps are used and no solution is found. CONFLICTS() is called by the main function with 40 as *max_steps*, meaning that it finds the best value of 40 tries - if any is found.

The program initially uses seven constraint or propagation functions, one for each constraint defined earlier and one for an interface function, which will be discussed afterwards. The first constraint places itself in every *banned_song* domain except for itself ensuring that no other variable can be assigned to the same song. The next two places artists and albums in the neighbouring songs *illegal_artists* and *illegal_albums*, respectively. The binary tempo constraint defines new lower and upper tempo limits for every domain by the following two functions:

$$lower\_limit_j = \texttt{max(}lower\_limit_j\texttt{,}\ \ tempo_i - 10\ \cdot\ \texttt{abs(}i - j\texttt{))}$$
$$upper\_limit_j = \texttt{min(}upper\_limit_j\texttt{,}\ \ tempo_i + 10\ \cdot\ \texttt{abs(}i - j\texttt{))}$$

The global tempo constraint is defined by the follwing functions:

$$lower\_limit_j = \texttt{max(}lower\_limit_j\texttt{,}\ \ tempo_i - 5\ \ \cdot size\texttt{)}$$
$$upper\_limit_j = \texttt{min(}upper\_limit_j\texttt{,}\ \ tempo_i + 5\ \ \cdot size\texttt{)}$$

size is the length of the playlist considered. Both tempo constraints contains a check so that it is never possible to assign a limit to a negative value. The last constraint is the most complex. It first updates its own domain and then uses a list of tuples with harmonic keys to determine new legal keys for every song composing the intersection with *legal_keys* generating the new legal keys. This check could have been made more efficient by converting keys to numerical values from 1 to 24 in stead[2], but is kept in the original signature so that information for a song is not changed.

The interface consists, as mentioned, of three functions APG(), SUGGESTSONG() and BANSONG(). The algorithm for APG() is in figure 4.2.

It uses the random restart hill climbing algorithm on the min conflicts local search. SUGGESTSONG() and BANSONG() uses the same pattern, but they respectively suggests and bans a selected song from a given complete assignment before the call to the min conflicts local search. SUGGESTSONG() makes use of the already defined constraints to propagate to the other domains and BANSONG() makes use of a to the purpose defined constraint. It places the specified song in every *banned_songs*$_i$ domain and removes the song from the playlist, so that no variables are assigned to this value.

4.3 Sample session

Below is a sample session of the program. It generates a playlist of 10 songs with "Wildcat.mp3" as seed song using the APG() function.

[2]See the Prolog experiment in figure 3.3 to see it implemented as described.

```
APG($seed$, $csp$) returns a solution
    inputs:
        $seed$, a seed song for the playlist
        $csp$, a constraint satisfaction problem
    $current \leftarrow$ RANDOMASSIGN($seed$, $csp$)
    $current' \leftarrow$ MINCONFLICTS($current$, SIZE[$current$] $\cdot$ 10)
    if NOT SOLUTION($current'$)
        $current' \leftarrow$ APG($seed$, $csp$)
    return $current'$
```

Figure 4.2: Algorithm for the recursive automatic playlist generator. It uses RANDOMASSIGN() to randomly and completely assign the variables in *csp* with *seed* as seed song and then call MINCONFLICTS() to try to solve the problem and then calls itself recursively until a solution is found.

```
- use "apg.sml";
> ...
- use "constraints.sml";
> ...
- use "library.sml";
> ...
- val s1 = setPlace(List.nth(l, 1363), 0);
> val s1 = (0, "Wildcat.mp3", ["Ratatat"], "Classics", 116.0, ["4A"]) :
  int * string * string list * string * real * string list
- val (ds1, pl1) = apg(s1, cs, l, 10);
> ...
- playlistPrint(pl1);
>
0, Wildcat.mp3, [Ratatat], Classics, 116.0, [4A].
1, Rain.mp3, [Kerri Chandler], Unknown Album, 123.0, [3A].
2, Couchez Avec Toi 2.mp3, [Vive la fete], Paris, 123.0, [2A].
3, The Call Of The Ktulu.mp3, [Metallica], S&M, 128.0, [1A, 3B].
4, Let There Be Light.mp3, [Justice], Cross, 123.0, [2A, 3B].
5, Take A Bow.mp3, [Muse], Black Holes & Revelations, 131.0, [2A, 5A].
6, Hallelujah.mp3, [Keith Jarrett], Unknown Album, 124.0, [10A, 2A].
7, The Thing That Should Not Be.mp3, [Metallica], S&M, 116.0, [2A].
8, I Should Know.mp3, [Dirty Vegas], Dirty Vegas, 123.0, [2A].
9, Pont Des Arts.mp3, [St. Germain], Tourist, 124.0, [2A, 1A].

> val it = () : unit
-
```

The SUGGESTSONG() and BANSONG() is used similarly but needs a domain list in addition to alter a given playlist. Sample queries suggesting “About A Girl.mp3” and banning song number 5 from the given playlist are given below.

```
- val s2 = setPlace(List.nth(l, 11), 4);
> val s2 = (4, "About A Girl.mp3", ["Nirvana"],
            "MTV Unplugged In New York", 122.0, ["2A"]) :
  int * string * string list * string * real * string list
- val (ds2, pl2) = suggestSong(s2, pl1, cs, ds1, l);
> ...
- val (ds3, pl3) = banSong(List.nth(pl2, 5), pl2, cs, ds2, l);
> ...
```

4.4 Application of the program

The program is meant to be used as the main engine for a playlist generator in, e.g. an online music community, a music playing program or wherever an automatic playlist generator can be of use. It should work together with information of every song in the library, which could be done by analysis of the digital signal when the song is added to the library. If the library is very large a collaborate filtering would be useful to narrow down the search space quickly. The interface functions makes relevance feedback possible and the application that implements the program should make use of these features.

Chapter 5

Discussion

This chapter will establish an evaluation of the implementation and discuss the quality of the solution based on the established theory and analysis. The evaluation will include a presentation of test results from functional and statistical tests.

5.1 Evaluation

Two types of tests have been run on the program; one for the functionalities and one for statistical reasons. The first test generates a playlist of a specified length from a specific seed song, s1. Then it suggests a specific song, s2 at place 4 in this list. The specifications of s1 (at place 0) and s2 is seen in their respective domains from the screen dump below.

```
(* Screen dump of domain from terminal in functional test *)
val ds =
    [(0, [s2], [], [],
      116.0, 116.0, ["4A"]),
     (1, [s1, s2], ["Artist1"], ["Album1"],
      106.0, 126.0, ["3A", "4A", "5A", "4B"]),
     (2, [s1,s2], [], [],
      102.0, 136.0, ["2A", "3A", "4A", "3B"]),
     (3, [s1,s2], ["Artist2"], ["Album2"],
      112.0, 132.0, ["1A", "2A", "3A", "2B"]),
(4, [s1], [], [], 122.0, 122.0, ["2A"]),
(5, [s1, s2], ["Artist2"], ["Album2"],
 112.0, 132.0, ["1A", "2A", "3A", "2B"]),
(6, [s1, s2], [], [], 102.0, 142.0,
 ["1A", "2A", "3A", "4A",
  "12A", "1B", "2B", "3B"]),
(7, [s1, s2], [], [], 92.0, 152.0,
 ["1A", "2A", "3A", "4A", "5A", "11A",
  "12A", "1B", "2B", "3B", "4B", "12B"]),
(8, [s1, s2], [], [], 82.0, 162.0,
 ["1A", "2A", "3A", "4A", "5A", "6A", "10A", "11A",
  "12A", "1B", "2B", "3B", "4B", "5B", "11B", "12B"]),
(9, [s1, s2], [], [], 72.0, 166.0,
 ["1A", "2A", "3A", "4A", "5A",
  "6A", "7A", "9A", "10A", "11A",
  "12A", "1B", "2B", "3B", "4B",
  "5B", "6B", "10B", "11B", "12B"]),
(10, [s1, s2], [], [], 67.0, 171.0,
 ["1A", "2A", "3A", "4A", "5A", "6A",
  "7A", "8A", "9A", "10A", "11A", "12A",
  "1B", "2B", "3B", "4B", "5B", "6B",
  "7B", "9B", "10B", "11B", "12B"])] : domain list
```

