An Interface for Melody Input

Lutz Prechelt and Rainer Typke
University of Karlsruhe

We present a software system, called Tuneserver, which recognizes a musical tune whistled by the user, finds it in a database, and returns its name, composer, and other information. Such a service is useful for track retrieval at radio stations, music stores, etc., and is also a step towards the long-term goal of communicating with a computer much like one would with a human being. Tuneserver is implemented as a public Java-based WWW service with a database of approximately 10,000 motifs. Tune recognition is based on a highly error-resistant encoding, proposed by Parsons, that uses only the direction of the melody, ignoring the size of intervals as well as rhythm. We present the design and implementation of the tune recognition core, outline the design of the web service, and describe the results obtained in an empirical evaluation of the new interface, including the derivation of suitable system parameters, resulting performance figures, and an error analysis.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*Error Control Codes*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Information Processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Selection Process*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services; H.5.4 [**Information Interfaces and Presentation**]: User Interfaces—*Interaction styles*; I.2.m [**Artificial Intelligence**]: Miscellaneous; I.5.4 [**Pattern Recognition**]: Applications—*Signal Processing*; K.4.m [**Computers and Society**]: Miscellaneous

General Terms: Algorithms, Human Factors, Performance

Additional Key Words and Phrases: input mode, melody, motif, recognition, theme, tune

1. ON INPUT MODES

Designing computer interactions so that they become human-like is one challenge of computer science research. We call this problem the *interaction challenge* and shall view it from the input perspective. Although most human interaction uses only two input channels, namely the auditory and the optical systems, a wide range of types of information is transmitted through them: the channels are used in many different *modes*. The support of all the various input modes of human interaction is one aspect of the interaction challenge. This article contributes to the set of input modes available for use with computers by presenting the design, implementation, and evaluation of Tuneserver, a melody recognition system that acts as a robust listening expert for musical themes.

1.1 Current input modes

Current research on direct human input (i.e., input not involving the explicit operation of hardware such as a keyboard or mouse or special-purpose devices [Jacob 1996]) focuses on input modes that have relatively broad applicability in business,

Address: Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany
E-mail: lutz@prechelt.de, rainer@typke.de,
http://wwwipd.ira.uka.de/Tichy/

engineering, or general computer use contexts. Some of them have already matured into commercial applications, while research on others has just begun.

The oldest direct input mode is probably speech recognition [Cole et al. 1995]. Attempts to boost the performance of speech recognition by using additional techniques like lip-reading [Stork and Hennecke 1996] have been under way for some time. For difficult words, such as special names, oral spelling or online handwriting [Manke et al. 1995] might prove helpful.

Gesture, posture, and detection of gaze-direction are used to identify the targets of a verbal (or other) utterance or the focus of attention [Stiefelhagen et al. 1997]. Detection of emotions in verbal utterances [Dellaert et al. 1996] or in mimics accompanying them allows for more accurate interpretation and reaction. General handwriting, sketching, and screen gestures are further input modes already being applied, e.g., in palmtop computers.

The user should be able to choose and mix these input modes as he sees fit, resulting in *multi-modal input*, which will become increasingly important in the future [Myers et al. 1996]; see [Benoit et al. 1999] for an overview. For achieving human-like interaction with a computer, the raw input processing should be supported by appropriate understanding and control intelligence. However, the time when computers will interact in a truly human way with their users is still in the future. For instance, Reddy [Reddy 1995] considers a translating telephone to be a “Grand Challenge in AI Megaproject”.

1.2 Melody input

The research work we present here adds another channel of computer-human interaction that may help with quick track retrieval at radio stations, make audio-on-demand WWW services easier to use, improve service at audio media stores or, in the far future, serve as another part of the human-like interaction capabilities of commodity computers.

The purpose is to process audio input signals (such as singing, humming, or whistling) that represent a melody, and to use a robust approximate representation of the input melody as a search key for retrieving whatever information is connected to it in a database. We call our implementation “Tuneserver”. See Section 4 for a discussion of related work.

Melody recognition is superficially similar to speech recognition, but there are important differences. On one hand, the signal structure is simpler, since there are no phonemes. At later stages, the information processed in melody input is far less ambiguous and, again, simpler in structure. On the other hand, melody recognition is also quite difficult, because, as we will see below, average humans are not very good at providing accurate input.

1.3 Usage scenarios

The most important obstacle towards wide deployment and use of Tuneserver functionality is the availability of the database contents required for finding a tune. As described in Section 2.2, a few bytes of special encoding describing the melody must be stored there for identifying a tune, linked to whatever shall be retrieved: the actual audio signal of a recording, audiographic information such as title, performer, duration, composer, publisher, ordering information etc. However, once

Tuneserver technology is reliable, it may become popular enough for music publishers to include this information on a small data track on each CD, and service providers might provide databases covering older music titles.

Given such infrastructural support, one can imagine the following scenarios: Adam has heard an old tune on the radio and wants to buy it on CD. In earlier days, he wouldn't even have tried, because, like most people, he is much too shy to sing, hum, or whistle the melody to a salesperson in a music store (and even people who dared often ended up with just a shrug). Today, he walks to the store, goes to the "Find your music" box in a quiet corner (much like a phone booth), hums secretly into a microphone, and is presented with a list of 27 possible recordings. To reduce the selection, he enters additional information from menus: "single female singer" and "recording is older than 5 years". Two titles remain, and he listens to each of them. He ends up buying both of them; one is what he was looking for, and the other one he likes as well.

