# Development of an automatic chunk segmentation tool for long transcribed speech recordings

**Nina Poerner**


München 2016

# Development of an automatic chunk segmentation tool for long transcribed speech recordings

**Nina Poerner**

Masterarbeit
am Institut für Phonetik und Sprachverarbeitung
der Ludwig–Maximilians–Universität
München

vorgelegt von
Nina Poerner
aus Hamburg

Matrikelnummer 11170393

München, den 07.07.2016

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 07.07.2016

Erstgutachter: Florian Schiel

# Contents

# List of Figures

# List of Tables

# Summary

The Munich Automatic Segmentation System (MAUS) [Schiel, 1999] is a forced aligner that has the additional ability to model pronunciation variation. Its runtime explodes with audio duration and transcription length, making it inefficient on long transcribed recordings. Therefore, a chunk segmentation tool was developed as a preprocessor.

The chunker builds upon a segmentation method introduced by [Moreno et al., 1998] and further developed, among others, by [Katsamanis et al., 2011] and [Bordel et al., 2012]. It uses term- or phoneme-based speech recognition to convert the input audio file into a symbolic representation, and then symbolically aligns that representation with the original transcription. The resulting alignment is screened for areas that satisfy a minimum length and maximum edit cost criterion. Within these areas, word boundaries are chosen as chunk boundaries. Any chunks that are not yet short enough at this point are recursively subjected to the same procedure. The chunker output can be directly processed by MAUS.

The chunker was evaluated on German recording-transcription pairs consisting of concatenated material from the Verbmobil corpora [Burger et al., 2000]. Evaluation metrics were the errors made when placing chunk boundaries, the duration of discovered chunks, runtime, as well as the effect that chunker preprocessing has on word boundary errors made by MAUS. Results were encouraging:

- More than 95 % of chunk boundary errors below 100 ms
- More than 95 % of material contained in chunks of duration below six minutes
- Chunker-plus-MAUS phonetic segmentation in less than real-time on recordings of duration up to three hours on three CPUs
- Positive effects on MAUS's word segmentation accuracy

Some of the tests were also run on data with corrupted transcriptions and audio files with cross-talk. It appeared that the chunker is robust in the face of single-word omissions from the transcription. In the face of speaker turn omissions, error rates were found to increase, while in the cross-talk condition, chunks became longer. Nonetheless, the effect on MAUS's segmentation accuracy was found to be mostly positive or neutral under these conditions.

# Chapter 1

# Introduction

## 1.1 Motivation

The internet and other media are teeming with transcribed speech recordings: e-books and audio books, scripted plays and films, subtitled broadcasts, transcribed interviews, to name only a few. They are often cheaper and more numerous than classical studio recordings; and for some dialects, genres or speakers, they may be the only resources available.

In order to make these recordings accessible to (many kinds of) phonetic analysis, they have to be segmented, meaning that the tokens and phones[1] of their transcriptions have to be linked to their temporal location in the audio signal [Harrington, 2010, p. 25]. One paradigm for automatic segmentation is forced alignment using the Viterbi algorithm [Viterbi, 1967], which can be characterized as the search for the optimal way of mapping the hypothesized pronunciation of the transcription, represented as a Hidden Markov Model (HMM), onto the signal.

The Munich Automatic Segmentation System (MAUS) [Schiel, 1999] developed at the Bavarian Archive for Speech Signals (BAS) of the Munich Institute of Phonetics and Speech Processing is a variation on forced alignment. It allows several alternative pronunciations instead of just one, representing them as an acyclic graph with forks and junctions. MAUS has been successfully evaluated on German data [Schiel et al., 2011], and is also available for a number of other languages such as Hungarian, Finnish, Italian and several regional varieties of English [Schiel, 2015]. At present, the BAS offers MAUS as a free web service to a growing number of academic users [Kisler et al., 2015].

Unfortunately, the runtime of MAUS grows super-linearly with input audio duration and transcription length[2], making it too slow for long transcribed recordings. This is particu-

[1] In the following, 'token' and 'phone' are used for the concrete realizations of abstract linguistic units, while the corresponding abstract units are referred to as 'term' and 'phoneme'.

