Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 178 (2008) 647–662
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Music playlist generation by adapted simulated annealing

Steffen Pauws *, Wim Verhaegh, Mark Vossen [1]

Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Abstract

We present the design of an algorithm for use in an interactive music system that automatically generates music playlists that fit the music preferences of a user. To this end, we introduce a formal model, define the problem of automatic playlist generation (APG), and prove its NP-hardness. We use a local search (LS) procedure employing a heuristic improvement to standard simulated annealing (SA) to solve the APG problem. In order to employ this LS procedure, we introduce an optimization variant of the APG problem, which includes the definition of penalty functions and a neighborhood structure. To improve upon the performance of the standard SA algorithm, we incorporated three heuristics referred to as song domain reduction, partial constraint voting, and a two-level neighborhood structure. We evaluate the developed algorithm by comparing it to a previously developed approach based on constraint satisfaction (CS), both in terms of run time performance and quality of the solutions. For the latter we not only considered the penalty of the resulting solutions, but we also performed a conclusive user evaluation to assess the subjective quality of the playlists generated by both algorithms. In all tests, the LS algorithm was shown to be a dramatic improvement over the CS algorithm.

Keywords: Local search; Simulated annealing; Music playlist generation; Music retrieval

1. Introduction

Some of the cornerstones of the Ambient Intelligence vision are related to the embedded and context-aware qualities of new technologies [1,14]. Diffuse technologies will make functions and content available everywhere, at large quantities, at all times, and within reach for everyone for any context-of-use. As a present-day example, efficient audio compression techniques and the Internet already gave the impulse to people to collect massive volumes of digital music files for personal use. Additionally, Ambient Intelligence strongly advocates the need of personalized, adaptive, and anticipatory qualities of the new technologies. In other words, to make the digital environment useful, useable, and pleasurable for people, the functions of the technologies need to be concerned with real user needs and desires. As a real need from the wide-spread

* Corresponding author.
E-mail addresses: steffen.pauws@philips.com (S. Pauws), wim.verhaegh@philips.com (W. Verhaegh).
[1] Present address: Quinity, Maliebaan 50, 3581 CS Utrecht, The Netherlands.

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.08.019

availability of music, people welcome help to sort out their music and a handle to an easy and enjoyable music listening experience.

1.1. Informal problem definition

In order to realize personalized assistance in music choice, we research the automatic generation of music playlists by means of 'intelligent algorithms' borrowed from mathematical programming and combinatorial optimization. As a first prerequisite, we need to be able to reason about songs. Therefore, we think of songs as a list of attributes that are deemed to be relevant for music choice. As shown in Table 1, song attributes can be *nominal* such as the song/album title or the performing artist, allowing only reasoning in terms of *equivalence* (e.g., these two songs are by the same artist) and *set membership* (e.g., that song belongs to this album). Attributes can also be *numerical* such as the duration and the tempo of the song; numerical attributes allow the computation of a *difference* between attribute values. One can also think of other types of attributes such as *categorical* types allowing partial order relationships between attribute values, and *ordinal* ones having a total order on the values.

Having the description of songs in place, we should be able to automatically create music playlists. Informally, a playlist is a sequence of the 'right' songs at the 'right' positions that can be played back in one go. What is 'right' in this respect depends on the prevailing purposes of everyday music listening, including the music preferences, interests, and mood of the music listener. In our work, we model these desired playlist properties as formal constraints that are defined over the playlist positions in terms of song attributes. We distinguish three types of constraints. A *unary* constraint poses a restriction for a single playlist position (e.g., the first song should be a 'jazz-song'). A *binary* constraint declares a desired relation between songs at two different positions. For instance, this relation can express a song order (e.g., the first song should be faster than the second song) or a similarity between two songs (e.g., both songs should have the same tempo). Finally, a *global* constraint is defined on any number of positions. For instance, they can express restrictions on cardinality for the entire playlist (e.g., there should be at most two different genres in a playlist) or group a set of unary or binary constraints all together (e.g., all songs should be 'jazz-songs').

By placing the user and her environment central to our research activities, we formulated a list of requirements that we used as input and evaluation criterion for our automatic playlist generation algorithms. Since the algorithms will be used in an interactive music system, demands on *efficiency*, *scalability*, and *optimality* are pressing. Time to compute a playlist should run in a few seconds, since there is a user waiting for the result. The algorithms should scale towards playlists of any length and music collections of any size and any variety. The returned playlist should be optimal and reflect the music preferences of the user, even if these preferences result into conflicting constraints and no playlist exists that meets all expressed preferences.

1.2. End user applications

The algorithm we present is at the heart of a range of interactive music systems, which are used for demonstration purposes and in end user evaluations. These systems allow music listeners to create their personal music playlist without any effort and in a playful interaction style. By manipulating interactive controls in a

Table 1
Song attributes, their description, their type, and an example as used in the evaluation

k	Attribute	Description	Type	Example
1	song ID	Unique identifier	Nominal	101
2	title	Song title	Nominal	Dancing Queen
3	artist	Artist name	Nominal	ABBA
4	album	Song album	Nominal	Greatest Hits
5	genre	Song genre	Nominal	Popular
6	duration	Song length (in seconds)	Numerical	232
7	year	Year of release	Numerical	1975
8	tempo	Tempo of song (in bpm)	Numerical	100

user-system interface, users can parameterize the playlist constraints needed for generation. Specific controls are coupled to a specific set of constraints. Note that the end user is not aware of the playlist generation method hidden behind the interface; she just manipulates some controls and observes the immediate results of her actions.

We show two concrete examples of an interactive music system that make use of the automatic playlist generation algorithm, with two different usage scenarios. The first system aims at usage with a television set and remote control combination [12]. Users can browse through a music collection using a two-panel visualisation, as shown in Fig. 1a. Movement through the panels is done by using the cursor keys on the remote control. The left-hand panel provides the current choices for the user. The right-hand panel displays the consequences of a choice. Selecting an item can be done by pressing the 'ok' button. Color keys can be used to invoke functions like 'reset', 'clear' and 'generate'. While navigating and listening, songs can be added to or removed from a playlist. In addition, a user can select playlist criteria on, for instance,

- the number of songs or duration of a playlist,
- the variety in genres, artists, and albums,
- the tempo and period of release of the songs, and
- the similarity of songs.

These playlist criteria are directly transformed into constraints to generate a playlist.

The second system aims at usage with a web-tablet touch interface. Users can browse through a music collection in a genre-artist-album menu hierarchy by touching the display [15]. The user can put 'chips' in a circular region, as shown in Fig. 1b. Each 'chip' is associated with an aspect like a genre, an artist, or an album. The closer the 'chip' is placed to the centre of the region, the more important the corresponding aspect is to the definition of the playlist. In this way, the variety in genres, artists, and albums can be adjusted according to personal view. Additional criteria on tempo, year of release, and length of the playlist can also be adjusted, if desired. The interaction allows the generation of several versions of the playlist by changing criteria and by fixing, removing, and adding songs to the playlist. Again, criteria for a playlist are transformed into constraints before generation.