Carla is a radio moderator. She routinely uses Tuneserver to retrieve titles that are stored in the online digital music archive. Since pure speech recognition input has been tried before by her radio station, and has proven to be too inefficient for music track retrieval, the station now uses Tuneserver in combination with a rather simple and robust speech recognition engine. Hence, she can select and play a title on the air purely by voice command, without handling a CD or a keyboard. At one point, she uses the keyboard for a text-based search for a specific cover version of "With a little help from my friends" that she knows her station has on a vinyl record, but none of her database queries leads to the desired result. Frustrated and ready to give up, she decides to try Tuneserver. She hums the melody, obtains 14 tracks, and among them finds the one she wanted: It is stored as "Help from friends"! Later she finds that this is indeed the title printed on the record for this track.

1.4 Article overview

In the following section, we will describe the overall system architecture of Tuneserver and the design and implementation issues that arise. In particular, we will describe the recognition algorithms and the optimization of their adjustable parameters. Section 3 presents and analyzes the results of an empirical study based on over 100 recordings from 24 different persons. We identify the factors that limit Tuneserver's performance and analyze their contributions. Subsequent sections discuss closely related work and possible future system improvements.

2. DESIGN AND TECHNIQUES

We first provide an overview of the system parts and how they interact. Then we describe the melody encoding, called Parsons code [Parsons 1975], that underlies Tuneserver. The core procedure, converting the sound signal into the Parsons code, is described in Section 2.3, and values for its parameters are derived in Section 2.4. Finally, a derivation of the distance function used to rank the database entries during the search process follows in Section 2.5, followed by a discussion of its parameters in Section 2.6. Readers not interested in the technical details may want to skip the later subsections and only read 2.1 and 2.2.

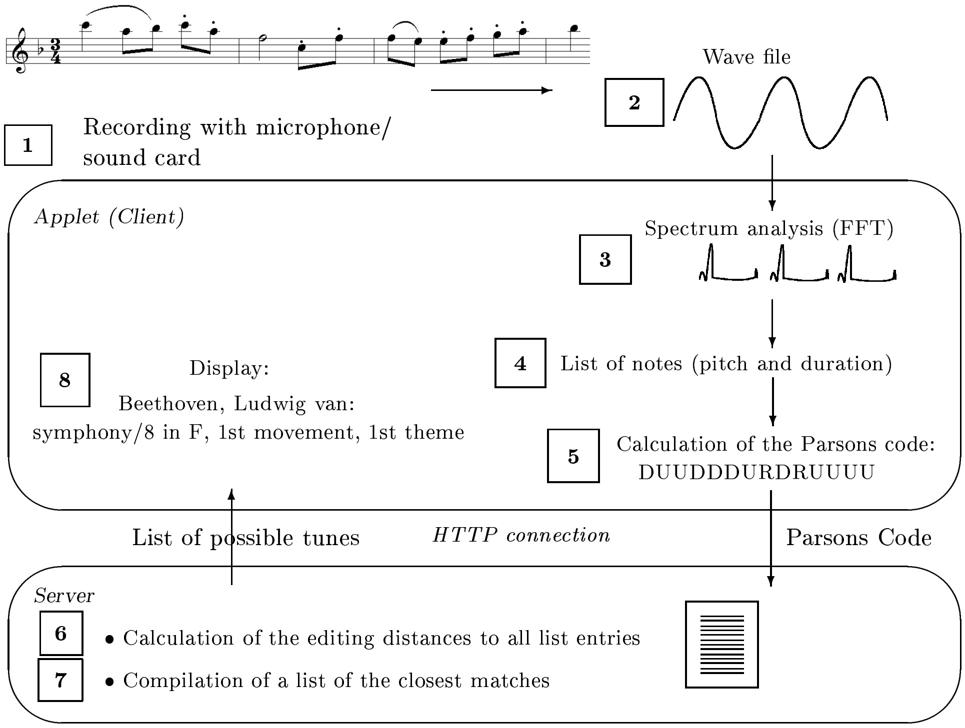

Fig. 1. Overview of the system architecture and the querying process. The large bottom box represents the server, the large middle box represents the client applet. The small boxes containing numbers indicate the sequence of events, arrows indicate data flow.

2.1 System architecture overview

Tuneserver is split into a client, implemented as a Java applet, and a web server. The latter is also written in Java, taking advantage of the language's internet features, but uses a core written in C++.

The main routine that converts sound into the Parsons code (described below) is located in the client. Hence, only a few bytes of Parsons code (typically about 5 to 30) need to be transfered to the server instead of an actual audio file of possibly several hundred kilobytes.

The web server accepts Parsons code from a client and responds with a ranked list of those musical themes from its database whose Parsons codes are most similar according to an appropriate distance function. See also Figure 1 for an overview. Tuneserver's database consists of 10,370 entries of classical music from the collection of themes published in Parsons's book [Parsons 1975]. Tuneserver can be found at http://wwwipd.ira.uka.de/tuneserver/.

2.2 Error-resistant encoding: The Parsons code

Tuneserver should work for everybody. However, especially with untrained users, there is the problem of input inaccuracies. Most people tend to whistle out of tune, some change key in the middle of a theme, and many use inaccurate rhythm. As a result, a system that uses rhythm and note intervals for analyzing the user input would have to deal with many potential problems.

Parsons showed that a simple encoding of tunes that ignores most of the information in the musical signal can still provide enough information for distinguishing between a large number of tunes. The Parsons code reflects only the directions of melodies. Each pair of consecutive notes is coded as "U" ("up") if the second note is higher than the first note, "R" ("repeat") if the pitches are equal, and "D" ("down") otherwise. Rhythm is completely ignored. Thus, the first theme from the last movement of Beethoven's 9th symphony (Ode to Joy) would be coded RUURDDDDRUURDR.[1] Note that the first note of any tune is used only as a reference point and does not show up explicitly in the Parsons code at all.

We use the Parsons code for representing tunes, thus ignoring all sound information except for the directional pattern. This approach makes our system error-tolerant with respect to both time and frequency. What remains is the problem of deciding where notes start and end, and whether a given pair of notes should be considered equal or not.