It is also seen that the other domains have been propagated so that a solution to any other place would be a solution to the current playlist. From the propagation it is also seen that the domains between the two assignments are very narrow due to close positions of the seed and the suggested song. It is only the last song in the playlist that has a full open domain, with respect to *banned_songs*, *illegal_artists*, *illegal_albums* and *legal_keys*, and a range of 104 bpm in tempo, which it is fair to say that almost every song would fulfil.

A second song is now suggested to list, that causes an empty domain, meaning that the request cannot be fulfilled. The test catches this failure in the query and throws a Failure exception of *impossible domain.* Lastly song number 5 is banned from the second list. The test is set up to run on playlists of length 10, but can easily be altered. It has been run several times on lengths 10, 50, 100 and 500, which it solves all problems for the first three within a few seconds and uses less than 1,5 minute to solve each with length 500.

The second test is a test for the rate of succes for the program. The random restart hill climbing has been turned of in the program and then a function generates a random assignment from a random seed song and then suggests a random song at a random place to the playlist for each 10th song in the playlist, besides the seed song. If a failure is encountered the test adds one to the number of failures and tries to solve the problem again. It does so until it has solved the problem and then it moves on to the next iteration. If at any time an impossible domain is generated the test restarts until it finds a problem that can be solved. The test is run 100 times for playlists of length 50 to find the best number of iterations for the main algorithm and then this number is used to run 100 times with length 10 and 100 to find the rate of succes. The results is shown in table 5.1

Length	Iterations	*max_steps*	Failures	p
50	100	4	25	0,75
50	100	5	23	0,77
50	100	6	7	0,93
50	100	7	12	0,88
50	100	8	7	0,93
50	100	9	3	0,97
50	100	10	1	0,99
10	100	10	0	1,0
100	100	10	1	0,99

Table 5.1: Length is the length of the playlist, *max_steps* is the number of steps used in the main algorithm, Failures is the number of failures generated by the statistical test and p is the probability of success.

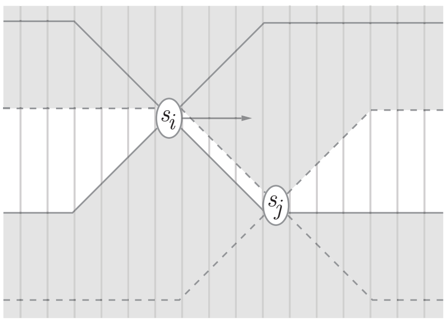
Both tests are included in appendix A.1. On the basis of the small statistical analysis it is fair to say that it rarely needs to use the random restart, but it is used for completeness.

The worst case time complexity can be estimated to $O(n^2 \cdot \frac{1}{p})$, where n is the length of the playlist and p is the probability of success, because *max_steps* in minConflicts() is linearly depended on the length of the playlist and so is mcv() and conflicts(). For the above used number of iterations on a playlist of length 50 the worstcase time comlexity would be estimated to be 2475 iterations. The probability of success depends both on the size of the library and the objective function and yields the chance for the conflicts() to hit a song in the library that fulfils the domain of the variable. The search is in the best case bound to linearly time complexity due to the initial assignment. The worst case space complexity is estimated to $O(n^2 + l)$, where n is the size of the playlist and l is the size of the library. Each domain can contain at most every song in $banned\_song_i$ generating the squared parameter and the playlist only uses n, because only a finite number neighboring states are generated in each step of the search. Finally the size of the library needs to be considered.

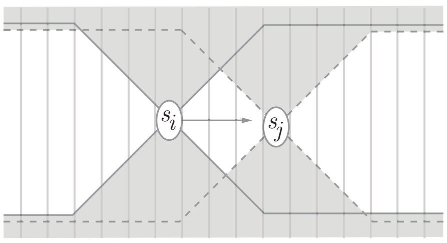
That the time comlexity is only bound to the library in the probability is good because it is qualitative factor and does not matter how big it is, only that it contains a good selection of songs to complement the playlist. An initial filtering of the library would improve the probability of success and thereby the efficiency.

These assesments are based solely on the algorithms and does not take into account that SML uses linear time when retrieving and changing information from lists and uses much space due to numerous recursive functions. This slows the program significantly. Also the choice of using $10 \cdot 40$ tries to find a good value (10 in the interfacefunctions and 40 in CONFLICTS()) for every variable is decided on the basis that variables had to have a number of tries each to find a solution. And the weight of giving the main algorithm 10 times each variable to choose the right variable and the CONFLICTS() 40 to find the right value was sensible. After practical testing these values also seemed to get good results.

The program uses a combination of the minimum conflicts and most constrained variable heuristic and assigns a variable to a value only if the value is a solution to the current state. This results in the in figure 5.1 showed scenarios.



(a) The worst-case scenario when assigning a variable after another, where the domain is decreased as much as possible.



(b) The best-case scenario when assigning a variable after another, where the domain is decreased with a minimum.

Figure 5.1: The two figures shows a playlist of horisontal spaces each outlined by vertical blur lines. A variable s_i from the list is assigned to a value. The solid dark line indicates the domain after this assignment. Another variable s_j is then assigned to another value within the domain. The dashed line shows the domain after assigning only this variable and the white space outlines the decreased domain after propagating when s_i is already assigned. The gray area is outside this domain.

The program strides with every new variable s_j to come as close to the level of the current state as possible due to these heuristics. The level of the state (here only one other assignment) is in figure 5.1 indicated by arrows from s_i. One could say that it for every assignment tries to leave the domain as open as possible.

The problem of APG have been formulated as a CSP and the priciples of CP have successfully been used to solve the problem. A constraint propagation function have been implemented to the APG and because constraint propagation is invoked once every time a variable is assigned to a value the principles of arc consistancy is also achieved.

The playlists that are generated by the program all fulfils the requirements and are with the library of about 5000 songs (to a point) different every time. The extensive use of randomness in the algorithms provides a solid basis for the playlists to differ. But the playlists are, nonetheless, somewhat useless. They greatly lack the use of timbral, rhythmic and melodics measures thus violating rules of similarity in music. The latter is obvious to a conneoisseur, when experiencing that e.g. rock easily is followed by electronica or classical music in the playlists that are generated. These are examples of violations of all three types.