1.3. Related work

Generating a music playlist in an automatic fashion has already been researched using different formulations of the problem. Unfortunately, the approaches were not paired with thorough evaluations. Here, we will only review approaches that are comparable to our model and solution.

(a) A TV/remote control-based interactive music system. (b) A web-tablet based interactive music system.

Fig. 1. Interactive music systems using automatic music playlist generation.

Alghoniemy and Tewfik present a network flow approach to playlist generation [4,3]. Nodes in the network represent songs and the arcs are attributed with weighted costs reflecting constraints to be satisfied. Generating a playlist is formulated as an objective to find a path with minimal cost connecting a source node and a sink node in the network (i.e., the first and last song in the playlist). To solve the flow problem, they transformed it into a binary linear program using *branch and bound* [10]. Their method incorporates a notion of optimality. Unfortunately, branch and bound is an exponential algorithm in the worst case.

Pachet et al. [9] use a constraint satisfaction formulation in which desired properties of the playlist are declared by constraints to be met. A standard constraint satisfaction programming (CS) method was used at first [13]. In a CS context, a playlist is modelled as a fixed-sized sequence of constrained variables in which each variable denotes a position in the playlist and has associated a domain of possible songs for that position. In a constructive search, CS assigns songs to variables from their domains in a one-by-one fashion under the condition that all constraints can be completely satisfied. If no songs for a given position are available that meet all constraints, the search backtracks by altering one of its previous assignments and proceeds with the assignment from there on. To improve CS, a wealth of heuristic improvements exists such as *constraint propagation*, *song and variable selection strategies*, and *backtrack methods* [13]. Similarly to branch and bound, CS algorithms have a worst-case exponential complexity. It does not incorporate a notion of optimality. Our own findings were that CS results in an adequate performance in an interactive music system for moderate instances [12], but it is not really scalable.

Aucouturier and Pachet [5] later re-formulate the problem to allow the use of approximating algorithms based on local search (LS) [2] to scale the approach towards very large music collections. Costs are associated with playlists: the more constraints are violated, the higher the cost. The use of this solution method is also the subject of this paper.

1.4. Overview of the paper

In this paper, we will go in-depth into the automatic playlist generation problem, its algorithmic solution, and a comparative evaluation of the algorithm. First, we present a formal model and the computational difficulties of the problem in Section 2. An optimization variant of the problem will be introduced in Section 3, which allows us to use a class of generic approximation algorithms known as local search. We will explain the use of simulated annealing, as a special case of local search, for our optimization problem in Section 4. A study of the problem structure provided us heuristics to improve simulated annealing for our problem. We will devote Section 5 to a thorough evaluation on efficiency and scalability of the developed algorithm by comparing it to a previously developed approach based on constraint satisfaction. Part of this evaluation is a conclusive user test in which we assessed to what extent the playlists of both algorithms were found to be optimal in the eyes of real prospective users. Finally, we will summarize the results, findings, and possible applications of our work in Section 6.

2. A formal model

In this section, we present a formal model of the playlist generation problem. The first element of the model is the input, which is given by a *music collection*. Formally, a song is given by a vector $s = (v_1, \ldots, v_K)$ of attribute values, denoting that the kth attribute of song s has value $v_k \in D_k$. For an example of attributes, see Table 1. Next, a music collection is given by a set $M = \{s_1, \ldots, s_m\}$ of m songs.

The second element of playlist generation is given by its desired output, a *playlist*, which is formally defined by a vector $p = (p_1, \ldots, p_n)$ of length n, where $p_i \in M$ denotes the song at the ith position, for all $i = 1, \ldots, n$. Each song p_i is again a vector of length K, so we can denote attribute k of song p_i by p_{ik}. Although the length n is not specified beforehand, we assume that a lower bound $n_{\min}$ and upper bound $n_{\max}$ are given.

Next, we formally define the constraints that a playlist has to meet. As mentioned in the introduction, we distinguish three types: unary, binary, and global constraints.

A *unary constraint* is a constraint that restricts the choice of songs for one specific position. In its general form, it is given by a triple (i, k, V), for a position $i \in \{1, \ldots, n_{\min}\}$, attribute $k \in \{1, \ldots, K\}$, and value set $V \subseteq D_k$, and it implies that $p_{ik} \in V$ has to hold. For instance, to specify that the first song of the playlist should

be of genre Rock or Alternative, we choose $i = 1$, $k = 5$, and $V = \{\text{Rock}, \text{Alternative}\}$. Note that we do not allow $i > n_{\min}$, as the resulting playlist may not be long enough to have such a position.[2]

To enable a more efficient specification of unary constraints, we introduce the following three specific forms.

- In an *exclude-unary* constraint, we specify a set $W \subseteq D_k$ of forbidden attribute values, meaning that $V = D_k \backslash W$.
- In a *range-unary* constraint, the set of desired values V is given by an interval $[v, w]$, i.e., $V = \{x \in D_k | v \leqslant x \leqslant w\}$. Note that this kind of constraint requires a (partial) order on the involved attribute.
- In a *similar-unary* constraint, the set V is given indirectly by $V = \{x \in D_k | l \leqslant f(v, x) \leqslant r\}$, using a similarity function $f : D_k \times D_k \to [0, 1]$, attribute value $v \in D_k$, and bounds l, $r \in [0, 1]$ on the desired similarity.

A *binary constraint* is a constraint that enforces a relation between the songs at two specific playlist positions. In its general form, it is given by a four-tuple (i,j,k,d), for positions i, $j \in \{1, \ldots, n_{\min}\}$, attribute $k \in \{1, \ldots, K\}$, and function $d : D_k \to 2^{D_k}$, and it implies that $p_{ik} \in d(p_{jk})$ has to hold. The function d is generally not given explicitly, but implicitly as in the following five specific forms of binary constraints.

- In an *equal-binary* constraint, the function d is given by $d(v) = \{v\}$ for all $v \in D_k$. This hence implies that $p_{ik} = p_{jk}$ has to hold.
- In an *inequal-binary* constraint we take the complement, i.e., we have $d(v) = D_k \backslash \{v\}$, implying $p_{ik} \neq p_{jk}$ has to hold.
- In a *smaller-equal-binary* constraint, d is given by $d(v) = \{x \in D_k | x \leqslant v\}$, implying that $p_{ik} \leqslant p_{jk}$ has to hold. Note that this constraint again requires a (partial) order on the attribute.
- Similarly, in a *greater-equal-binary* constraint, $d(v) = \{x \in D_k | x \geqslant v\}$.
- Finally, in a *similar-binary* constraint, the function d is given by $d(v) = \{x \in D_k | l \leqslant f(x, v) \leqslant r\}$, again using a similarity function $f : D_k \times D_k \to [0, 1]$, and bounds l, $r \in [0, 1]$ on the desired similarity. So, this constraint implies $l \leqslant f(p_{ik}, p_{jk}) \leqslant r$.