The following subsections cover technical details of the recognition mechanism. If you are not interested in these, just skip to Section 3.

2.3 Converting sound into the Parsons code

We decided to focus on whistled input (as opposed to singing or humming) because the sound of whistling varies much less among people than singing; it is unrelated to gender, for example. Furthermore, the frequency spectrum of whistling is far simpler than that of other vocal emissions, which makes analysis easier.

We first make a digital recording, sampled with 8 bit at 11 kHz. Higher resolution is not required, because, as we will see below, we will use only very little of the actual acoustic information. We group the sequence of samples into overlapping windows, ignoring the very first t_{skip} milliseconds to eliminate start noises (see Table 1 for t_{skip} and other user-adjustable parameters of the system). The window size is 46 ms and each window overlaps half of the previous one and half of the next one. We smooth the window margins using a cosine function. There are other common smoothing functions that would work just as well, because only coarse spectrum differences are relevant for the Parsons code. Then we apply a Fast Fourier Transform to obtain frequency spectrum samples, one per window.

The next step is to take the maximum amplitude (the peak) from each spectrum; we regard the associated frequency as the pitch whistled during this window and the associated amplitude as the volume of the window. If, however, the maximum is not significantly higher than the rest of the spectrum, we mark the window as "silence" by setting the volume to zero. To decide this, we compare the quotient of the highest peak and second highest peak to the fixed threshold r_{peaks}. Thus, we obtain a sequence of frequency-volume pairs, one per FFT window. This relatively simple approach is sufficient for pitch determination, because a whistling signal contains only one single base frequency plus overtones, and hence a unique pitch.

[1] More than fault tolerance, the aim was to enable people who cannot read notes and do not know much about music to find the title of a given theme. Before Parsons's book had been published, the easiest way of doing this was to transpose the theme into the key of C and then write down the actual notes and look them up in the dictionary of Barlow and Morgenstern [Barlow and Morgenstern 1948].

parameter	meaning	value
t_{skip}	Skip the first	10ms
$rvol_{min}$	Ignore everything below	5% of rec. max. volume
f_{min}	Ignore notes below	400 Hz
r_{peaks}	Ignore spectra w. $\frac{1st}{2nd}$ peak ratio below	2
$rvol_{min,note}$	Split note if volume drops below	18% of local avg. volume
$t_{rvolmin,note}$	Local avg. volume covers	253 ms
$t_{min,note}$	Ignore notes shorter than	46 ms
$r_{f,note}$	Different notes threshold quotient is	1.03
$D_{ins(R)}$	Penalty for inserting/deleting an R is	1
$D_{ins(DU)}$	Penalty for inserting/del. a D or U is	2
$D_{R\leftrightarrow DU}, D_{D\leftrightarrow U}$	Penalty for confusing any of R,D,U is	∞

Table 1. User-selectable parameters and their settings used in the empirical evaluation. The last three lines will be explained in Section 2.5.

It is then necessary to group these pairs into notes. As an indicator for note boundaries we use (1) the silence windows and (2) sudden changes in frequency. If the quotient of consecutive frequency values exceeds a user-defined threshold $r_{f,note}$, we insert an artificial silence window to mark the end of the note. Afterwards, we combine all windows between any two silence windows into a single note with the corresponding duration and remove the silence windows. The result is a sequence of notes with certain frequencies and durations.

The steps in this process, starting with the frequency-volume pairs, are:

—Set all windows where the volume, the frequency, or the peak amplitude ratio are below the user-defined thresholds $rvol_{min}$, f_{min}, or r_{peaks}, respectively, to "silence". This avoids erroneous interpretation of unclear input signals.

—If the volume drops below a user-specified threshold $rvol_{min,note}$ in relation to the average volume of the surrounding $t_{rvolmin,note}$ milliseconds, it is assumed that the whistler repeated a note without a proper break. Insert an artificial silence window in all such positions.

—If the predecessor and successor of a window have the same frequency, but differ from the current window's frequency, set the current window's frequency to that of its neighbors. This smoothes the frequency curve.

—Combine consecutive non-silence windows into one note if the frequency ratio of all these windows is below $r_{f,note}$. Calculate the average of the frequency and the sum of the durations of the windows.

—Delete silence windows and delete notes that are shorter than the user-specified minimum length $t_{min,note}$.

The resulting list of notes is later displayed together with the search result, aiding the user in finding technical parameter changes needed to enhance the recognition.

2.4 Parameter values for Parsons code conversion

As described earlier, the algorithm for the conversion is constructed in a relatively straightforward way. However, to complete the contribution, we must explain how we obtained the values for the parameters that are listed in Table 1.

For three of these parameters, namely $rvol_{min}$, $rvol_{min,note}$, and f_{min}, we performed a thorough search of the parameter space, together with the four parameters of the distance function discussed in Section 2.5. For this search we used the data recorded for the empirical evaluation as described in Section 3.1. We used half of the data for finding good parameters and the other half for validating that these values did not overfit these particular recordings (and would hence not be good values for other recordings). We found that Tuneserver was generally not prone to parameter overfitting, which is not surprising given that we fit only seven parameters to more than 1600 notes.

The 7-dimensional search worked by multiple discrete descent. From each of a large number of random starting points in the parameter space, the recognition performance at that point was compared to each of two neighboring points along each of the seven axes. The best of these was taken as the next reference point, until a local minimum was reached. Then a new random starting point was chosen. As search step size, we used $\pm5\%$ for $rvol_{min}$, $\pm3\%$ for $rvol_{min,note}$, and multiplication/division by 1.1 for f_{min}. The performance was measured by determining the rank of the intended tune in Tuneserver's result list for every recording. For tunes which did not appear among the first 100 results, a rank of 200 was counted.