5.2 Efficiency improvements

The first obvious improvement would be to choose another programming language so that the program is not slowed by look-up in lists. A programming language that has a library that supports constraint programming could aid the structure of the program and the use a general algorithm for constraint propagation. Also this could provide the option of using a generic Split function to divide the problem into subproblems. Regardsless of what a Split function would aid solve process by generating a search tree so the techniques of concurrency are applicable. The split would also, and maybe more importantly, apply the ”divide and conquer” procedure for the main algorithm which could improve the search. One way is to find the most constrained part of the domain of the variable under consideration. The MCV() function could easily be divided into sub functions, that each could handle a part of the domain. Then a solution to the next most constrained part of the domain could be found until one is found that fulfils the whole domain. But for this to work the constraint library needs to be able to operate on variables of both strings and integers. The domain for strings could be defined as regular expressions, but that would leave them infinite due to the pumping lemma of regular expressions [1]. Another approach would be to implement a generic constraint that could tie variables together, as it is done in the Prolog `bounds` library with `tuples_in(`$+Tuples$`, `$+Extension$`)`, just with variables. This approach, though, leaves the propagation procedure less efficient because it has to go through every tuple to find relations between variables. What makes the propagation procedure so efficient with integers is that equations defines relations between variables generally and domains restrict the values. The variables are just inserted into the equation and values can be determined. If the problem should be applied to a generic constraint library a convertion to integers would be the best solution. A system of values for each variable is needed so domains can be defined, e.g. artists are hashed to values between 1 to n where n is the number of

[1] For more information see `http://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages`.

different artists in the music library. This does not take combinations of different artists into account. Further comblications to this solution might arise, but is outside the scope of this project to research.

The global constraints tie the problem very closely together, but as stated in the theory section 2.2, every high-order constraint can be redefined as binary constraints if enough auxiliary variables are introduced. If the problem is redefined so every constraint are at most binary, then a definition of a Split function would be even more effective. A way to do this for the `all_diff()` constraint is to define a variable within the $banned\_songs_i$ domain that refers to the next variable's $banned\_songs_{i+1}$, i.e. $banned\_songs_{i+1} \in banned\_song_i$.

The program only implements arc consistancy. But because the problem is defined with mutual relations between variables, the problem is always consistent - that is if the music library can fulfil the domain and the users query is not impossible to fulfil. Consider again the two cases in figure 5.1. Even if the worst-case scenario is applied every time a solution to the problem still exists, given an infinite library of songs. There will always exist a path, represented by the white space between s_i and s_j in figure 5.1(a). But the library is finite and a possibility exists that no other variable fulfils the narrow demands between s_i and s_j.

It could have been educational to see the difference between the now implemented initial random assignment and an initial genetic assignment. It is hard to predict the outcome of a this due to binary constrained problem. An assessment is that the genetic algorithm works well with global and unary constraints, but could be left powerless with binary constraints. This assessment is based on the fact that if one variable is assigned to value that does not fulfil its domain then the variables linked to this will also be unfit for the solution. This is not the case when dealing with global or unary constraints.

5.3 Improvements of quality

The playlists that are generated by the program is not ready for qualitative research at the time being due to lack of similarity measures. It is therefore problematic to discuss the effect of the implemented features. These circumstances are changed if the program is extended to include similarity measures in rhythm, melody and timbre. Fortunately the program works dynamically on attributes and domains of songs and the constraints of these, so it is very easy expand the CSP of the program to include these measures. Some programming languages has libraries to process audio signals and retrieve information from the music just by invoking a function. This way the abstraction from the digital signal processing can be kept. Some technical improvements will be discussed in the following.

Because the problem already uses an objective function the problem would be easy to convert to a COP. If it is done the techniques of simulated annealing can be applied to find the best solution to a problem that may be impossible to solve. This would improve the quality of the playlists because no playlist is generated by the program at this point if an impossible query is met.

Also it could be educational to try to observe the effects of a collaborate filtering with the APG engine. A test similar to the statistical test presented in the beginning of this chapter could aid the assessment of the rate of success.

The relevance feedback seems to operate quite good with the suggestion of songs, but the ban song does make use of the information given by the user, when a song is banned. It just removes the song from the lists and makes sure that it is not added again. A research on the subject what information can be drawn from banning a song would be beneficial.

Finally the program does not make use of information about the users habits. It is, though, at this point not possible because the APG is not implemented in a music program.

Chapter 6

Conclusion

The problem of Automaitc Playlist Generation (APG) have proven suited for Constraint Programming (CP). The technique of constraint propagation is successfully applied to the problem. The problem is thus kept at a level of consistancy that insures a solution to every problem that is initially consistent, given an extensive music library. The problems are also solved effectively with the help from the appliance of local search to CP. The implementation of the program in another programming language is, though, preferable to get rid of unnecessary iterations.

The playlist that are generated are somewhat useless at this point. They could, though, be improved drastically by adding timbral, rhythmic and melodic measures to the similarity function in the program. The program is left open for these new features to be implemented.

If the project was to be extended the first step would be to gather information of more musical measures so a qualitative research of the playlists could be conducted. Next step would be to transfer the program to another programming language to get the full benefit of the efficiancy improvements. And maybe with this consider a language that has a library with digital signal processing functions so that the previous stated requirements can be met. After this the problem could be redefined to a COP and the techniques of simulated annealing could be applied. Finally an application of the APG in a music program so that its potetial can be utilised.