Thirdly, we have *global constraints*, which pose restrictions on the songs at a number of positions. The set of positions on which a global constraint acts, is denoted by an *interval* $[i,j]$, with $i \in \{1, \ldots, n_{\min}\}$ and $j \in \{1, \ldots, n_{\min}, n_{\max}\}$, which is formally defined as the set $\{l \in \mathbb{N} \mid i \leqslant l \leqslant j \wedge l \leqslant n\}$. Note that if $j = n_{\max}$, then this set depends on the size n of the playlist, and contains at least all positions $n_{\min}, \ldots, n_{\max}$ in the tail of the playlist.

There is no general form of a global constraint, except that it always contains an interval $[i,j]$ as described above, and an attribute $k \in \{1, \ldots, K\}$ on which it applies. In our model, we consider the following seven types of global constraints.

- A *cardinality-global* constraint is given by a five-tuple (i,j,k,a,b), where apart from the interval $[i,j]$ and attribute k a lower bound a and upper bound b are given on the number of different attribute values that are allowed. More specifically, this constraint implies that $a \leqslant |\{p_{lk} | l \in [i,j]\}| \leqslant b$ has to hold.
- A *count-global* constraint is given by a six-tuple (i,j,k,V,a,b), with $V \subseteq D_k$ and $a, b \in \mathbb{N}$, implying that $a \leqslant |\{l \in [i,j] | p_{lk} \in V\}| \leqslant b$ has to hold. In other words, the number of songs in the interval with an attribute value from V should be between a and b.
- Related to the previous one, a *fraction-global* constraint is also given by a six-tuple (i,j,k,V,a,b), with $V \subseteq D_k$, but now with real-valued bounds $a, b \in [0, 1]$, implying that $a \leqslant \frac{1}{|[i,j]|} \cdot |\{l \in [i,j] | p_{lk} \in V\}| \leqslant b$ has to hold. So, the *fraction* of relevant songs should be between a and b.
- A *sum-global* constraint is given by a five-tuple (i,j,k,a,b), with bounds $a, b \in \mathbb{R}$, and it denotes that $a \leqslant \sum_{l \in [i,j]} p_{lk} \leqslant$. Note that it is only defined for numerical attributes.

[2] Or, conversely, if one wants to restrict the song on a certain position i, one has to choose $n_{\min} \geqslant i$.

- An *each-global* constraint is given by a four-tuple (i,j,k,V), with $V \subseteq D_k$, and implies a unary constraint (l,k,V) on all positions in l the interval $[i,j]$, i.e., it denotes that $p_{lk} \in V$ has to hold for all $l \in [i,j]$.
- A *chain-global* constraint is given by a four-tuple (i,j,k,d), with $d : D_k \to 2^{D_k}$, and poses a binary constraint $(l,l+1,k,d)$ on every pair of consecutive songs in the interval. So, it implies that for each $l \in [i,j-1]$ the condition $p_{lk} \in d(p_{l+1,k})$ has to hold.
- Finally, a *pairs-global* constraint is also given by a four-tuple (i,j,k,d), with $d : D_k \to 2^{D_k}$, but this poses a binary constraint (l_1,l_2,k,d) on every pair $l_1,l_2 \in [i,j]$, with $l_1 < l_2$. So, it considers more pairs than a chain-global constraint.

For examples of the global constraints, see Tables 2–4.

Table 2
Constraint set 'user simple'

Description	Constraint
All different artists	pairs-global$(1,n_{\max},3,d(v)=\{x\|x \neq v\})$
From fast to slow	chain-global$(1,n_{\max},8,d(v)=\{x\|x \geqslant v\})$
50% Rock	fraction-global$(1,n_{\max},5,\{\text{Rock}\},.5,.5)$
50% R&B	fraction-global$(1,n_{\max},5,\{\text{R\&B}\},.5,.5)$

Table 3
Constraint set 'user hard'

Description	Constraint
All different songs	pairs-global$(1,n_{\max},1,d(v)=\{x\|x \neq v\})$
Different succ. genres	chain-global$(1,n_{\max},5,d(v)=\{x\|x \neq v\})$
Songs less than 6 min	each-global$(1,n_{\max},6,[0,360])$
Similar succ. tempi	chain-global$(1,n_{\max},8,d(v)=\{x\|\text{sim}(x,v) \in [0,.06]\})$
50% Rock	fraction-global$(1,n_{\max},5,\{\text{Rock}\},.5,.5)$
50% R&B	fraction-global$(1,n_{\max},5,\{\text{R\&B}\},.5,.5)$
8 different artists	cardinality-global$(1,n_{\max},3,8,8)$
$\geqslant$10% Golden Earring	fraction-global$(1,n_{\max},3,\{\text{Golden Earring}\},.1,1)$
$\geqslant$10% Queen	fraction-global$(1,n_{\max},3,\{\text{Queen}\},.1,1)$
$\geqslant$10% The Beatles	fraction-global$(1,n_{\max},3,\{\text{The Beatles}\},.1,1)$
$\geqslant$10% Alan Parson	fraction-global$(1,n_{\max},3,\{\text{Alan Parson}\},.1,1)$
$\geqslant$10% Marvin Gaye	fraction-global$(1,n_{\max},3,\{\text{Marvin Gaye}\},.1,1)$
$\geqslant$10% Prince	fraction-global$(1,n_{\max},3,\{\text{Prince}\},.1,1)$
$\geqslant$10% Stevie Wonder	fraction-global$(1,n_{\max},3,\{\text{Stevie Wonder}\},.1,1)$
$\geqslant$10% Michael Jackson	fraction-global$(1,n_{\max},3,\{\text{Michael Jackson}\},.1,1)$

Table 4
Constraint set 'typical'

Description	Constraint
All different songs	pairs-global$(1,n_{\max},1,d(v)=\{x\|x \neq v\})$
Release in 1980–2001	each-global$(1,n_{\max},7,[1980,2001])$
$\geqslant$20% Stevie Wonder	fraction-global$(1,n_{\max},3,\{\text{Stevie Wonder}\},.2,1)$
$\geqslant$10% Seal	fraction-global$(1,n_{\max},3,\{\text{Seal}\},.1,1)$
$\geqslant$10% Peter Gabriel	fraction-global$(1,n_{\max},3,\{\text{Peter Gabriel}\},.1,1)$
$\geqslant$10% Janet Jackson	fraction-global$(1,n_{\max},3,\{\text{Janet Jackson}\},.1,1)$
$\geqslant$10% Mariah Carey	fraction-global$(1,n_{\max},3,\{\text{Mariah Carey}\},.1,1)$
$\geqslant$20% Phil Collins	fraction-global$(1,n_{\max},3,\{\text{Phil Collins}\},.2,1)$
$\geqslant$40% R&B	fraction-global$(1,n_{\max},5,\{\text{R\&B}\},.4,1)$
$\geqslant$40% Popular	fraction-global$(1,n_{\max},5,\{\text{Popular}\},.4,1)$
2–3 different genres	cardinality-global$(1,n_{\max},5,2,3)$
Different succ. genres	chain-global$(1,n_{\max},5,d(v)=\{x\|x \neq v\})$
Similar succ. tempi	chain-global$(1,n_{\max},8,d(v)=\{x\|\text{sim}(x,v) \in [0,0.1]\})$

Having everything in place now, we can give a formal definition of the playlist generation problem, as follows.