We obtained performance data for about 5000 parameter sets, using several weeks of CPU time. This performance data was then analyzed to find, from the most successful parameter sets, those parameters that exhibited optimum performance robustness, i.e., best performance stability in the parameter neighborhood.

We found that the global silence threshold $rvol_{min}$ should be 5 percent, much lower than what we had initially thought. The optimum note-local volume threshold $rvol_{min,note}$ turned out to be 18 percent, also lower than expected. We found that the lower cutoff frequency f_{min} has only little influence on performance, good performance was obtained in the range 300 Hz to 800 Hz.

Early trial-and-error investigations with a few recordings had shown that the other parameters mentioned in the top part of Table 1 are even less sensitive to changes than f_{min}. Therefore, we chose them based on a few trials alone. For $r_{f,note}$, the obvious choice is a quarter tone, which is half of a twelvth octave or a frequency ratio of $\sqrt[2\cdot12]{2}$ or 1.03. We found that this works as well as expected. The times used for initial cutoff (t_{skip}), minimum note duration ($t_{min,note}$), and determining local volume ($t_{rvolmin,note}$) are not very critical. 10ms, 40ms, and 110ms, respectively, work well. Note that for whistlers producing highly fluctuating tones, $t_{min,note}$ and $t_{rvolmin,note}$ could in principle be quite important, but it turns out that such tones are usually so unstable that even the best parameter values do not help.

2.5 Search process

The server's task is to create a list of pieces whose Parsons codes are most similar to the code that was provided by the applet. To accomplish this, we used the concept of editing distance [Ukkonen 1985]. In essence, we calculate the number of insertions, deletions, and replacements of single characters that are necessary for converting one given code string into the other one. The minimum weighted sum of these elementary steps is the editing distance. It is computed by dynamic programming with fixed weights.

The user usually does not provide the same number of notes that are encoded for the tune in the database. Since we do not want this difference to influence our distance measurements, we have to avoid adding or deleting notes that are missing at the end of one of the two strings and counting these steps for the editing distance. To accomplish this, we cut off additional characters, comparing two strings of equal length.

For our problem of comparing Parsons codes, we should distinguish between different classes of replace, delete, and insert operations. Unlikely distortions of the Parsons code should lead to bigger differences. Here are the considerations:

—It is probably less likely that a user omits or adds a note that goes up or down (leading to a D or U), compared to a repetition (leading to an R). Our system also detects boundaries between different notes more reliably than boundaries between repeated notes. Hence inserting an R should have a weight different from (presumably smaller than) the weight associated with inserting a D or U.

—For replacement, the distance for R-to-D or R-to-U should obviously be smaller than the distance for D-to-U or U-to-D.

—Replacing R with U should have the same weight as replacing R with D, because otherwise the distance function would be sensitive to melodic inversion, which seems inappropriate.

—Insertions and the corresponding deletions must have the same weight, because a distance function must always be symmetric.

Applying these rules, we arrive at the following set of four different weights for the various editing operations:

—$D_{D \leftrightarrow U}$ for replacing a D with a U, or vice versa,

—$D_{ins(DU)}$ for inserting or deleting a D or a U,

—$D_{R \leftrightarrow DU}$ for replacing an R with a D or a U, or vice versa,

—$D_{ins(R)}$ for inserting or deleting an R.

The server performs an exhaustive search by comparing the query string separately to each database entry. The closest matches are returned, sorted by their editing distance. Note that only the ratio of the editing weights is important for the ordering of results, not their absolute values, so there are effectively only three degrees of freedom in the four parameters. An exhaustive search is hardly a performance problem for the 10,000 entries that we have — it can be done in about a second on our 167 MHz UltraSparc1. (The initial Java implementation of the server using JDK 1.1.5, however, took about 30 times as long.)

2.6 Parameter values for distance function

When we designed the distance function, we originally believed that the weights should be chosen so that

$$D_{ins(R)} < D_{R \leftrightarrow DU} < D_{ins(DU)} < D_{D \leftrightarrow U} < 2D_{ins(R)}$$

However, during the parameter optimization as described in Section 2.4, we found that this was a bad choice. For all four parameters, we investigated random starting points in the range 1...40, with search steps of size 1. The results indicated that direct replacements in the Parsons code should be forbidden by the distance function, that is, the weights $D_{R \leftrightarrow DU}$ and $D_{D \leftrightarrow U}$ could both be infinitely large! Note that this is equivalent to setting $D_{R \leftrightarrow DU}$ to $D_{ins(R)} + D_{ins(DU)}$ and setting $D_{D \leftrightarrow U}$ to $2D_{ins(DU)}$, because a replacement can always be performed by a deletion plus an insertion.

For the remaining parameters, our optimization suggests $D_{ins(DU)} = 2D_{ins(R)}$ to be the best choice. $D_{ins(R)}$ should be smaller because, as mentioned above, determining the correct number of Rs is often difficult for the recognition system.

3. RESULTS AND DISCUSSION

We now describe an empirical evaluation of Tunserver, based on 110 recordings that were made under adverse real-world conditions. We begin with a description of the design and participants of the study and give definitions of important terms. In section 3.3 through 3.6, we describe Tuneserver's recognition performance and analyze the factors contributing to recognition errors. The analyses are based on the individual recordings, on recordings grouped by tune or by person, and on the types of individual notes in the recording. Finally, we report on the speed of Tuneserver. The statistical evaluation was performed with S-Plus 3.4 on Solaris. For a summary of the results, you may want to peek into the conclusion section (Section 5). As for usability and qualitative aspects of Tuneserver performance, we describe some user reactions in Section 3.8.

3.1 Design and subjects of the empirical evaluation

The recordings stem from 24 different persons and were made in various live environments using a laptop computer. The recordings thus reflect different sorts and levels of background noise, various kinds of acoustic environment, different mindsets of the subjects, etc. The evaluation uses the parameter settings given in Table 1. Of the 24 subjects, 18 were computer scientists, and a few were musicians. The data of subject 13 was ignored in the analysis, as this person brought out only the same note over and over, and even that was hardly a whistled tone at all.