Appendix A

Appendix

A.1 SML source

A.1.1 apg.sml

```
load "List";
load "Int";
load "Real";
load "Random";

(*****************************************************
 *                                                   *
 *Contains functions for Automatic Playlist Generation *
 *                                                   *
 *Author: Michael øLune                             *
 *                                                   *
 *****************************************************)

(*******************
 * Type decleration *
 *******************)

(* place, name, artist(s), album, tempo (bpm), key(s) *)
type song =
    int * string * string list * string * real * string list;
```

```
(* library of songs *)
type library =
    song list;

(* playlist *)
type playlist =
    song list;

(* place, banned songs, illegal artists, illegal albums
   lower limit tempo, upper limit tempo, legal keys *)
type domain =
    int * song list * string list * string list *
    real * real * string list;

(* function that constrains domains *)
type constraint =
    song * domain list -> domain list;

(* argument for solution calculation *)
datatype arg = InsertA of song
             | ListA of playlist;

exception Failure of string;

(* list of possible keys *)
val possKeys =
    ["1A", "2A", "3A", "4A", "5A", "6A",
     "7A", "8A", "9A", "10A", "11A", "12A",
     "1B", "2B", "3B", "4B", "5B", "6B",
     "7B", "8B", "9B", "10B", "11B", "12B"];

(******************
 * Basic functions *
 ******************)

(* return true if two song are equal otherwise false *)
infix 6 eq;
fun (nam, art, al, _, tem, keys) eq
    (nam', art', al', _, tem', keys') =
        nam = nam' andalso
        art = art' andalso
        al = al' andalso
        tem = tem' andalso
        keys = keys';

infix 6 neq;
fun s1 neq s2 = not(s1 eq s2);

(* getters for attributes in song *)
fun getPlace(place, _, _, _, _, _) = place;
fun getName(_, name, _, _, _, _) = name;
fun getArtists(_, _, artist, _, _, _) = artist;
fun getAlbum(_, _, _, album, _, _) = album;
fun getTempo(_, _, _, _, tempo, _) = tempo;
fun getKeys(_, _, _, _, _, keys) = keys;
```

```
(* setters for attributes in song *)
fun setPlace((_, nam, art, al, tem, keys), p) =
        (p, nam, art, al, tem, keys);
fun setName((p, _, art, al, tem, keys), nam) =
        (p, nam, art, al, tem, keys);
fun setArtists((p, nam, _, al, tem, keys), art) =
        (p, nam, art, al, tem, keys);
fun setAlbum((p, nam, art, _, tem, keys), al) =
        (p, nam, art, al, tem, keys);
fun setTempo((p, nam, art, al, _, keys), tem) =
        (p, nam, art, al, tem, keys);
fun setKeys((p, nam, art, al, tem, _), keys) =
        (p, nam, art, al, tem, keys);

(* getters for attributes in domain *)
fun getDPlace(place, _, _, _, _, _, _) = place;
fun getBans(_, bans, _, _, _, _, _) = bans;
fun getDArtists(_, _, artists, _, _, _, _) = artists;
fun getDAlbums(_, _, _, albums, _, _, _) = albums;
fun getLower(_, _, _, _, lLimit, _, _) = lLimit;
fun getUpper(_, _, _, _, _, uLimit, _) = uLimit;
fun getDKeys(_, _, _, _, _, _, keys) = keys;

(* setters for attributes in domain *)
fun setDPlace((_, bans, art, alb, lLimit, uLimit, keys):domain, p) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setBans((p, _, art, alb, lLimit, uLimit, keys):domain, bans) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setDArtists((p, bans, _, alb, lLimit, uLimit, keys):domain, art) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setDAlbums((p, bans, art, _, lLimit, uLimit, keys):domain, alb) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setLower((p, bans, art, alb, _, uLimit, keys):domain, lLimit) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setUpper((p, bans, art, alb, lLimit, _, keys):domain, uLimit) =
        (p, bans, art, alb, lLimit, uLimit, keys);
fun setDKeys((p, bans, art, alb, lLimit, uLimit, _):domain, keys) =
        (p, bans, art, alb, lLimit, uLimit, keys);

(* places a song at specified place in playlist *)
fun place(s:song,pl:playlist) =
    let val p = getPlace(s)
        val hd = List.take(pl, p)
        val tl = List.drop(pl, p)
        val tl' = List.map (fn x =>
                    setPlace(x, getPlace(x)+1)) tl
    in hd@[setPlace(s, p)]@tl'
    end;

(* removes a song from a playlist *)
fun remove(s:song, pl:playlist) =
    let val p = getPlace(s)
        val hd = List.take(pl, p)
        val tl = List.drop(pl, p+1)
        val tl' = List.map (fn x =>
                        setPlace(x, getPlace(x)-1)) tl
    in hd@tl'
    end;

(* default full domain *)
fun defaultDomain(p) =
    (getPlace(p), [], [], [], 0.0, 1000.0, possKeys);

(* remove domain from domain list *)
fun removeD(d:domain, ds:domain list) =
    let val p = getDPlace(d)
        val hd = List.take(ds, p)
        val tl = List.drop(ds, p+1)
        val tl' = List.map (fn x =>
                        setDPlace(d, getDPlace(x)-1)) tl
    in hd@tl'
    end;

(* internal ban constraint *)
fun banC(s, ds) =
    List.map (fn x =>
        setBans(x, getBans(x)@[s])) ds;

(* generates a random number in range [min; max] *)
local val gen =
    let val g = Random.newgen()
    in ref g
    end;
in fun ran(min, max) =
    let val g = !gen
    in Random.range(min, max) g
    end;
end;

(* selects a random song from domain *)
fun getSong([]) = raise Failure "getSong"
  | getSong(l:library) =
    let val r = ran(0, List.length(l))
    in List.nth(l, r)
    end;

(* returns the number of times a song is in a list *)
fun containsSong(s:song, pl:playlist) =
    List.foldr (fn (x, b) =>
        if x eq s
        then 1.0+b
        else b) 0.0 pl;

(* returns whether the song or playlist is a solution or not *)
fun solution(arg:arg, ds:domain list) =
    case arg of InsertA(s) =>
        let val d = List.nth(ds, getPlace(s))
            val temp = getTempo(s)
           (* song is not banned *)
        in containsSong(s, getBans(d)) = 0.0 andalso
           (* is not of illegal artist *)
           (List.all (fn x =>
               List.all (fn y =>
                   x <> y) (getDArtists(d))) (getArtists(s))) andalso
           (* is not in illegal album *)
           (List.all (fn x =>
                x <> getAlbum(s)) (getDAlbums(d))) andalso
           (* within lower limits of tempo *)
           getLower(d) <= temp andalso
           (* within upper limits of tempo *)
           getUpper(d) >= temp andalso
           (* has legal key *)
           (List.exists (fn x =>
               List.exists (fn y =>
                   x = y) (getDKeys(d)))
                       (getKeys(s)))
        end
     | ListA([]) => true (* empty playlist *)
     | ListA(s::pl) => solution(InsertA(s), ds) andalso
                       solution(ListA(pl), ds); (* recursive call *)

(* returns the objective for a domain *)
fun obj(d) =
    let val art = Real.fromInt(List.length(getDArtists(d)))
        val alb = Real.fromInt(List.length(getDAlbums(d)))
        val diffT = getUpper(d)-getLower(d) + 0.1
        val key = Real.fromInt(List.length(possKeys) -
                             List.length(getDKeys(d)))
    in art + alb + 100.0/diffT + key
    end;

(* propagates constraints from one variable onto others *)
fun constraintProp(s:song, cs:constraint list, ds:domain list) =
    (* project each constraint on every variable domain *)
    List.foldr (fn (c, b) => c(s, b)) ds cs;