[2] For simplicity, it will be assumed from now on that transcription length is proportional to audio

larly problematic for the non-scientific recording types mentioned above, as they, contrary to the output from typical phonetic experiments, do not tend to come in short one-carrier-sentence-at-a-time kind of portions.

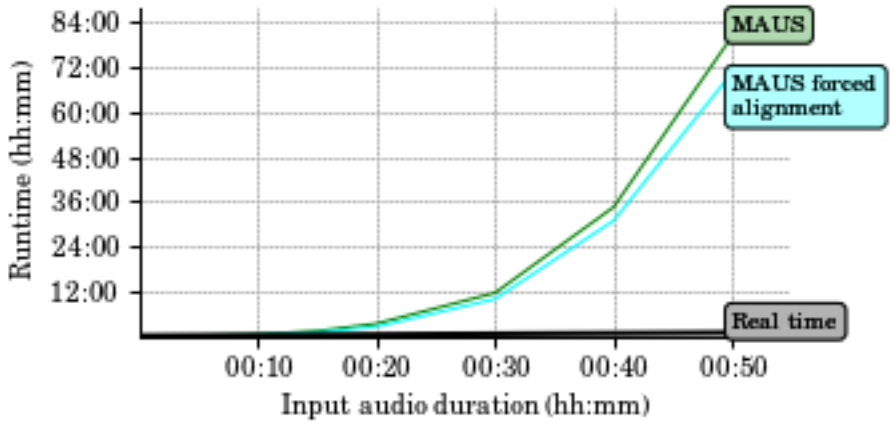



Figure 1.1: MAUS runtime in MAUS mode (pronunciation modeling mode) and forced alignment mode (see Section 4.3 for details).

### 1.1.1 Alternatives to MAUS

There are a number of segmentation tools that are less susceptible to increases in input size than forced alignment (e.g. [Moreno et al., 1998], [Katsamanis et al., 2011] and [Bordel et al., 2012], see Chapter 2 for details). However, none of them appear to support the modeling of pronunciation variation in the way that MAUS does. This, while perfectly acceptable in many scenarios, might be undesirable when performing segmentation for phonetic research. The aim of this project is therefore to make MAUS faster while preserving this ability.

### 1.1.2 Chunk segmentation

One possible strategy for coping with MAUS's runtime issues is to presegment the target material into chunks, i.e. relatively short pairs of corresponding audio and transcription subsequences that can be processed individually[3]. MAUS is able to cut the audio and transcription accordingly, perform segmentation on the individual chunks, and then recombine

duration, and that runtime can therefore be expressed as a function of audio duration alone. While this assumption is reasonable for the data used in this project (see Figure 3.4), there might be scenarios where it does not hold, and where a more nuanced analysis of runtime is necessary.

[3] In practice, manually segmented chunks may often correspond to semantically or pragmatically meaningful units such as sentences or speaker turns, but this is not a formal requirement.

the results. Due to the above-mentioned super-linear runtime complexity of MAUS, the sequential segmentation of all chunks combined takes less time than the full segmentation. What is more, the fact that the chunks are independent of one another opens up the opportunity for parallelized computation, although this is not done in the present implementation of MAUS.

Despite its usefulness for runtime reduction, the BAS does not currently offer a tool for chunk segmentation, leaving this task to the user. This is problematic in cases where a manual chunk segmentation is not feasible, e.g. if the analyzed language is not spoken by those working on it, or if the corpus is simply too big.