Every subject was asked to first whistle two themes of his or her own choice, then the theme from Haydn's Kaiserquartett (Emperor's quartet) which is used as the German national anthem, and finally the opening theme of Mozart's "Eine kleine Nachtmusik". Due to the circumstances there are a few exceptions to this rule, e.g., 3 persons whistled less than four themes while 2 whistled more than four. If immediately after a recording (and before the Tuneserver query results were known) a subject wanted to repeat the recording, such a repetition was performed, but still both recordings are included in the evaluation. Overall, there are 13 repetitions plus 2 second repetitions within the 106 valid recordings.

3.2 Definitions

If, in the Tuneserver output for a recording of theme t, there are n other themes with a distance smaller than t, we call $n + 1$ the *rank* of the recording. We call the recording a *hit* if its rank is 1 and *found* if its rank is 40 or better. A recording may not be found either because Tuneserver did not decode it properly or because it is not *known*, i.e., the theme is not in Tuneserver's database at all. A recording is *final* if it was not repeated by the subject. We distinguish between three different

Parsons codes for each recording. The *recognized code* is what Tuneserver actually produces from the input, the *intended code* is the code of the tune stored in the database and the *whistled code* is what the subject actually did input, as judged by a human expert. The whistled code is usually "in between" the other two and allows to separate recognition errors made by the system from recognition errors provoked by incorrect input. For each recording, we subjectively evaluated several conditions that might make recognition more difficult; for each we use a four-point ordinal scale (none, little, some, much): *Noise* is the presence of background sounds, usually wind, sometimes other voices. *Breath* characterizes the average lack of clarity of the whistled tones, which usually stems from additional air sounds. *Vibrato* refers to frequency oscillation within the individual notes, and *glissando* describes how much the tones were "smeared" into one another with respect to their frequency changes. The *error density* of a recording is the editing distance from recognized to whistled Parsons code divided by the length of the whistled Parsons code. The latter will also be called the *number of notes* or the *recording length*.

3.3 Results by recording

Twelve recordings were not *known*, we ignore them for the rest of this subsection. The overall *found* rate is 77%, the overall *hit* rate is 44%. If Tuneserver's note recognition were perfect, that is, if the database could be queried with the Parsons code of what was actually whistled (without any recognition errors), the *found* rate would be 95% and the *hit* rate 94%. The remaining 6% mistakes are due to melodies that were whistled so wrongly by the subjects that even their accurate Parsons code became incorrect. We can correct for these mistakes that are not recognition errors by dividing the actual *found* rate of 77% by the optimal *found* rate of 95% and obtain a corrected *found* rate of 81%. The correspondingly corrected *hit* rate is 47%. Considering only final recordings, these numbers improve to 84% and 49%, respectively.

What causes the remaining misses? We will analyze misses by several factors: first by the conditions *noise, breath, vibrato,* and *glissando*, and then by the number of notes whistled.

Breath is a significant predictor for differences in the error density; a statistical test (one-sided Wilcoxon rank sum test with Lehmann correction for ties) showed that the median error density was significantly larger when there was some or much *breath* compared to recordings with none or little *breath* ($p = 0.0008$)[2]. The problem areas *glissando* ($p = 0.597$), *vibrato* ($p = 0.773$), and *noise* ($p = 0.485$) were found to be insignificant. Recordings with vibrato actually tend to have less errors, probably because vibrato was typically used by the more capable whistlers. Finding noise to be insignificant means that our recognition is almost immune to the sometimes substantial amounts of background noise that were present during our recordings — at least with the relatively insensitive type of microphone we used.

If we restrict the set of recordings to the 62 known final recordings with *breath*

[2]This p value means the following: It seems as if one can use the *breath* level for improving predictions of error density. However, this apparent connection might be purely incidental. But the chances that it actually *is* purely incidental are only 0.0008 (that is, about 1 in 1200).

level "none", the corrected *found* ratio improves to 86%, the corrected *hit* ratio to 59%. See also Figure 2 for an overview of these results.

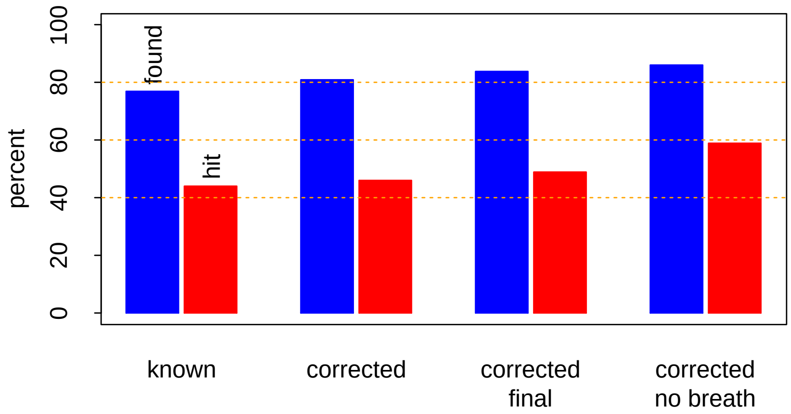

Fig. 2. *Found* rates and *hit* rates for various subsets of the known recordings from the empirical evaluation, see the discussion in the text.

The recordings consisted of between 3 and 39 notes, with a median of 16. The *found* rate increases for longer recordings, e.g., from about 70% to about 80% when the number of whistled notes increases from 8 to 16. This is not surprising because the number of exact matches in the data base can be quite large for short Parsons codes, but we allow at most 100 matches. Note that input far beyond 15 notes is not useful because no database entry covers more than 15 notes.