(* builds or resizes a domain from a playlist *)
fun buildDomain(pl:playlist, cs:constraint list, ds:domain list) =
    let val dSize = List.length(ds)
        val plSize = List.length(pl)
    in if dSize < plSize
                    (* fill with default domains *)
       then let val ds' = List.foldl (fn (x, b) =>
                    if getPlace(x) > dSize-1
                    then b@[defaultDomain(x)]
                    else b) ds pl
                    (* find suggested songs *)
                val sugg = List.foldr (fn (x, b) =>
                             (List.filter (fn y =>
                                 List.all (fn z =>
                                     y neq z) b)
                                         (getBans(x)))@b)
                                             [] ds'
                    (* update domain for every suggested song *)
```

```
            in (List.foldr (fn (x, b) =>
                    constraintProp(x, cs, b)) ds' sugg)
            end
        else List.take(ds, plSize)
    end;

(* returns whether the domain is impossible to fulfull *)
fun impDomain([]) = false (* empty domain *)
  | impDomain((d::ds):domain list) = (* general case *)
    (* impossible tempo *)
    (getUpper(d) < getLower(d)) orelse
    (* no legal keys left *)
    (List.length(getDKeys(d)) = 0) orelse
    (* recursive call *)
    (impDomain(ds));

(*****************
 * Main functions *
 *****************)

(* generates a random assignment *)
fun randomAssign(hdInt, tlInt, pl:playlist, l:library) =
    (* if last song to place *)
    if hdInt >= tlInt
    (* place last song *)
    then pl@[setPlace(getSong(l), tlInt)]
    (* recursive call *)
    else randomAssign(hdInt+1, tlInt, pl@
            [setPlace(getSong(l), hdInt)], l);

(* recursive function to find the most constrained domain *)
fun recVar(d:domain, _, []) = d
 |  recVar(d:domain, dC, (d'::ds):domain list) = (* general case *)
        let val dC' = obj(d')
        in  if dC' > dC
            then recVar(d', dC', ds) (* update domain *)
            else recVar(d, dC, ds) (* discard domain *)
        end;

(* selects the most constrained variable *)
fun mcv(pl:playlist, ds:domain list) =
        let val ds' = List.filter (fn x =>
                          not(solution(InsertA(List.nth(pl,
                                              getDPlace(x))),
                                       ds))) ds
            val d = List.hd(ds')
            (* find most constrained variable *)
            val d' = recVar(d, obj(d), ds')
        in List.nth(pl, getDPlace(d'))
        end;

(* returns the playlist that minimizes s *)
fun conflicts(var, _, pl:playlist, cs:constraint list,
    ds:domain list, _, 0) = (* no more attempts, return current *)
    (* if new variable found *)
```

```
     if var neq List.nth(pl, getPlace(var))
     (* place in playlist *)
     then (constraintProp(var, cs, ds),
             place(var, remove(var, pl)))
     (* no change *)
     else (ds, pl)
   | conflicts(var, v, pl:playlist, cs:constraint list,
     ds:domain list, l:library, max) = (* general case *)
     let val p = getPlace(var)
         val s = setPlace(getSong(l), p)
        (* song is a solution *)
     in if solution(InsertA(s), ds)
        (* return new domain, pl *)
        then let val ds' = constraintProp(s, cs, ds)
                 val v' = List.foldr (fn (x, b) =>
                                  b+obj(x)) 0.0 ds'
             in if v' < v
                (* update var *)
                then conflicts(s, v', pl, cs, ds, l, max-1)
                (* discard s *)
                else conflicts(var, v, pl, cs, ds, l, max-1)
             end
        else conflicts(var, v, pl, cs, ds, l, max-1) (* discard s *)
     end;

(* main recursive algorithm:                *
 * Replaces conflicted variables max times. *
 * Returns a minimized playlist             *)
fun minConflict(pl, _, ds, _, 0) =
     raise Failure "no solution" (* return current if no more attemps *)
   | minConflict(pl:playlist, cs:constraint list,
     ds:domain list, l:library, max) = (* general case *)
     if solution(ListA(pl), ds) (* if playlist is a solution *)
     then pl (* return current *)
     else if impDomain(ds) (* domain cannot be fulfilled *)
     then raise Failure "impossible domain"
     else let val var = mcv(pl, ds) (* select variable *)
              val n = Real.fromInt(List.length(pl))
              (* pl that minimizes var *)
              val (ds', pl') =
              conflicts(var, n*n*1000.0, pl, cs, ds, l, 40)
          in minConflict(pl', cs, ds', l, max-1)
          end;

(**********************
 * Interface functions *
 **********************)

(* suggests a song for a playlist *)
fun suggestSong(s:song, pl:playlist, cs:constraint list,
     ds:domain list, l:library) =
     let val pl' = place(s, pl) (* place song in pl *)
         val ds' = constraintProp(s, cs, ds) (* propagation *)
         val ds'' = buildDomain(pl', cs, ds') (* expand domain *)
        (* try solve problem *)
```

```
    in (ds'', minConflict(pl', cs, ds'', l,
                List.length(pl')*10))
       handle Failure "no solution" =>
                suggestSong(s, pl, cs, ds, l)
    end;

(* bans a song from a playlist *)
fun banSong(s:song, pl:playlist, cs:constraint list,
    ds:domain list, l:library) =
        (* remove domain and propagate *)
    let val ds' = banC(s, removeD(List.nth(ds, getPlace(s)), ds))
        (* remove banned song from playlist *)
        val pl' = remove(s, pl)
        (* try solve problem *)
    in (ds', minConflict(pl', cs, ds', l, List.length(pl')*10))
       handle Failure "no solution" =>
                banSong(s, pl, cs, ds, l)
    end;

(* main algorithm generates a playlist from a seed song *)
fun apg(s:song, cs:constraint list, l:library, size) =
    if size >= 1 (* must at least contain 1 song *)
    then let val s' = setPlace(s, 0) (* place first in playlist *)
             (* randomly assign rest of playlist *)
             val pl = randomAssign(1, size-1, [s'], l)
             (* build new domain list *)
             val ds = buildDomain(pl, cs, [])
             (* propagate *)
             val ds' = constraintProp(s', cs, ds)
             (* try to sovle problem *)
         in (ds', minConflict(pl, cs, ds', l, size*10))
            handle Failure "no solution" =>
                   apg(s, cs, l, size)
         end
    else raise Failure "the playlist must at least contain 1 song";

(**********************
 * Printing functions *
 **********************)

(* song to string *)
fun songToString(s:song) =
    let val artl = getArtists(s) handle Blind => [""]
        val art = List.foldr (fn (x, b) => x ^ ", " ^ b) "" artl
        val art' = String.substring(art, 0, String.size(art)-2)
        val keyl = getKeys(s) handle Blind => [""]
        val key = List.foldr (fn (x, b) => x ^ ", " ^ b) "" keyl
        val key' = String.substring(key, 0, String.size(key)-2)
    in Int.toString(getPlace(s)) ^ ", " ^
       getName(s) ^ ", [" ^
       art' ^ "], " ^
       getAlbum(s) ^ ", " ^
       Real.toString(getTempo(s)) ^ ", [" ^
       key' ^ "]."
    end;
```

```
(* pretty printer *)
fun playlistPrint(pl:playlist) =
    let val str = "\n" ^ List.foldr (fn (x, b) => songToString(x) ^ "\n"
         ^ b) "" pl ^ "\n"
    in print(str)
    end;
```