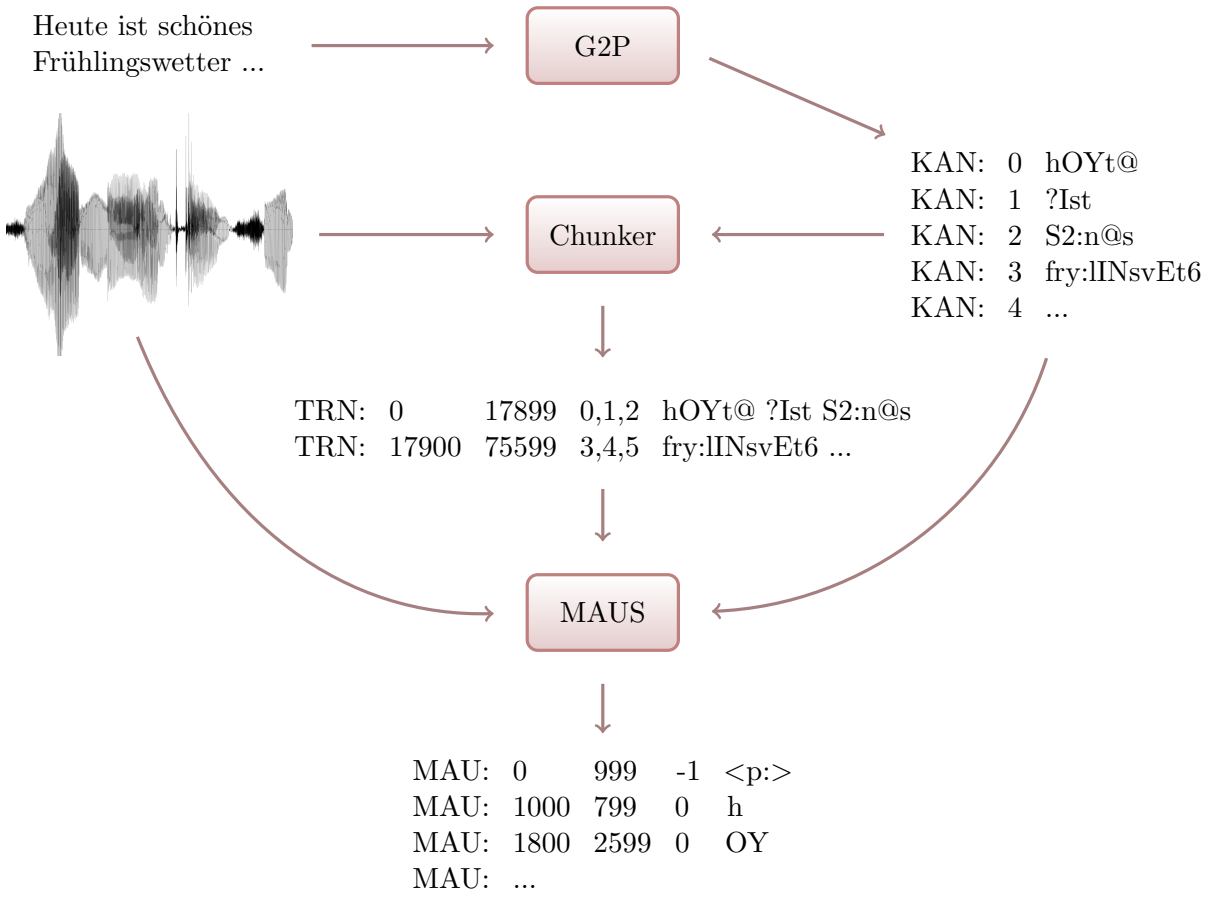



Figure 1.2: Chunker in- and output examples in context

## 1.2 Closing the tool chain gap

The goal of this project is the development of an automatic chunk segmentation tool that would be useful in these kinds of scenarios. This tool, while building on the fast methods mentioned in Section 1.1.1, should output a chunk segmentation instead of a full phonetic segmentation. This way, it can be used in conjunction with MAUS to produce a phonetic segmentation that, while more quickly available than a MAUS-only segmentation, still profits from MAUS's ability to model pronunciation variation.

Since the chunker is meant to be used in a tool chain with other tools developed at the BAS, their input and output formats should be compatible (see Figure 1.2). Therefore, the input consists of a WAVE signal file (which is also accepted by MAUS) and a transcription in form of a BAS Partitur Format file canonical pronunciation (KAN) tier [Schiel et al., 1998]. For supported languages, this KAN tier can be derived from an orthographic transcription using the G2P tool [Reichel and Kisler, 2014], which is also offered as a web service by the BAS.

The chunker should output a TRN tier, which is the chunk segmentation format expected by MAUS. This TRN tier can then be passed to MAUS along with the KAN tier and the original signal file, to produce the actual phonetic segmentation.