Summarizing, we can state that Tuneserver successfully recognizes about four out of any five tunes. The analysis of entire recordings reveals two conditions under which Tuneserver tends to make mistakes: too much breathing noise in the input, or an insufficient number of notes supplied for proper discrimination of multiple tunes.

3.4 Results by person

Each subject made between 2 and 7 final recordings of known tunes, on average 3.5. During the process of making these recordings, we found that some persons were highly capable of producing nice-sounding, accurate input that was presumably easy to process for Tuneserver, while others were bad whistlers who we expected to produce lots of failures. Looking at the results, there are indeed large differences between the success rates of the individual subjects. The most successful quarter of our subjects has a *found* rate of 100% and a *hit* rate of 67% or better, while the least successful quarter has a *found* rate at or below 67% and a *hit* rate at or below 25%. See also Figure 3.

However, although there definitely are good and bad whistlers, this distribution of *found* and *hit* rates is *not* significantly different from the distribution one would get if the rates were completely independent of the person. Hence, the available

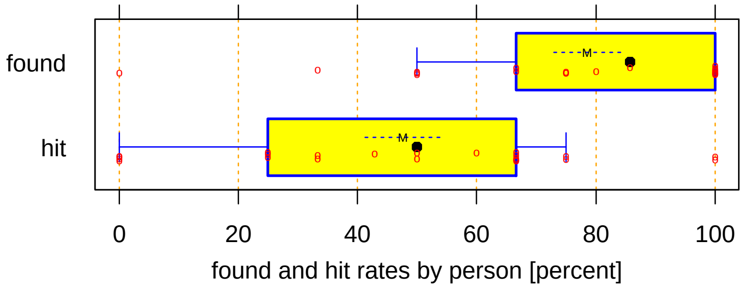

Fig. 3. Distributions of *found* and *hit* rates over the subjects. The whiskers indicate the top and bottom 10 percent, the box edges indicate the top and bottom 25 percent, the fat dot is the median, the M-line marks the mean and its standard error.

data is insufficient for measuring how much the individual whistling capabilities influence Tuneserver's recognition performance.

3.5 Results by tune

As one might expect, not all tunes are created equal with respect to their probability of being successfully recognized by Tuneserver. Plotting the *found* rates and *hit* rates separately for the different tunes, we find clear differences; see Figure 4. The

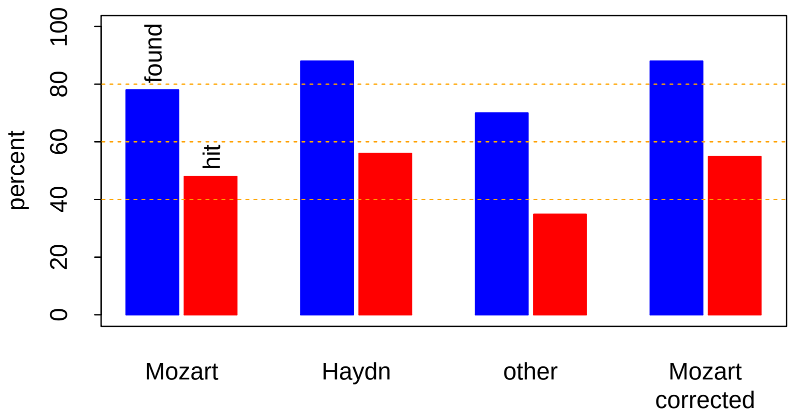

Fig. 4. *Found* rates and *hit* rates by tune whistled.

found and *hit* rates for the Mozart theme are close to the average; the success for the Haydn theme is clearly higher (two combined Fisher exact p tests indicate $p = 0.028$ that the corrected found and hit rates are in fact not different). This difference is a result of the musical difficulty of the themes: The Haydn theme has all long notes, with modest intervals in between. In contrast, the Mozart theme also contains shorter notes and larger intervals; it also covers a greater frequency range. We cannot say anything specific about any of the other themes here, since none of them was chosen often enough (only one was chosen more than twice), but apparently the whistlers often picked themes that they did not know well enough or that were difficult. As a result, the success rates for the arbitrarily chosen themes are distinctly lower than the rest. However, the Mozart theme was much more often whistled wrongly than the others: even with perfect recognition, the *found* rate would have been only 89%. As a consequence, for the Mozart recordings, the difference between corrected *found* and *hit* rate and the uncorrected rates are above average.

3.6 Results by note

One other analysis that helps understand Tuneserver's performance is based on the different types of notes (up, down, or repeat). Figure 5 shows that the performance decreases with an increasing number of R notes that need to be recognized. This indicates that R notes actually cause more difficulties than U and D notes, because, as mentioned above in Section 3.3, the performance is generally expected to be *better* for longer recordings, not worse.

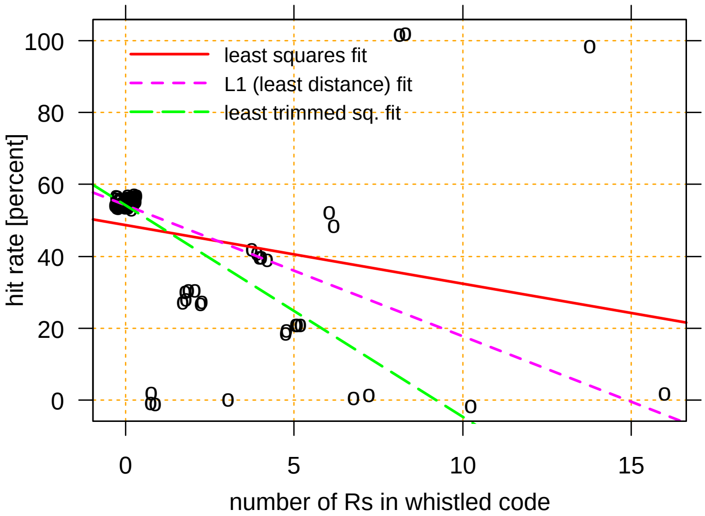

Fig. 5. Average *hit* rate depending on the number of Rs in the whistled Parsons code. The plot also shows a standard regression line and two kinds of robust regression lines.