Definition 1 (*Automatic playlist generation problem* (*APG*)). Given a music collection M, a set of constraints C, and length bounds $n_{\min}$ and $n_{\max}$, find a playlist p of $n \in \{n_{\min}, \ldots, n_{\max}\}$ songs from M such that p satisfies all constraints in C.

Without going into details, which can be found in [16], we mention that APG is NP-hard. This is caused by aspects corresponding to four different NP-complete problems [6]. For instance, finding a playlist in which each two consecutive songs are similar is comparable to the Hamiltonian path problem. Next, finding a playlist in which for each attribute the occurring values are different corresponds to the 3-dimensional matching problem. Finding a playlist with a total duration of a certain length corresponds to the subset sum problem. Finally, finding a playlist in which each two songs are sufficiently different is comparable to the independent set problem.

As APG is NP-hard, and as there are several elements that cause this, we opt for a quite generic approximation method. To this end, we first convert APG into an optimization variant in the next section, after which we present a local search technique to find good solutions.

3. An optimization variant

We convert APG into an optimization variant APG-O by introducing a non-negative penalty function that represents the amount of violation of the constraints. Then, instead of searching for a playlist that meets all constraints, we search for a playlist that minimizes the penalty. If the penalty is zero, all constraints are met. Introducing a penalty also overcomes the problem of over-constrained problem instances. In that case, no solution exists that meets all constraints, but the user is still interested to get a playlist, that should meet the constraints as well as possible.

The penalty function we use for a playlist is as follows. First, we define for each of the constraints a penalty function that returns a value from $[0,1]$. Next, the penalty of a playlist is given by a weighted average of each of the constraint penalties. Hence, this will also return a value from $[0,1]$. The weights can be used to give more importance to one constraint over the other. In case of an over-constrained instance, this allows to trade-off different constraints.

Before defining the penalty functions per constraint, we first need to define a difference function between attribute values, which is denoted by $\ominus$, and which will return a value between zero and one. This definition depends on the type of attribute, as follows.

- For a nominal attribute k we can only determine (in)equality between values. Hence, we define the difference between two values $a, b \in D_k$ by $a \ominus b = 0$ if $a = b$, and $a \ominus b = 1$ otherwise.
- For a numerical attribute k there is already a difference function defined, which we make non-negative and normalize to give a result between zero and one. More precisely, the difference is defined by $a \ominus b = \frac{|a-b|}{\max D_k - \min D_k}$.

Also for other types of attributes, not discussed in this paper, a suitable difference function can be defined [16].

Now we can define the constraint penalties. For a unary constraint (i,k,V), we define the penalty as the minimum difference to any element from V, i.e., as $\min\{p_{ik} \ominus v | v \in V\}$. We however make an exception for two of the three specific forms.

- For an exclude-unary constraint, the set V is indicated by its complement $W = D_k \setminus V$, with W typically very small. To prevent very small penalty values due to normalization, we define the penalty for this constraint as 0 if $p_{ik} \notin W$ and 1 otherwise.
- For a similar-unary constraint, the requirement is that $f(v,p_{ik})$ should lie in the interval $[l,r]$. Hence we define the penalty to be equal to the distance to this interval, i.e., the penalty is given by $\min\{|x - f(v,p_{ik})| \, | \, x \in [l,r]\}$. This is comparable to the general definition of unary-constraint penalties, except that we defined it on the co-domain of f, instead of on attribute values directly.

For a binary constraint (i,j,k,d) the requirement is that $p_{ik} \in d(p_{jk})$. The corresponding penalty is therefore defined as $\min\{p_{ik} \ominus v | v \in d(p_{jk})\}$, comparable to unary constraints. Again, we make the following two exceptions to this definition.

- For an inequal-binary constraint we define a penalty of 0 if $p_{ik} \neq p_{jk}$ and 1 otherwise.
- For a similar-binary constraint we again use the similarity function f and bounds l, r in the definition of the penalty, resulting in a penalty $\min\{|x - f(p_{ik}, p_{jk})| \, | x \in [l,r]\}$.

Thirdly, we define the penalties for global constraints, as follows.

- For a cardinality-global constraint (i,j,k,a,b), the number $\gamma = |\{p_{lk} | l \in [i,j]\}|$ of different attribute values is required to lie between a and b, hence we define the penalty by $\frac{1}{\delta} \cdot \min\{|x - \gamma| \, | \, x \in \{a, \ldots, b\}\}$, where the normalization constant δ is given by $\max\{a, |[i,j]| - b\}$ to yield a penalty value between zero and one.
- For a count-global constraint (i,j,k,V,a,b) the number of songs $\mu = |\{l \in [i,j] | p_{lk} \in V\}|$ with an attribute value from V should lie between a and b, so we define the penalty by $\frac{1}{\delta} \cdot \min\{|x - \mu| \, | x \in \{a, \ldots, b\}\}$, with again $\delta = \max\{a, |[i,j]| - b\}$.
- For a fraction-global constraint, with fractional bounds a, b, we define the penalty by $\frac{1}{\delta'} \cdot \min\left\{ | \, x - \frac{\mu}{|[i,j]|} \, | \, | x \in [a,b] \right\}$, with $\delta' = \max\{a, 1 - b\}$.
- For a sum-global constraint (i,j,k,a,b), where the sum $\sigma = \sum_{l \in [i,j]} p_{lk}$ should lie in $[a,b]$, the penalty is given by $\frac{1}{\delta''} \cdot \min\{|x - \sigma| \, | x \in [a,b]\}$. As the minimum possible sum equals $v = |[i,j]| \min D_k$ and the maximum possible sum equals $w = |[i,j]| \max D_k$, we choose the normalization constant $\delta'' = \max\{a - v, w - b\}$.
- An each-global constraint (i,j,k,V) poses a unary constraint (l,k,V) on each position $l \in [i,j]$. The penalty for this constraint is hence defined as the sum of the penalties of the unary constraint on each of these positions, divided by the number $|[i,j]|$ of positions in the interval to obtain again a value between zero and one.
- For a chain-global constraint and a pairs-global constraint (i,j,k,d) we similarly take the sum of the binary constraint penalties of the involved combinations of positions, and normalize it. For the normalization, we again divide the sum by the number of position-combinations, which equals $|[i,j]| - 1$ for a chain-global constraint and $\binom{|[i,j]|}{2}$ for a pairs-global constraint.

4. Local search

The definition of the optimization variant APG-O allows us to solve it with a generic approximation method such as local search (LS) [2,7]. The key feature of local search is that it searches the solution space by iteratively stepping from one solution to a *neighboring* solution, and comparing their quality. A *neighborhood structure* defines which solutions are neighbors to a given solution, which are usually obtained by making small alterations to the given solution. For APG-O, solutions are given by playlists, and the neighborhood structure we use is given in Section 4.2, consisting of replacements, insertions, deletions, and swaps of songs. The cost function we use for the search is obviously given by the total weighted penalty of a playlist, which we denote by $f(p)$.