## 1.3 Evaluation

After development, the chunker was fine-tuned and evaluated on German data from the Verbmobil corpora [Burger et al., 2000]. It can be extended to other languages for which there are compatible acoustic models (see Section 3.3.1). A number of benchmarks were defined prior to evaluation:

### 1.3.1 Benchmarks

#### Chunk boundary errors

As errors in the placement of chunk boundaries are likely to have a negative impact on MAUS, chunk boundary errors should be small. Since it was found that MAUS manages to place 95 % of word onsets within 110 ms of their ground truth location on clean data from the evaluation corpus (see Figure 4.5), this threshold was chosen as a benchmark.

**Benchmark I**: At least 95 % of chunk boundaries are no more than 110 ms from their location according to some ground truth segmentation (meaning no more than 110 ms to the left of the end of the previous chunk's last token, or to the right of the beginning of the next chunk's first token, whichever is closer).

### Token accessibility

The chunker should be successful at making material accessible to MAUS. Since material is accessible when it is contained in short chunks, this means that the chunker should return short chunks. In the data underlying Figure 1.1, the five minute recording was MAUS-segmented in less than real-time, while all recordings longer than that took more than real-time. Therefore, this threshold was chosen as a benchmark.

**Benchmark II**: At least 95 % of tokens end up in chunks of duration 5 minutes or less.

### Effects on MAUS

Of course, the chunker is of limited use if it ends up deteriorating the segmentation accuracy of MAUS. Therefore, the third benchmark was defined as follows:

**Benchmark III**: The chunker should not deteriorate MAUS's word segmentation accuracy to a statistically significant degree (as measured by a paired t-test, see Section 4.2 for details).

### Performance

The chunker should speed up phonetic segmentation with MAUS, such as to make even long transcribed audio files segmentable in a relatively short period of time.

**Benchmark IV**: The combined runtime of the chunker and a subsequent MAUS segmentation should lie below real-time on audio files of duration up to three hours.

# Chapter 2

# Previous works

At the time of writing, there do not appear to be any tools available that are purely consecrated to chunk segmentation. However, there are tools for the segmentation of long audio-transcription pairs that implicitly perform chunk segmentation as an intermediate step towards a more fine-grained segmentation. One such paradigm is the recursive application of speech recognition and symbolic alignment for segmentation, which was proposed by [Moreno et al., 1998] and further developed by, among others, [Katsamanis et al., 2011] and [Bordel et al., 2012]. As the present project builds heavily on their work, their methods and evaluation results are described in some detail in Section 2.1. Two other solutions for long audio-transcription alignment are briefly presented in Section 2.2.

## 2.1 Recognition and symbolic alignment

### 2.1.1 [Moreno et al., 1998]

#### Method

[Moreno et al., 1998] describe a recursive text-to-speech alignment algorithm that performs speech recognition on the audio file and then symbolically aligns the recognition result with the transcription. Within the resulting alignment, they look for so-called anchors, or 'islands of confidence', where the strings match for a given number of words. These anchors are considered successfully aligned, while the space between them is recursively subjected to the same, above-mentioned procedure. The intermediate result at each recursion layer can in fact be considered a chunk segmentation, where the end of one anchor and the beginning of the next are the boundaries of a non-aligned chunk (see Figure 2.1).

When performing recognition, [Moreno et al., 1998] take advantage of the fact that the transcription for a particular region of interest is a strong hint towards the content of

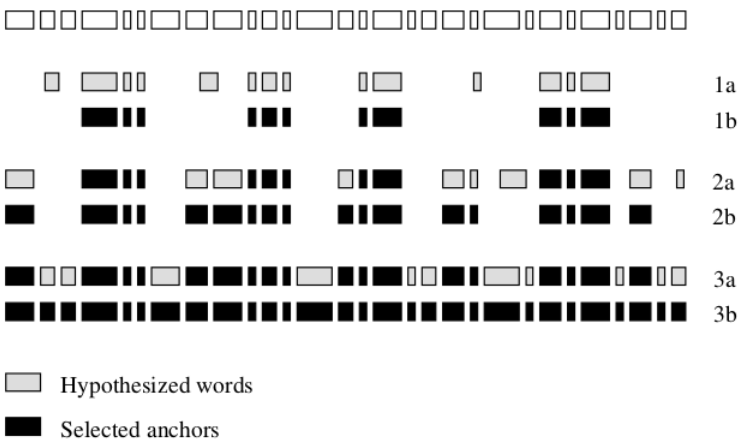



**Figure 2:** Example of the aligner run. Each set of two lines, *a* and *b*, represents one iteration of the algorithm.

Figure 2.1: Schematic representation of recursive alignment from [Moreno et al., 1998, p. 2712]