../SML/apg.sml

A.1.2 constraints.sml

```
(************************
 * Constriant functions *
 ************************)

(* place song only once in from playlist: Global *)
fun allDiff(s, ds) =
    List.map (fn x =>
           (* songs own domain *)
        if getPlace(s) <> getDPlace(x) andalso
           (* song is not already banned *)
           List.all (fn y => s neq y) (getBans(x))
           (* add to banned songs *)
        then setBans(x, getBans(x)@[s])
           (* do nothing *)
        else x) ds;

(* adds s to illegal artists to a given domain *)
fun updateArtists s d =
    let val sArtists = getArtists(s)
        val dArtists = getDArtists(d)
       (* domain belongs to song next to s *)
    in if Int.abs(getPlace(s) - getDPlace(d)) = 1
       (* add if not in illegal domain artists *)
       then setDArtists(d, List.foldl (fn (y, b) =>
                                 if List.all (fn z =>
                                        y <> z) dArtists
                                 then y::b
                                 else b) dArtists sArtists)
       else d
    end;

(* artist of the next song must not be the same: Binary *)
fun neqArtists(s, ds) =
    let val p = getPlace(s)
        (* get all before s-1 *)
        val hd = List.take(ds, p-1) handle Subscript => []
        (* get all after s+1 *)
        val tl = List.drop(ds, p+2) handle Subscript => []
        (* get the list of [0,..., s+1] *)
        val curr = List.take(ds, p+2)
```

```
                   (* s does not have a successer: [....,s] *)
                   handle Subscript => List.take(ds, p+1)
        (* get the list of [s-1, s, s+1] *)
        val curr' = List.drop(curr, p-1)
                    (* s does not have a preceeder: [s,...] *)
                    handle Subscript => List.drop(curr, p)
        (* update illegal artists in [s-1, s, s+1] *)
        val curr'' = List.map (updateArtists s) curr'
    in hd@curr''@tl
    end;

(* adds s to illegal album to a given domain *)
fun updateAlbum s d =
    let val sAlbum = getAlbum(s)
        val dAlbums = getDAlbums(d)
       (* domain belongs to song next to s *)
    in if Int.abs(getPlace(s) - getDPlace(d)) = 1
       (* add if not in illegal domain album *)
       then setDAlbums(d, if List.all (fn x =>
                                 x <> sAlbum) dAlbums
                          then sAlbum::dAlbums
                          else dAlbums)
       else d
    end;

(* album of the next song must not be the same: Binary *)
fun neqAlbums(s, ds) =
    let val p = getPlace(s)
        (* get all before s-1 *)
        val hd = List.take(ds, p-1) handle Subscript => []
        (* get all after s+1 *)
        val tl = List.drop(ds, p+2) handle Subscript => []
        (* get the list of [0,..., s+1] *)
        val curr = List.take(ds, p+2)
                   (* s does not have a successer: [....,s] *)
                   handle Subscript => List.take(ds, p+1)
        (* get the list of [s-1, s, s+1] *)
        val curr' = List.drop(curr, p-1)
                    (* s does not have a preceeder: [s,...] *)
                    handle Subscript => List.drop(curr, p)
        (* update illegal album in [s-1, s, s+1] *)
        val curr'' = List.map (updateAlbum s) curr'
    in hd@curr''@tl
    end;

(* updates upper tempo limit of domain *)
fun updateUpper(s, d) =
    let val uLimit = getUpper(d)
            (* calculate new upper limit *)
        val newLimit = getTempo(s)+10.0 *
            Real.fromInt(Int.abs(getPlace(s)-getDPlace(d)))
            (* update upper limit *)
    in setUpper(d, Real.min(newLimit, uLimit))
    end;
```

```
(* updates lower tempo limit of domain *)
fun updateLower(s, d) =
    let val lLimit = getLower(d)
        val diff = Real.fromInt(Int.abs(getPlace(s)-getDPlace(d)))
        val temp = getTempo(s)
                            (* new limit not lower than 0.0 *)
        val newLimit = if temp > 10.0*diff
                       then temp - 10.0*diff
                       else 0.0
       (* update lower limit *)
    in setLower(d, Real.max(newLimit, lLimit))
    end;

(* tempo must be within [-10; 10] for the next song: Binary *)
fun binaryTempo(s, ds) =
    (* update upper and lower tempo limit *)
    List.map (fn x =>
        updateLower(s, updateUpper(s, x))) ds;

(* updates upper and lower tempo limit from size of playlist *)
fun updateLimits(size, temp, d) =
    let val uLimit = getUpper(d)
        val lLimit = getLower(d)
        val newLow = if temp > 5.0 * size then temp - 5.0 * size else
            0.0
    in setLower(setUpper(d, Real.min(temp + 5.0 * size, uLimit)),
                Real.max(newLow, lLimit))
    end;

(* tempo of every song must be within [-5*length; 5*length]: Global *)
fun globalTempo(s, ds) =
    (* update global tempo limits *)
    List.map (fn x =>
        updateLimits(Real.fromInt(List.length(ds)),
                     getTempo(s), x)) ds;

(* tuples of harmonic keys *)
val harmKeys =
[("1A", "1B"), ("1A", "12A"), ("1A", "2A"), ("1A", "1A"),
 ("2A", "2B"), ("2A", "3A"), ("2A", "2A"), ("2A", "1A"),
 ("3A", "3B"), ("3A", "4A"), ("3A", "3A"), ("3A", "2A"),
 ("4A", "4B"), ("4A", "5A"), ("4A", "4A"), ("4A", "3A"),
 ("5A", "5B"), ("5A", "6A"), ("5A", "5A"), ("5A", "4A"),
 ("6A", "6B"), ("6A", "7A"), ("6A", "6A"), ("6A", "5A"),
 ("7A", "7B"), ("7A", "8A"), ("7A", "7A"), ("7A", "6A"),
 ("8A", "8B"), ("8A", "9A"), ("8A", "8A"), ("8A", "7A"),
 ("9A", "9B"), ("9A", "10A"), ("9A", "9A"), ("9A", "8A"),
 ("10A", "10B"), ("10A", "11A"), ("10A", "10A"), ("10A", "9A"),
 ("11A", "11B"), ("11A", "12A"), ("11A", "11A"), ("11A", "10A"),
 ("12A", "12B"), ("12A", "12A"), ("12A", "11A"), ("12A", "1A"),
 ("1B", "12B"), ("1B", "2B"), ("1B", "1B"), ("1B", "1A"),
 ("2B", "3B"), ("2B", "2B"), ("2B", "1B"), ("2B", "2A"),
 ("3B", "4B"), ("3B", "3B"), ("3B", "2B"), ("3B", "3A"),
 ("4B", "5B"), ("4B", "4B"), ("4B", "3B"), ("4B", "4A"),
 ("5B", "6B"), ("5B", "5B"), ("5B", "4B"), ("5B", "5A"),
 ("6B", "7B"), ("6B", "6B"), ("6B", "5B"), ("6B", "6A"),
 ("7B", "8B"), ("7B", "7B"), ("7B", "6B"), ("7B", "7A"),
 ("8B", "9B"), ("8B", "8B"), ("8B", "7B"), ("8B", "8A"),
 ("9B", "10B"), ("9B", "9B"), ("9B", "8B"), ("9B", "9A"),
 ("10B", "11B"), ("10B", "10B"), ("10B", "9B"), ("10B", "10A"),
 ("11B", "12B"), ("11B", "11B"), ("11B", "10B"), ("11B", "11A"),
 ("12B", "12B"), ("12B", "11B"), ("12B", "1B"), ("12B", "12A")];

(* updates a domains legal keys given its preceeding *)
fun updateDKeys(d, d') =
    if getUpper(d') = getLower(d')
    then d'
    else let val keys = getDKeys(d)
             val keys' = getDKeys(d')
                 (* find new possible keys *)
             val newKeys = List.foldr (fn ((k1, k2), b) =>
                                 if List.exists (fn x =>
                                        k1 = x) keys
                                 then k2::b
                                 else b) [] harmKeys
                 (* intersection of newKeys and keys' *)
             val keys'' = List.filter (fn x =>
                              List.exists (fn y =>
                                     x = y) newKeys) keys'
          in setDKeys(d', keys'')
          end;

(* successer functions: updates the second given two arguments *)
(* roll right *)
fun succr _ [] = []
  | succr _ [d] = [d]
  | succr f (d::d'::ds) =  d::succr f (f(d, d')::ds);
(* roll left *)
fun succl _ [] = []
  | succl _ [d] = [d]
  | succl f (d'::d::ds) = f(d, d')::succl f (d::ds);

(* key must be harmonic: Binary *)
fun harm(s, ds) =
    let val p = getPlace(s)
        val hd = List.take(ds, p)
        val tl = List.drop(ds, p+1)
            (* update corresponding domain *)
        val d = setDKeys(List.nth(ds, p), getKeys(s))
            (* update left from current *)
        val hd' = succl updateDKeys (hd@[d])
            (* update right from current *)
        val tl' = succr updateDKeys (d::tl)
    in hd'@List.drop(tl', 1)
    end;

(* list of constraints *)
val cs = [allDiff, neqArtists, neqAlbums,
          binaryTempo, globalTempo, harm];
```