A solution is called *locally optimal* if there is no neighboring solution with better cost. A solution is called *globally optimal* if there is no solution in the whole solution space with better cost. The objective of APG-O to find such a global optimum, i.e., a playlist with minimal penalty.

4.1. Simulated annealing

Basic LS algorithms like *iterative* (*first and best*) *improvement* were found *not* to be well equipped to solve our problem as they lumped into local optima. Therefore, we consider *simulated annealing* (SA), which incorporates a mechanism to escape from local optima without a need for restarting. In contrast to the basic LS

algorithms, SA replaces the deterministic (strict improving) acceptance criterion by a stochastic criterion, which circumvents the need of an in-depth study of the problem structure in order to design better tailored algorithms.

More specifically, a control variable t is introduced, and the chance of accepting a neighboring solution p' to a given solution p is defined by the acceptance probability

$$\Pr(p'|p) = \begin{cases} 1 & \text{if } f(p') \leqslant f(p), \text{ and} \\ \exp\left(\frac{f(p)-f(p')}{t}\right) & \text{otherwise.} \end{cases}$$

As we can see, the chance of accepting a deteriorating solution depends on the amount of deterioration, as well as the control parameter t. For each value of t, sequences of solutions are generated and evaluated, after which the control variable is lowered by a decrement function. As a result, the chance of accepting deteriorating solutions decreases during the course of the algorithms. For further explanation of SA, we make a forward reference to Fig. 6 for the design scheme of our final algorithm.

A SA algorithm makes use of a so-called *cooling schedule*, which consists of the sequence length of solutions L_k, the initial value of the control parameter t_0, the decrement function used for decreasing t, and a stop criterion. We use a *geometric cooling schedule* that has been successfully applied to many problems described in literature. For APG-O, this results in a choice of $L_k = 10$, $t_0 = 1$, decrement function $t_{k+1} = 0.9 \cdot t_k$, and stop criterion $f(p) = 0$, i.e., we stop only if all constraints are satisfied. Note that, as at all times we have a playlist, we can abort SA at any time.

4.2. Neighborhood definition

What remains is the definition of a neighborhood structure, which describes how to move from one solution to another. For the neighborhood, we defined the following four types of moves.

A *replace move* chooses a playlist position and a new song from the music collection and replaces the song that is at that position by the new song, as shown in Fig. 2.

An *insert move* chooses a position in the playlist and a new song from the music collection and inserts that song into the playlist at the chosen position; see Fig. 3. Insert moves are only allowed on playlists that have fewer than $n_{\max}$ songs to ensure that the resulting playlist is not too long.

A *delete move* chooses a position in the playlist and removes the song at that position, as shown in Fig. 4. Delete moves are only allowed on playlists that have more than $n_{\min}$ songs.

Finally, a *swap move* chooses two positions in the playlist and swaps the songs that appear at these positions, as shown in Fig. 5.

Each of the above four types of moves defines a neighborhood. The complete neighborhood is given by the union of these four neighborhoods. To balance the selection of the four individual neighborhoods for

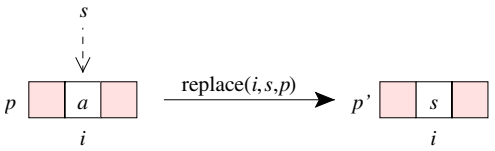


Fig. 2. A replace move example. The song at the ith position in p, which is a, is replaced by song s.

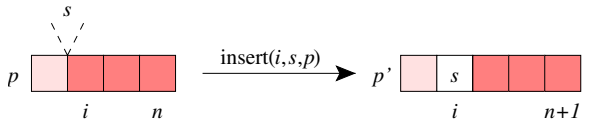


Fig. 3. An insert move example. Song s is inserted into playlist p at the ith position, resulting in a playlist with one song more.

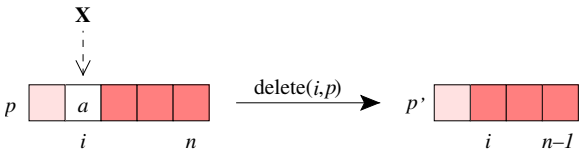


Fig. 4. A delete move example. The song at the *i*th position of playlist *p* is removed, resulting in a playlist with one song fewer.

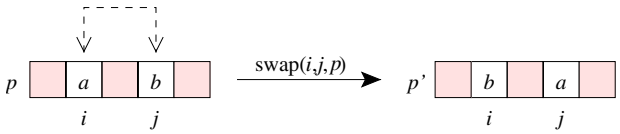


Fig. 5. A swap move example. The song at the *i*th position in *p*, which is *a*, is replaced by the song at the *j*th position, which is *b*, and vice versa.

generating a new solution in our SA algorithm, we introduce *probability weights* w_{replace}, w_{insert}, w_{delete}, and w_{swap}, which determine the probability of performing a particular type of move. Determining an optimal set of weight values will be addressed in Section 5.2.

As the moves described above make small modifications to a playlist, the changes in penalty function can be calculated incrementally, and thus more efficiently.

4.3. Heuristic improvements

In order to increase the performance of SA, that is, to find better playlists faster, we propose three heuristic improvements based on the various types of constraints in APG-O: song domain reduction, a two-level neighborhood, and partial constraint voting.

4.3.1. Song domain reduction

Song domain reduction resembles a form of constraint propagation to guarantee node consistency for unary constraints used in constraint satisfaction methods [13]. To this end, we denote a song domain M_i of a position i as the subset of the music collection, $M_i \subseteq M$, that defines the possible songs that are allowed at that position; for a playlist p, it should hold that $p_i \in M_i$. By reducing the song domains M_i, we can in this way trim the search space for our LS. If a position is not over-constrained, the reduction is basically established by removing all songs from a given song domain that do not meet all unary constraints that are declared for a given problem instance. We have developed different reduction mechanisms for all four individual neighborhood structures.

4.3.2. Two-level neighborhood structure

As mentioned in the introduction, APG is about choosing the 'right' songs and putting them in the 'right' order. The former aspect is mostly described by the first five global constraints, if defined on the entire playlist. An example of this is the constraint that there should be at most three different genres. The penalty of such a constraint is not affected by swap moves. Other constraints, such as unary, binary, and the last two global constraints, are affected by the song order.

Based on the above two aspects, we make a distinction between replace, insert, and delete moves on the one hand (which we combine in a *reselect* neighborhood), and swap moves on the other hand. Next, we introduce a so-called *two-level neighborhood*, splitting the search into two procedures, that are applied alternatingly. The first one consists of a sequence of β reselect moves, that attempts to get the right set of songs. Next, a sequence of swap moves is applied to put the songs in the right order. For the latter, we employ a simple procedure, called *non-deteriorating reordering* (NDR), which applies iterative improvement with a maximum of γ swap moves. The selection of β and γ is discussed in Section 5.2.