that region, and train the recognition engine's language model on it. This means that, as we progress deeper into recursion and unaligned stretches become shorter, the recognition engine becomes more and more strongly attuned to the target material. What is more, the threshold for anchor length is decreased with every iteration, meaning that low-confidence decisions are held off until late in the recursion process, when their impact is smaller.

### Evaluation results

[Moreno et al., 1998] report alignment accuracy scores on a three-hour recording from the hub-4 broadcast corpus, which contains a mixture of noisy and clean audio data from English media broadcasts. They report that 98.5 % of all words could be located within 500 ms of their ground truth location, and 99.75 % within 2 seconds. What is more, they report promising results for cases in which the data is obscured by artificial white noise (silence-to-noise ratio 15 dB) or audio compression. In the white noise scenario, 94.4 % of words are aligned within 2 seconds of their ground truth location, while in the compression scenario, this figure rises to 99 % of all words.

### 2.1.2 [Katsamanis et al., 2011]

#### Method

[Katsamanis et al., 2011] build on the above-mentioned approach. Their segmentation tool SailAlign performs segmentation in five steps: three iterations of recursive alignment as proposed by [Moreno et al., 1998], followed by one alignment pass with an automaton language model that allows for insertions or deletions of words, and finally a forced alignment on any stretches that are still not aligned. Contrary to [Moreno et al., 1998], they also adapt their acoustic models to the already segmented anchors.

#### Evaluation results

SailAlign's alignment accuracy scores, as tested on an artificially concatenated one-hour chunk from the TIMIT corpus, are comparable to those reported by [Moreno et al., 1998]. While no exact figures are cited, a graph included in the paper suggests that about 97 % of words are aligned within 300 ms of their ground truth location, and about 75 % are within 50 ms. What is more, [Katsamanis et al., 2011] report that their algorithm is robust against noise and transcription errors. When introducing babble noise at a signal-to-noise ratio of 15 dB, they could align just under 70 % of words within 50 ms of their ground truth location. At signal-to-noise ratios of 10 dB and 5 dB, these figures drop to just over 60 % and just under 50 % respectively. Similar results are reported for alignment tasks with corrupted transcription tokens, where just under 70 % of words are aligned within 50 ms when 10 % of transcription tokens are deleted, inserted or substituted.

### 2.1.3 [Bordel et al., 2012]

#### Method

[Bordel et al., 2012] further adapt [Moreno et al., 1998]'s approach by replacing the term-based speech recognition engine with a phoneme recognition engine. This means that they perform both recognition and symbolic alignment on the level of the phoneme instead of the level of the word, which improves performance but reduces segmentation accuracy, relative to the term-based approaches of [Moreno et al., 1998] and [Katsamanis et al., 2011]. Contrary to them, [Bordel et al., 2012] also do not appear to use their algorithm recursively. Instead, it seems that they simply return the first alignment, without looking for anchors.

#### Evaluation results

The evaluation results obtained by [Bordel et al., 2012] are somewhat weaker than those reported for the term-based approaches, with under 96 % of words within 500 ms of their ground truth location on the hub-4 corpus. What is more, the approach is reported to be more susceptible to noise, with high error rates (6% of words off target by more than 2 seconds) in the presence of background music.

### 2.1.4 Other implementations

There are other researchers who have implemented and adapted [Moreno et al., 1998]'s algorithm, often with very different applications and evaluation metrics. One particularly prominent application domain is broadcast caption alignment. Examples of such tools include [Martone et al., 2003], [Huang et al., 2003] and [Knight and Almeroth, 2012], who use term-based speech recognition and symbolic alignment to derive timestamps for closed captions. Contrary to [Moreno et al., 1998] and [Katsamanis et al., 2011], they do not appear to apply their algorithms recursively.