We may understand the effect better by looking at the individual recognition errors. Figure 6 compares the number of R characters in the recognized code and the whistled code. The plot shows the differences of these numbers, separately for the Mozart and the Haydn theme; it also contains the same comparisons for the D and U characters. Ideally, the difference should always be zero; anything else indicates recognition errors.

We see that R is more prone to errors than D or U and that the R errors are always insertions. One could reduce the number of R insertions by decreasing the $rvol_{min,note}$ or r_{peaks} parameters or by increasing the $t_{min,note}$ parameter, but both would produce more mistakes elsewhere than could be avoided here. The difficulty of correctly recognizing R notes is inherent in the task and Tuneserver actually performs quite well on it.

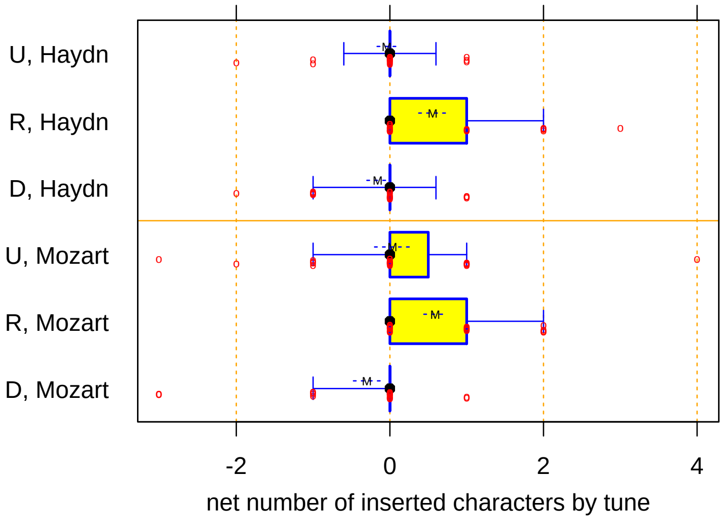

Fig. 6. Number of U characters in recognized code minus number of U characters in whistled code for Haydn recordings. Ditto for R character and for D character; ditto for Mozart recordings.

3.7 Recognition speed

Most of the time in the retrieval process is needed for the preprocessing stage. The FFT takes about 0.5 s per second of recorded sound (for a sampling frequency of 22,050 Hz) on our 300 MHz AMD K6 Linux machine when the applet is running under Netscape 4.05. The slow speed is mostly due to the not yet very efficient Java implementation in the browser. The time required for the rest of the conversion into the Parsons code is negligible.

During the stage involving our 167 MHz UltraSparc server, a Parsons code of length 20 will be searched in approximately one second.

One can say that although the retrieval takes some time, the speed is satisfactory. Actually, most of the time for a query will be spent in the manual steps that are required because the Java implementation of our current browswer Netscape Communicator 4.05 does not yet support sound recording. Hence, one has to record manually, store the recording into a file, and tell the applet the file name.

3.8 User reactions

From a purely quantitative point of view (retrieval reliability), Tuneserver certainly leaves room for improvement. However, the most important usability problem of Tuneserver has little to do with its objective performance. The biggest obstacle to successful Tuneserver use is embarrassment. Many potential users are reluctant to use Tuneserver — in particular when other people are present — because they feel ashamed of their whistling capabilities. We feel that this problem would be much reduced if Tuneserver would accept humming instead of only whistling, because most of our users find humming a lot easier and are more competent for it. Note that the severity of this problem is highly dependent on culture. We assume that it is, for example, less severe for American users compared to our German ones. It is probably impossible, though, to solve the problem entirely.

This issue set aside, most of our users reacted very positively. We quote one e-mail sent to us in verbatim. It combines several of the more frequent types of comments:

> Brilliant applet! It got all the tunes I tried except for the famous Mozart Horn Concerto KV495 rondo theme, which didn't seem to be in the database! I was particularly impressed when it got the national Australian anthem, even though I remembered it wrong!

4. RELATED WORK

We know of four published attempts at tune recognition. The first one is *Query by Humming* (QbH) by Ghias, Chamberlin and Smith [Ghias et al. 1995]. QbH is similar to the core of TuneServer (without the web service) and also uses Parsons coding. The major technical difference is that QbH uses autocorrelation instead of spectrum analysis for determining the pitch, allowing humming to be input instead of whistling. Unfortunately, the article provides only very vague information on the actual retrieval performance obtained, allowing no direct comparison of the system capabilities. The QbH database contains 183 songs; the data were extracted from MIDI files. The search implementation does not scale well as it is more than 30 times slower compared to ours: it searches about 30 songs per second on a Sun SparcStation 2.

The QbH article mentions similar work by Kageyama and Takashima that was apparently published only in Japanese.

Query by Humming also inspired a module of the same name in the *WorldBeat* system [Borchers 1997]. In that implementation, the pitch tracking is done by special-purpose hardware, a Roland GI-10 Guitar-to-MIDI converter. Again, no serious empirical evaluation of the performance is available. The database consists of 20 Beatles songs.[3]

Finally, the New Zealand Digital Library project [Witten et al. 1998] contains a tune recognition module called MELDEX [McNab et al. 1997]. It has an extensive database of 9,400 tunes (mostly folk songs, including 2,100 chinese songs), but a rather simple front-end, requiring the individual notes to be clearly separated by the user. Pitch determination is based on filtering for obtaining the base frequency and is fast in MELDEX, but the approximate code matching for database recall is almost 10 times slower than in Tuneserver. The article claims that input can be "sung, hummed, or otherwise entered", but no data on actual recognition performance is presented at all. However, the article presents a nice investigation comparing different encoding methods that either use or ignore rhythmic information and either use note intervals or the ternary R-D-U code only. The results indicate, for instance, that for the given database of 9400 tunes, an average of about 8 notes are required to reduce the number of matchings songs to just one if only the Parsons code is used,[4] about 4 notes if full melodic information (rhythm and intervals) is processed, assuming perfect quality of the user's input.