4.3.3. Partial constraint voting

Simply applying random moves at randomly chosen playlist positions in our neighborhoods leads to an inadequate coverage of the restrictions as posed by some global constraints, notably the *cardinality-global*, the *count-global*, and the *fraction-global* constraints. We have taken special measures to account for the fact that these constraints need specific types of songs at specific positions for their fulfillment. In particular, we apply a voting mechanism in which all constraints vote for or against the possible playlist positions in a move and, in the case of the replace and insert neighborhood, the song. Every constraint can cast a positive vote for a position with a song in a given solution that contributes to its violation to indicate that it wants to replace that song. On the other hand, a constraint can cast a negative vote for a position with a song that helps in the satisfaction of the constraint to indicate that the song should stay at that position. The votes from all constraints are tallied, and a playlist position is chosen biased by these votes. So, a position with many positive votes and a few negative votes is more likely to be chosen than a position with fewer positive votes and more negative votes. If the replace or the insert neighborhood was selected, we also have to vote for a specific song to be added to the playlist. To increase efficiency, we only let the above-mentioned global constraints vote for songs to direct the song selection towards their satisfaction. Therefore, we call this heuristic improvement *partial constraint voting*. Again, the song votes are tallied and one song is chosen, biased by the collected votes.

Though this voting mechanism is effective in directing satisfaction of the mentioned global constraints, it is computationally intensive in comparison to random selection of positions and songs. Therefore, we limit its use to a fraction δ of the reselect moves.

4.3.4. Final algorithm

Applying the heuristic improvements to a standard SA algorithm results into the algorithm as depicted in Fig. 6.

5. Evaluation

This section presents the evaluation of our algorithmic solution to APG-O. First, we set the algorithmic parameters in Section 5.2. Next, we present tests on the scalability issues of our solution and a conclusive user

```
INITIALIZE p, t_0, L_0;
k := 0;
r := 0;
repeat
    for l := 1 to L_k do
    begin
        if r < β then
        begin
            if δ > random[0,1) then GENERATE RANDOM p' ∈ N_reselect(p)
            else GENERATE p' ∈ N_reselect(p) BY VOTING;
            if f(p') ≤ f(p) or exp((f(p)−f(p'))/t) > random[0,1)
            then p := p';
            r := r + 1
        end
        else begin
            p := NDR(p, γ);
            r := 0
        end
    end;
    k := k + 1;
    CALCULATE LENGTH L_k;
    CALCULATE CONTROL t_k
until STOP CRITERION
```

Fig. 6. The resulting simulated annealing algorithm for APG-O, including the discussed heuristic improvements.

study in Sections 5.3 and 5.4, respectively. Our solution method was compared with a constraint satisfaction (CS) algorithm. All tests were performed using a PC with a 3 GHz Pentium 4 processor and 1 GB of RAM, running the Windows XP operating system. The algorithms were implemented in Java.

5.1. Music collection

A set of 2248 popular music songs from 169 albums from 111 different artists covering 7 different musical genres released in the period from 1963 to 2001 served as the music collection for all tests in the current evaluation. Table 1 in the introduction shows an overview of the eight song attributes that were used to describe the songs.

5.2. Algorithmic parameters

The first tests were to arrive at a set of values used in our heuristic extension to the standard SA algorithm, which we chose such that we obtained a minimal mean run time of the LS algorithm satisfying a pre-defined set of constraints. These constraints were constructed by having user intentions for everyday music listening in mind, as we had identified in a previous user study [12]. We refer to Tables 2–4 for typical examples of the constraint set used. Various settings have been tried by Vossen [16]; here we only present the resulting choices.

The weights w_{replace}, w_{insert}, and w_{delete} represent the chance that one of their corresponding neighborhoods is chosen for changing songs in the current playlist. The algorithm outperforms in mean run time and has smallest standard deviation around the mean when all three weights have equal proportions, that is, are set to 1/3, in comparison to various other weight settings.

The next two parameters control the exploitation of the two-level neighborhood structure. The first, β, determines the number of reselect moves (i.e., insert, replace, and delete moves) that is investigated between two successive calls of the song reordering (i.e., swapping) procedure NDR. The second parameter, γ, determines the number of swap moves that are investigated in the NDR procedure. If β is too low, too few reselect moves were performed between swap moves. This is beneficial to constraints that declare a particular song ordering, but severely hamper all other constraints in a given problem instance. On the other hand, a low value for γ results in too few swap moves that can change the set of songs. We settled in assigning both parameters the value of 100, which was found to be a good compromise.

The last parameter, δ, represents the portion of using partial constraint voting for generating new solution candidates. Since voting evidently takes up more computing resources than randomly generating solution candidates, it should not be used to intensively. Nevertheless, voting helps in satisfying some hard global constraints. The lowest run times were achieved with a δ value of 0.3.

The above parameter setting results in a typical run time of the algorithm of 2 s to generate a playlist of one hour (i.e., 10 songs) using our music collection.

5.3. Tests on scalability

Prime concern of the current research was, on the one hand, to scale APG towards longer playlists and larger music collections, and, on the other hand, to make APG also applicable in situations where conflicting constraints were posed. To test for scalability, we compared the performance of our LS solution to the performance of the more traditional constraint satisfaction solution. To allow for a direct comparison between both solution methods, we restricted the playlist to be of fixed length (i.e., $n_{\min} = n_{\max}$) and we employed three problem instances with feasible solutions, as shown in Tables 2–4. The aim of the scalability tests was to assess the effect on the run time performance of both solution methods on the length of the playlist and the size of the music collection.

5.3.1. Method

In the first set of tests, the playlist length varied from 10 to 30 songs in steps of two songs. The size of the music collection was held constant at 2248 songs.

In the second set of tests, the size of the music collection was varied from 2248 songs up to $30 \cdot 2248 = 71{,}194$ songs, in 13 multiplicative steps. Since we had only 2248 songs with attributes at our disposal, the music collection was artificially extended with generated songs based on statistics of the original song collection. The length of the playlist was held constant at 10 songs.

For all tests, we performed 10-fold estimates on mean run time and its standard deviation by conducting ten runs for each problem instance. If the run time of a single run of a test surpassed 200 s, the test was aborted and it was stated that the run did not return a (complete) solution to the given problem instance.

5.3.2. Results

The results of varying the playlist length are shown in Fig. 7. In general, the graphs for the LS algorithm show an approximately linearly increasing run time as function of the playlist length. In contrast, the mean run times of the CS algorithm show a more erratic behavior over different playlist lengths.

For the constraint set 'user simple', CS is clearly outperformed by LS. From a playlist length of 14 onwards, CS is not able to return a solution within the time allowed (i.e., 200 s). In contrast, the LS algorithm finds a solution at all times within 10 s for even a playlist of 30 songs. The tempo ordering constraint makes the problem instance hard to solve for the CS algorithm, as it requires a lot of backtracking.

The constraint set 'user hard' was considerably harder to solve for the LS algorithm. It was found to be insufficiently equipped to handle the tempo similarity constraint. However, due to this constraint, the mean run time of the CS algorithm shows a more erratic behavior than the mean run time of the LS algorithm. For three playlist lengths (10, 16, and 22), CS was able to find swiftly a solution, though at highly varying run times, reflected by the large standard errors. For all other lengths, it was outperformed by LS.