A very recent example of a caption alignment algorithm building on [Moreno et al., 1998]'s work is a system developed by [Lanchantin et al., 2015]. They perform a single iteration of term-based speech recognition, symbolic alignment and anchor discovery to derive a preliminary chunk segmentation, before following up with forced alignment. This workflow is similar to the one intended for the chunker and MAUS, apart from the fact that the chunk segmentation step is done without recursion in [Lanchantin et al., 2015] and with recursion in the chunker (see Section 3.8).

Finally, there is a related tool implemented by [Jang and Hauptmann, 1999] for the purpose of acoustic model training. In their case, the anchors or 'islands of confidence' discovered after symbolic alignment are not treated as an intermediate step towards a segmentation of the whole recording. Instead, material within anchors is used to train or improve acoustic models, while all material outside anchors is simply ignored.

## 2.2 Other approaches to long recording segmentation

### 2.2.1 [Anguera et al., 2011]

While [Moreno et al., 1998] and their successors perform alignment in the symbolic domain, [Anguera et al., 2011] do the opposite and perform dynamic time warping (DTW) in the signal domain. More specifically, they perform silence filtering on the original signal, synthesize the transcription to create a second signal, and then warp both signals onto one another. Due to the higher granularity in the signal domain, the search space for the

DTW algorithm grows quickly with recording duration, making it necessary to break the alignment down into windows. [Anguera et al., 2011] report a mean word alignment error of 68.8 ms, with 90 % of the errors below 121.4 ms, on several five minute audio book excerpts.

### 2.2.2 [Moreno and Alberti, 2009]

[Moreno and Alberti, 2009] use factor automata for the segmentation of long transcribed recordings. Factor automata are finite state automata that accept all subsequences of a symbolic sequence. This means that a factor automaton built on the transcription 'Heute ist schönes Frühlingswetter' would accept sequences such as 'Heute', 'Heute ist', 'ist schönes Frühlingswetter', but not 'Heute schönes Frühlingswetter', as the latter is not a subsequence. When using a factor automaton as a language model, the recognition engine should therefore be able to recognize and thereby segment subsequences of the audio file with relatively high accuracy. This makes it possible to cut the long audio file into short portions, segment them separately (and potentially in parallel), and to concatenate the results. See also [Moreno and Alberti, 2012].

# Chapter 3

# The tool

## 3.1 Overview

While the chunker draws on the segmentation algorithms presented by [Moreno et al., 1998], [Katsamanis et al., 2011] and [Bordel et al., 2012], its goal is not a full segmentation into tokens or phones. Instead, it returns chunks with a minimum duration set by the user, leaving the more fine-grained segmentation to MAUS or other tools. The chunker is implemented as a C++ module (2011 standard library) and requires access to a number of HTK tools [Young et al., 2006] (see Appendix A). The chunker's workflow can be divided into seven phases, which are briefly presented below. For a more detailed description of each phase, see Sections 3.3 through 3.9.

#### Input phase

The input phase consists of a number of checks on the parameters and input files supplied by the user. The provided source audio file is converted into the format expected by the recognition engine, while the transcription is parsed into an internal representation.

#### Language model phase

In the language model phase, an n-gram language model or phonotactic model is derived from the transcription. It is stored in a unique working directory in the format expected by the recognition engine.

### Recognition phase

In the recognition phase, the files derived in the input and language model phases are inputted into HTK's Viterbi decoder HVite, along with a set of preexisting acoustic models. HVite outputs a so-called label file that represents the optimal recognition result as a sequence of tokens or phones, with associated start and end times. The label file is parsed into an internal representation, which serves as input for the following stage.

### Alignment phase

In the alignment phase, the transcription and the recognized string are symbolically aligned using naive Levenshtein edit costs [Levenshtein, 1966]. The output of this phase is an alignment path, meaning a series of symbol insertions, deletions, substitutions and zero-substitutions.

### Chunk boundary selection phase

In the chunk boundary selection phase, the alignment path is traversed to find candidate anchors, i.e. sequences of alignment operations that match a user-defined set of criteria related to length, edit cost and the global frequency of contained tokens. The candidate anchors are then screened for word boundaries that can serve as chunk boundaries in the output.