../SML/constraints.sml

A.1.3 functionalTest.sml

```
use "apg.sml";
use "constraints.sml";
use "library.sml";

(*****************
 * Test functions *
 *****************)

(* playlist of 10 songs with:         *
 * (Wildcat.mp3, [Ratatat], Classics) *
 * as seed song                       *)
val s1 = setPlace(List.nth(l, 1363), 0);
val (ds1, pl1) = apg(s1, cs, l, 10);

(* suggest:                                                *
 * (About A Gril.mp3, [Nirvana], MTV Unplugged In New York) *
 * in place number 4 to playlist 1                         *)
val s2 = setPlace(List.nth(l, 11), 4);
val (ds2, pl2) = suggestSong(s2, pl1, cs, ds1, l);

(* suggest:                                      *
 * (Pile Of Gold.mp3, [The Blow], Paper Television) *
 * in place number 7 to playlist 2               *)
val s3 = setPlace(List.nth(l, 242), 7);
val (ds3, pl3) = suggestSong(s3, pl2, cs, ds2, l)
                 (* if no solution exists *)
                 handle Failure "impossible domain" => ([], []);

(* bans song number 5 from playlist 2 *)
val (ds4, pl4) = banSong(List.nth(pl2, 5), pl2, cs, ds2, l);

(********************************
 * print everything to terminal *
 ********************************)
print("\nPlaylist of 10 songs with (Wildcat.mp3, [Ratatat], Classics) as
     seed song:\n");
playlistPrint(pl1);
val sol1 = solution(ListA(pl1), ds1);

print("\nSuggest (About A Gril.mp3, [Nirvana], MTV Unplugged In New York
    ) in place number 4 to playlist 1:\n");
playlistPrint(pl2);
val sol2 = solution(ListA(pl2), ds2);

print("\nSuggest (Pile Of Gold.mp3, [The Blow], Paper Television) in
    place number 7 to playlist number 2:\n");
```

```
playlistPrint(pl3);
val sol3 = if List.length(pl3) = 0
           then false
           else solution(ListA(pl3), ds3);

print("\nBans song:\n" ^ songToString(List.nth(pl2, 5)) ^ "\nfrom
    playlist 2:\n");
playlistPrint(pl4);
val sol4 = solution(ListA(pl4), ds4);

val all = sol1 andalso sol2 andalso not(sol3) andalso sol4;
```

../SML/functionalTest.sml

A.1.4 statisticalTest.sml

```
use "apg.sml";
use "constraints.sml";
use "library.sml";

(******************
 * Test functions *
 ******************)

(* list of constraints *)
val cs = [allDiff, neqArtists, neqAlbums,
          binaryTempo, globalTempo, harm];

(* suggest a number of songs to a playlist *)
fun sugg(pl, _, ds, _, 0) = (ds, pl)
  | sugg(pl, cs, ds, l, n) =
        (* get random song from library and place randomly *)
    let val s = setPlace(getSong(l), ran(0, List.length(pl)))
        val pl' = place(s, pl) (* place song in pl *)
        val ds' = constraintProp(s, cs, ds) (* propagation *)
        val ds'' = buildDomain(pl', cs, ds') (* expand domain list *)
    in if not(impDomain(ds'')) (* not impossible domain *)
       then sugg(pl', cs, ds'', l, n-1) (* suggest a new song *)
       else sugg(pl, cs, ds, l, n) (* try again *)
    end

(* counts the number of failures given a length *
 * of a playlist and a number of iterations     *)
fun countFails(_, fails, 0, _) = fails
  | countFails(size, fails, n, p) =
        (* place first in playlist *)
    let val s = setPlace(getSong(l), 0)
        (* randomly assign rest of playlist *)
        val pl = randomAssign(1, size-1, [s], l)
        (* build new domain list *)
        val ds = buildDomain(pl, cs, [])
        (* propagate *)
        val ds' = constraintProp(s, cs, ds)
        (* one song suggestion per 10th song *)
        val suggs = Real.trunc(Real.fromInt(size-1)/10.0)
        val (ds'', pl') = sugg(pl, cs, ds', l, suggs)
        (* try to solve problem *)
        val pl'' = minConflict(pl', cs, ds'', l, size*p)
                   handle Failure "impossible domain" => []
                        | Failure "no solution" => [s]
                                (* impossible domain try again *)
    in case List.length(pl'') of 0 => countFails(size, fails, n, p)
                                (* failure found try again *)
                               | 1 => countFails(size, fails+1, n, p)
                                (* problem solved *)
                               | _ => countFails(size, fails, n-1, p)
    end;

val fails1 = countFails(50, 0, 100, 4);
val prob1 = Real.fromInt(fails1)/10.0;

val fails2 = countFails(50, 0, 100, 5);
val prob2 = Real.fromInt(fails2)/10.0;

val fails3 = countFails(50, 0, 100, 6);
val prob3 = Real.fromInt(fails3)/10.0;

val fails4 = countFails(50, 0, 100, 7);
val prob4 = Real.fromInt(fails4)/10.0;

val fails5 = countFails(50, 0, 100, 8);
val prob5 = Real.fromInt(fails5)/10.0;

val fails6 = countFails(50, 0, 100, 9);
val prob6 = Real.fromInt(fails6)/10.0;

val fails7 = countFails(50, 0, 100, 10);
val prob7 = Real.fromInt(fails7)/10.0;

val fails8 = countFails(10, 0, 100, 10);
val prob8 = Real.fromInt(fails8)/100.0;

val fails9 = countFails(100, 0, 100, 10);
val prob9 = Real.fromInt(fails9)/100.0;

(*val fails10 = countFails(500, 0, 100, 10);
val prob10 = Real.fromInt(fails10)/100.0;*)
```