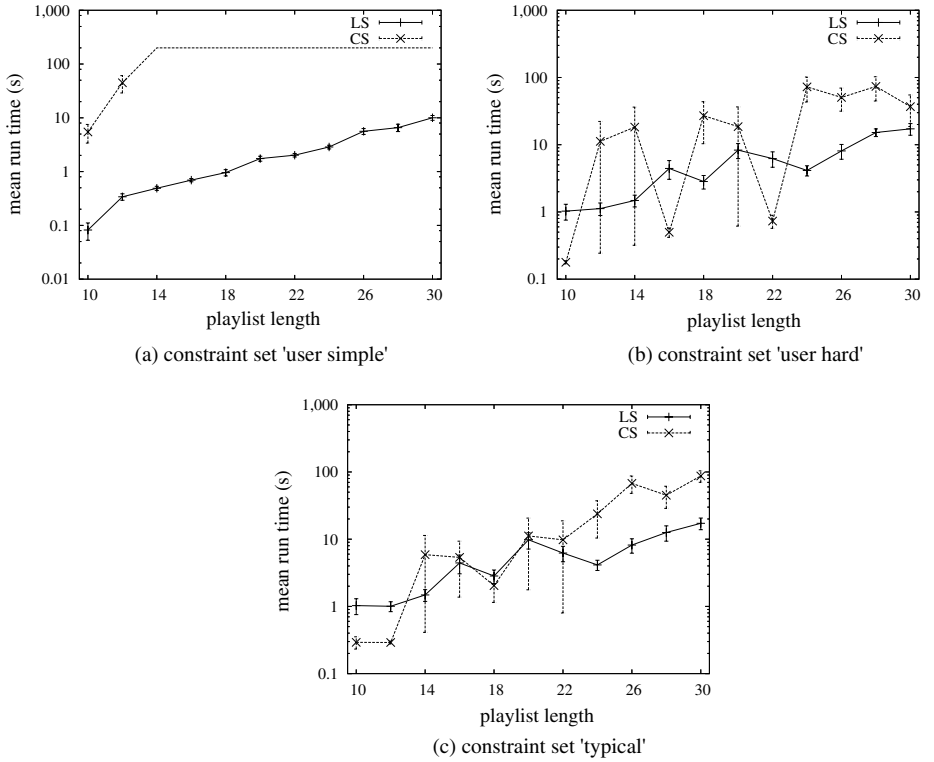


Fig. 7. Mean run time (in seconds) over 10 runs as a function of the playlist length for various constraint sets and both algorithms (LS: local search by simulated annealing; CS: constraint satisfaction). Cross-bars represent standard errors of the mean. Note the logarithmic scale of the vertical axis; this explains also the asymmetric lengths of the cross-bars. The straight line at 200 s for the CS algorithm in figure (a) indicates aborted test runs.

In contrast to the previous constraint sets, the constraint set 'typical' did not feature a constraint on tempo ordering/similarity. Instead, a constraint declaring dissimilar genres for successive songs was included. Consequently, both algorithms experience less difficulty for solving these instances. The run times of CS were even lower than the run times of LS for short playlists (10 and 12), but if the playlist was lengthened, the comparison in run time performance reversed in advantage to the LS algorithm.

The results of varying the music collection size are shown in Fig. 8. The run times of both algorithms increase more or less linearly in the size of the music collection. However, the CS algorithm was unable to generate a playlist for music collections larger than 22,875 songs. This was due to lack of computer memory resources, as CS needs to hold all partial solutions for backtracking purposes. No swapping/paging strategy to secondary memory was implemented to solve this issue. The LS algorithm has a far more efficient policy in memory usage, since it only inspects a single solution at a time. It still managed to solve the problem instance at music collection sizes of 71,194 songs, though at relatively extended run times (i.e., 27–40 s).

5.4. Quality evaluation

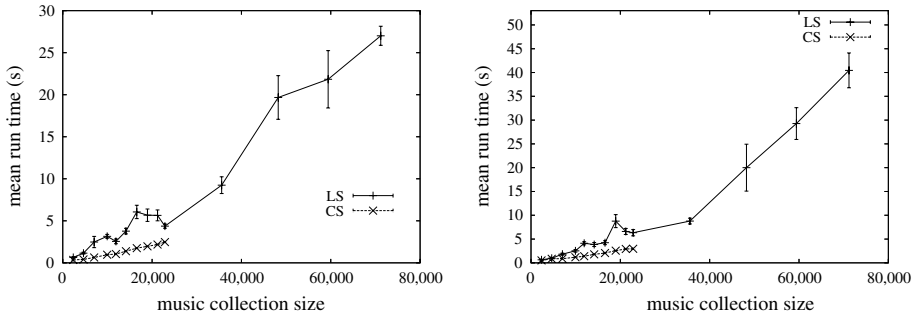
To test for playlist optimality, we compared the penalties of the playlists returned by the LS and the CS algorithm after 5 s of run time for the three sets of constraints as given in Tables 2–4. For playlists varying from 10 to 30 songs, LS outperformed CS in all cases. For the constraint set 'user simple', CS was only slightly better than an algorithm in which songs were randomly drawn from the music collection. To continue the test, we involved real prospective users in the evaluation process.

In this user evaluation, we asked eighteen participants (22–41 years; 13 men, 5 women) with no specific musical skills to assess the subjective quality of music playlists generated by the LS algorithm and the CS algorithm, again after 5 s of run time.

5.4.1. Method

From the music collection, different playlists were automatically generated in a complete factorial design using both *algorithms* (CS and LS), two *playlist lengths* (10 and 20 songs), two *problem sizes* using four sets of constraints (two sets of four constraints and two sets of fifteen constraints). We used the constraint sets 'user simple' (see Table 2) and 'user hard' (see Table 3) and variants, which only differed in the set of desired genres and artists.

For each participant, eight playlists including song information and their descriptions were contained in a questionnaire according to the factorial design of the experiment. This description reflected the set of constraints that was used to generate the playlists in well-formed sentences of natural language. Participants could listen to the songs at will using a web browser. Their main task was



(a) constraint set 'user hard' (b) constraint set 'typical'

Fig. 8. Mean run time (in seconds) over 10 runs as a function of the music collection size for various constraint sets and both algorithms (LS: local search; CS: constraint satisfaction). Cross-bars represent standard errors of the mean. Because the implementation of the CS algorithm did not allow for music collections of more than approximately 22,000 songs, we have no data for those instances.

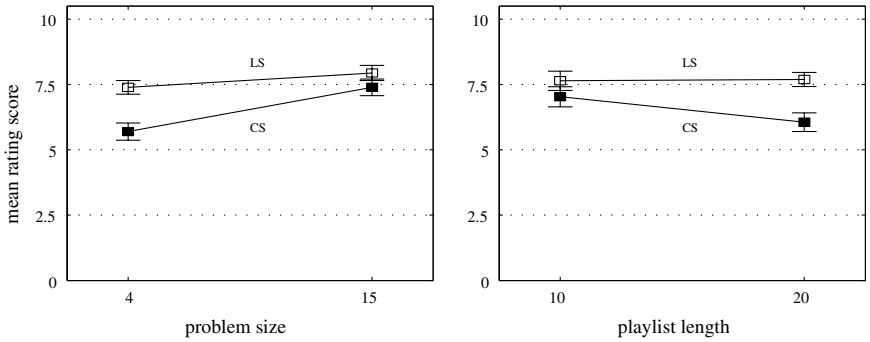


Fig. 9. Mean *rating score* across different *algorithms* (LS: local search; CS: constraint satisfaction), different *problem sizes*, and different *playlist lengths*. Cross-bars represent standard errors of the mean.