---

[3] Jan Borchers, personal communication, June 1998.

[4] The value is surprising, as 8-note Parsons codes can discriminate only 6,561 different tunes, but is plausible as an average.

5. CONCLUSIONS AND FUTURE WORK

Melody input is viable. We presented a design and implementation of a tune recognition system that is fast enough for interactive use and provided parameters with which its recognition accuracy is good enough for practical application. Our empirical evaluation found that Tuneserver lists the intended tune, if it is known, among the 40 most probable ones 77% of the time — despite difficult real-world conditions such as background noise, subjects who cannot whistle well, and subjects providing too few notes for proper recognition or providing entirely incorrect melodies. We consider this performance quite satisfying. The success rate will be even higher if the input is whistled cleanly and if it is sufficiently long (about 15 notes).

We see the following desirable extensions of this work:

—More powerful front end: A more elaborate analysis of the audio signal should allow for singing or humming in addition to whistling. This will make it subjectively and objectively easier for most users to provide input that is accurate and clear enough for an error-free analysis.

—Learning from frequent mistakes: The search machinery should learn from user feedback and automatically build and employ a model of typical user input errors. This could improve the precision and recall of the retrieval for incorrect inputs.

—Learning new melodies: It should be possible to incorporate new melodies into the database automatically. This could best be done by analysis of full musical audio signals. Alternatively, techniques such as those used (but not published) by [Ghias et al. 1995] could be used to extract melodies from MIDI files.

ACKNOWLEDGMENTS

We thank those who participated in the empirical study and all other Tuneserver users who gave us feedback. Thanks also to Michael Philippsen, Agatha Walczak, Dörte Neundorf, and Bernhard Haumacher for commenting on drafts of this article. Thanks to our reviewers for suggesting to put in some more qualitative and speculative information.

REFERENCES

Barlow, H. and Morgenstern, S. 1948. *A directory of musical themes.* Crown Publishers, New York, NY.

Benoit, C., Pelachaud, C., and Suhm, B. 1999. Multimodal speech systems. In D. Gibbons, R. Moore, and R. Winski Eds., *Handbook of Standards and Resources in Spoken Language Systems*, Volume supplement. Berlin: Mouton de Gruyter. to appear, http://werner.ira.uka.de/˜bsuhm/.

Borchers, J. 1997. Worldbeat: Designing a baton-based interface for an interactive music exhibit. In *Proc. ACM Intl. Conf. on Computer-Human Interaction (CHI97)* (Atlanta, GA, March 1997), pp. 131–138. acm press.

Cole, R. A., Mariani, J., Uszkoreit, H., Zaenen, A., and Zue, V. 1995. *Survey of the State of the Art in Human Language Technology.* Center for Spoken Language Understanding CSLU, Carnegie Mellon University, Pittsburgh, PA.

Dellaert, F., Polzin, T., and Waibel, A. 1996. Recognizing emotion in speech. In *Proc. Intl. Conf. on Spoken Language Processing* (Philadelphia, Oct. 1996), pp. 1970–1973.

Ghias, A., Logan, J., Chamberlin, D., and Smith, B. C. 1995. Query by humming – musical information retrieval in an audio database. In *ACM Multimedia* (San Francisco, CA, Nov. 1995).

JACOB, R. J. 1996. Human-computer interaction input devices. *ACM Computing Surveys 28*, 1 (March), 177–179.

MANKE, S., FINKE, M., AND WAIBEL, A. 1995. Npen++: A writer independent, large vocabulary on-line cursive handwriting recognition system. In *Proc. Intl. Conf. on Document Analysis and Recognition* (Montreal, Canada, Aug. 1995). IEEE CS press.

MCNAB, R. J., SMITH, L. A., BAINBRIDGE, D., AND WITTEN, I. H. 1997. The New Zealand digital library MELody inDEX. *D-Lib magazine 3*, 5 (May). www.dlib.org, ISSN 1082-9873.

MYERS, B., HOLLAN, J., CRUZ, I., BRYSON, S., BULTERMAN, D., CATARCI, T., CITRIN, W., GLINERT, E., GRUDIN, J., AND IOANNIDIS, Y. 1996. Strategic directions in human-computer interaction. *ACM Computing Surveys 28*, 4 (Dec.), 794–809.

PARSONS, D. 1975. *The Directory of Tunes and Musical Themes*. Spencer Brown.

REDDY, R. 1995. Grand challenges in AI. *ACM Computing Surveys 27*, 3 (Sept.), 301–303.

STIEFELHAGEN, R., YANG, J., AND WAIBEL, A. 1997. A model-based gaze tracking system. *Intl. Journal of Artificial Intelligence Tools 6*, 2, 193–209.

STORK, D. G. AND HENNECKE, M. E. 1996. Speechreading: An overview of image processing, feature extraction, sensory integration and pattern recognition techniques. In *Intl. Conf. on Automatic Face and Gesture Recognition* (Killington, VT, Oct. 1996), pp. xvi–xxvi. IEEE CS press.

UKKONEN, E. 1985. Finding approximate patterns in strings. *Journal of Algorithms 6*, 132–137.

WITTEN, I. H., NEVILL-MANNING, C., MCNAB, R., AND CUNNINGHAM, S. J. 1998. A public library based on full-text retrieval. *Communications of the ACM 41*, 4 (April), 71–75.