../SML/statisticalTest.sml

A.2 Prolog source

A.2.1 apg.pl

```
:- ensure_loaded(library(bounds)).

/** Knowledge base:                        *
  * Sublist of songs from the library,     *
  * that can be combinded in a Playlist   */
s('Wildcat.mp3', ['Ratatat'], 'Classics', 116.0, [4]).
s('GimmeGimmeGimme.mp3', ['Black Flag'],
  'Unknown Album', 124.0, [2, 3]).
s('For Whom The Bells Tolls.mp3', ['Metallica'],
  'S&M [Disc 2]', 128.0, [2]).
s('Fin Fang Foom (Feat. Little Princ.mp3', ['Stereoheroes'],
  'SaGs Indie Electro Rock Playlist August', 130.0, [4, 3]).
s('About A Girl.mp3', ['Nirvana'],
  'MTV Unplugged In New York', 122.0, [2]).
s('Behind the Bushes.mp3', ['The Knife'],
  'Deep Cuts [Bonus Tracks] Disc 1', 120.0, [2]).
s('Wasting your time.mp3', ['Raised Fist'],
  'Ignoring the guidelines', 114.0, [3]).
s('Let There Be Light.mp3', ['Justice'],
  'Cross', 123.0, [2, 15]).
s('Bach, Johann Sebastian - Air on a G.mp3', ['Bach'],
  'Unknown Album', 125.0, [2]).
s('The Changeling.mp3', ['Compilations'],
  'L. A. Woman', 119.0, [3]).
s('11 Theme De Gerbier - Bof Larmee De.mp3', ['Nouvelle Vague'],
  'Coming Home', 124.0, [4]).

/* Returns a subset of Y */
subset([A|X], Y) :- member(A, Y), subset(X, Y).
subset([], _).  % The empty set is a subset of every set.

/* Returns two succeeding elements of Pl */
succ(X, Y, Pl) :-
 append(_, [X, Y|_], Pl).

/* Returns if two succeeding artist lists contains equal artists */
neq_art(Ar1, Ar2) :-
 intersection(Ar1, Ar2, []).

/* Returns if two succeeding tempi are corresponding */
in_temp(T1, T2) :-
 T1 =< T2 + 10,
 T1 >= T2 - 10.

/* Returns if two succeeding lists of keys are harmonic */
harm(Ks1, Ks2) :-
 member(K1, Ks1),
 member(K2, Ks2),
 ((K1 =< K2 + 1,
    K1 >= K2 - 1);
  K1 =:= K2 + 12; % from minor to major
  K1 =:= K2 - 12), !. % from major to minor

/** Automatic Playlist Generator:                    *
  * Computes a playlist, Pl, from library of songs, *
  * a given seed song and length                     */
apg((N, Ar, Al, T, Ks), L, Pl) :-
 length(Pl, L),
 s(N, Ar, Al, T, Ks), % seed in library
 findall((X1, X2, X3, X4, X5),
  s(X1, X2, X3, X4, X5),
  Library), % compute library
 append([(N, Ar, Al, T, Ks)], Tl, Pl), % seed first in list
 subset(Tl, Library), % rest a subset of the library
 all_different(Pl),
 forall(succ((_, Ar1, Al1, T1 ,Ks1),
      (_, Ar2, Al2, T2, Ks2), Pl), % two successors
        (neq_art(Ar1, Ar2), % different artists
  Al1 \= Al2, % different albums
  in_temp(T1, T2), % corresponding tempo
  harm(Ks1, Ks2))), % harmonic keys
 !. % cut after finding a solution
```

../Prolog/apg.pl

A.2.2 test.pl

```
:- consult(apg).

/* test computing a playlist of 7 with Wildcat.mp3 as seed */
test(Pl):-
 apg(('Wildcat.mp3', _, _, _, _), 7, Pl).
```

../Prolog/test.pl

A.3 Process

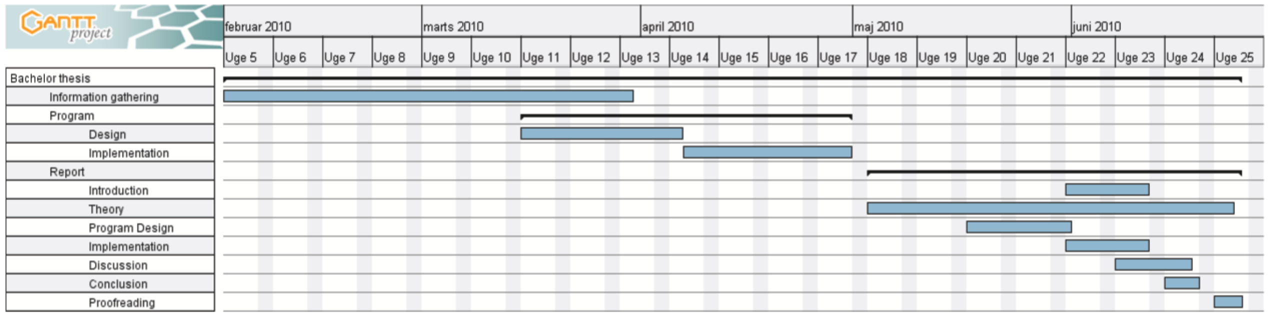


Figure A.1: The figure shows a gantt diagram of the process project.

Bibliography

[Allan *et al.*, 2007] Hamish Allan, Daniel Mullensuefen, and Geraint Wiggins, editors. *Methodological Considerations in Studies of Musical Similarity.* Austrian Computer Society, 2007.

[Anglade *et al.*, 2009] Amélie Anglade, Rafael Ramirez, and Simon Dixon, editors. *Genre Classification Using Harmony Rules Induced from Automatic Chord Transcriptions.* International Society for Music Information Retrieval, 2009.

[Apt, 2006] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambrige University Press, 2006.

[Cunningham *et al.*, 2006] Sally Jo Cunningham, David Bainbridge, and Anette Falconer, editors. *'More of an Art than a Science': Supporting the Creation of Playlists and Mixes.* University of Victoria, 2006.

[Foote *et al.*, 2002] Jonathan Foote, Matthew Cooper, and Unjung Nam, editors. *Audio Retrieval by Rhythmic Similarity.* Centre Pompidou, 2002.

[Logan, 2002] Beth Logan, editor. *Content-Based Playlist Generation: Exploratory Experiments.* Hewlett-Packard Labs, 2002.

[Nielsen, 1995] Frank Nielsen. *Grafteori - algoritmer og netværk.* Matematisk Institut, DTU, 1995.

[Pampalk and Gasser, 2006] Elias Pampalk and Martin Gasser, editors. *An Implementation of a Simple Playlist Generator Based on Audio Similarity Measures and User Feedback.* University of Victoria, 2006.

[Pampalk *et al.*, 2005] Elias Pampalk, Tim Pohle, and Gerhard Widmer, editors. *Dynamic Playlist Generation Based on Skipping Behavior*. Queen Mary, University of Lodon, 2005.

[Pauws *et al.*, 2006] Steffen Pauws, Wim Verhaegh, and Mark Vossen, editors. *Fast Generation of Optimal Music Playlist using Local Seach*. Philips Research Europe, 2006.

[Reynolds *et al.*, 2007] Gordon Reynolds, Dan Barry, Ted Burke, and Eugene Coyle, editors. *Towards a Personal Automatic Music Playlist Generation Algorithm: The Need for Contextual Information*. The Audio Research Group, School of Electrical Engineering Systems, 2007.

[Russell and Norvig, 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Mordern Approah*. Prentice Hall, 2nd edition, 2003.