(1) to mark what sentences in the playlist description did not fit the actual playlist.
(2) to mark what songs in the playlist did not fit the original playlist description.
(3) to provide a rating between 0 (extremely bad) to 10 (extremely good) on how well the playlist reflected the playlist description.

We allowed the participants to have different interpretations of genres and tempo. They were explicitly instructed not to rate the music according to their personal preferences. They were not informed about the purpose of the experiment.

Using the data from the questionnaires, playlist quality was measured by the proportion of constraints violated (i.e., violation), proportion of songs that suited the description (i.e., precision), and a rating score.

5.4.2. Results

All data were subjected to an ANalysis Of VAriance (ANOVA) with repeated measures in which *algorithm*, *playlist length*, and *problem size* were treated as within-subject variables to assess any statistically significant effects [8]. In general, it was found that playlists generated by the LS algorithm were judged to be of statistically significant higher quality than those generated by the CS algorithm.

In particular, averaging over playlist length and problem size, participants indicated that proportionally more constraints were violated by the playlists generated by CS than by the playlists generated by LS (mean violation for LS playlists: 0.23; mean violation for CS playlists: 0.36; $F(1,17) = 14.6$, $p < 0.005$).

Also, averaging over playlist length and problem size, playlists generated by LS had a significantly higher precision than playlists generated by CS (mean precision for LS playlists: 0.85; mean precision for CS playlists: 0.78; $F(1,17) = 9.4$, $p < 0.01$).

Finally, averaging over playlist length and problem size, playlists generated by LS were rated significantly higher by more than a full rating scale degree than playlists generated by CS (mean rating score for LS playlists: 7.7; mean rating score for CS playlists: 6.5; $F(1,17) = 56.6$, $p < 0.001$). Fig. 9 shows the mean rating score for both *algorithms*, both *problem sizes*, and both *playlist lengths*. The difference in mean rating scores between the two algorithms decreases as the problem size is increased ($F(1,17) = 13.0$, $p < 0.005$). We conjecture that when more constraints are present in a problem instance, the violation of one constraint is not found as severe as when fewer constraints are present. Also, we see that, while mean rating scores for LS playlists are equal for both playlist lengths, the mean rating score for CS playlists decreases from around 7 to 6 ($F(1,17) = 7.6$, $p < 0.05$).

6. Conclusion

We introduced a formal model and an optimization variant of the automatic music playlist generation problem and designed an optimization algorithm for it based on simulated annealing. The main requirements for the algorithm were concerned with efficiency, scalability, and optimality. In a set of tests, the algorithm was

shown to be an improvement over a previously designed algorithm based on constraint satisfaction on all three requirements. We found that the typical run time of the algorithm was about 2 s on a PC platform for playlists of at most 14 songs, a music collection of two thousand songs, and various problem instances. Enlarging the playlist had only a linear impact on the run time. As indicated by real users, the playlists of the new algorithm had a higher subjective quality than those of constraint satisfaction. The requirement for scalability needs further attention, since run time of the algorithm for very large music collections (more than 71,000 songs/6000 albums) became unacceptably high for interactive use (i.e., more than 30 s). Though it is not expected that end-users will own personal collections of that size, online music services already provide an active catalogue of one million songs. Adding a pre-processing step to our algorithm, in which a reasonable and relevant subset of the entire music collection is chosen first, would be a good candidate to resolve this scalability issue.

The presented algorithm has already been embedded in interactive music prototype systems designed for consumer electronic devices. In usability studies, we found that the novel, instant, and easy way of creating a music playlist is well-appreciated by end-users [11,12,15]. It opens up completely new methods to them to use, (re-)discover, and experience their personal music by the art of re-combining songs in various, previously unforeseen ways. There are numerous other applications for automatic music playlist generation as well. For instance, an automatic DJ-ing application requires that songs are first ordered on meter, tempo, key and the like before they are mixed one after the other. Online music sales application can be easily augmented with a service to automatically compile and download a personal album. Music streaming and broadcasting can excel using on-the-fly generation of music programming allowing truly personal and interactive (Internet) radio and podcasting. Lastly, other media like photos can be automatically sequenced for a slide show using a set of pre-defined constraints.

References

[1] E.H.L. Aarts, S. Marzano, The New Everyday: Views on Ambient Intelligence, 010 Publishers, Rotterdam, The Netherlands, 2003.
[2] E.H.L. Aarts, J.K. Lenstra, Local Search in Combinatorial Optimization, Wiley, 1997.
[3] M. Alghoniemy, A.H. Tewfik, A network flow model for playlist generation, in: Proceedings of the IEEE International Conference on Multimedia and Expo 2001 (ICME 2001), Waseda University, Tokyo, Japan, August 22–25, 2001.
[4] M. Alghoniemy, A.H. Tewfik, User-defined music sequence retrieval, in: Proceedings of the 8th ACM International Multimedia Conference, Los Angeles, USA, November 2000.
[5] J.-J. Aucouturier, F. Pachet, Scaling up music playlist generation, in: Proceedings of the IEEE International Conference on Multimedia and Expo 2002 (ICME2002), Swiss Federal Institute of Technology, Lausanne, Switzerland, August 26–29, 2002.
[6] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York, 1979.
[7] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[8] S.E. Maxwell, H.D. Delaney, Designing Experiments and Analyzing Data: A Model Comparison Perspective, Brooks/Cole Publishing Company, CA, USA, 1990.
[9] F. Pachet, P. Roy, D. Cazaly, A combinatorial approach to content-based music selection, IEEE Multimedia 7 (1) (2000) 44–51.
[10] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Inc., 1982.
[11] S.C. Pauws, J.H. Eggen, Realization and user evaluation of an automatic playlist generator, Journal of New Music Research 32 (2) (2003) 179–192.
[12] S.C. Pauws, S. van de Wijdeven, User evaluation of a new interactive playlist generation concept, in: J.D. Reiss, G.A. Wiggins (Eds.), Proc. Sixth International Conference on Music Information Retrieval (ISMIR2005), 11–15 September 2005, pp. 638–643.
[13] E.P.K. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.
[14] A. Vasilakos, W. Pedrycz, Ambient Intelligence, Wireless Networking, Ubiquitous Computing, ArtecHouse, USA, 2006.
[15] F. Vignoli, S.C. Pauws, A music retrieval system based on user-driven similarity and its evaluation, in: J.D. Reiss, G.A. Wiggins (Eds.), Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR2005), 11–15 September 2005, pp. 272–279.
[16] M.P.H. Vossen, Local search for automatic playlist generation, M.Sc. thesis, Technische Universiteit Eindhoven, The Netherlands, 2005.