### Recursion phase

If the **-recursion** option is enabled, the chunker may spawn a child chunker for every discovered chunk of sufficient duration. Since child chunkers may spawn new children themselves, this results in the creation of a recursion tree. Any chunk boundaries discovered by a child chunker are added to the parent chunker's own set of chunk boundaries.

### Output phase

The output function, which is called by the topmost chunker in the recursion tree, is responsible for converting the collected information into a BAS Partitur Format TRN tier. The TRN tier can then be processed by MAUS.

## 3.2 Three algorithms in one

The chunker is made up of three distinct but related algorithms, which perform the same task but differ in runtime and accuracy. The algorithms do not have distinct code bases, but are implemented using a single set of modules polymorphically. For a comparison of the algorithms' runtimes and general performances, see Chapters 4 and 5. The algorithm can be chosen by setting the command line argument **-chunkertype** to **t**, **tp** or **p**.

#### The T chunker

The T chunker (token-based chunker) is based on the approaches of [Moreno et al., 1998] and [Katsamanis et al., 2011], meaning that it performs both recognition and symbolic alignment on the level of the term or token. More specifically, the recognition engine runs on a term-based language model and produces a string of tokens, which is symbolically aligned with the transcription on a token-by-token-basis.

#### The P chunker

The P chunker (phone-based chunker) is based on [Bordel et al., 2012]'s approach, meaning that both recognition and symbolic alignment are performed on the level of the phoneme or phone. More specifically, the recognition engine runs on a phoneme-based phonotactic model and produces a string of phones, which is then symbolically aligned with the transcription on a phone-by-phone-basis.

#### The TP chunker

The TP chunker (token-phone-based chunker) is a mixture of the aforementioned approaches. Here, the recognition engine runs on a term-based language model but produces a string of phones, which is then symbolically aligned with the transcription on a phone-by-phone-basis. The TP chunker was implemented in the hope that results could be improved by combining the higher quality of term-based recognition with the higher granularity of phone-by-phone alignment (see Section 3.6.3).

## 3.3 Input phase

There are two ways of passing input to the chunker: via command line arguments or via a configuration file, with the command line taking precedence.

### 3.3.1 Acoustic model input

For recognition, a set of acoustic models must be supplied in form of a HTK master model file (MMF), via the argument **-mmffile**. As the chunker does not support the training of acoustic models, this file must be preexistent. For the experiments in Chapters 4 and 5, the master model file from German MAUS was used.

### 3.3.2 Audio input

At present, the chunker only accepts WAVE audio files as its audio input. Using HTK's HCopy tool, the audio file is converted into an HTK parameter file, which is the format expected by HTK's recognition engine HVite. The parameter file contains a sequence of 39-dimensional vectors made up of 12 Mel Frequency Cepstral Coefficients, plus energy, with delta and delta-delta coefficients[1]. Note that the conversion is done only once, meaning that any subsequent operations are performed directly on the HTK parameter file. The parameter file is stored in a unique working directory created inside the directory passed via the argument **-workdir**.

### 3.3.3 Transcription input

The transcription must be supplied in form of a BAS Partitur Format file. The BPF file must at least contain a sampling rate entry and a KAN tier (transcription in canonical pronunciation). Any tiers other than the KAN tier are ignored.

At this point, the tokens in the KAN tier are not yet segmented into phonemes[2]. Furthermore, depending on the origin of the transcription, the symbols used may not fully correspond to the model names in the MMF. Therefore, a two-column table must be supplied via the argument **-labelmap**, which maps every phoneme used in the KAN tier onto the name of the acoustic model representing that phoneme. Every model name must of course correspond to an entry in the MMF. Additionally, there is the possibility of passing a one-column table of symbols that should be ignored during parsing via the argument **-exceptionfile**.

Usually, these tables and the master model files won't be supplied by the end user, but set at the BAS via the configuration file. The tables on the appended USB drive, along with the German MAUS master model file, were sufficient for the material used in the experiments in Chapters 4 and 5.

[1] This format corresponds to the format of the German MAUS master model file, which is used in the evaluation experiments. It is presently hardcoded, although this may change in the future to ensure compatibility with other types of master model files.

[2] This means that, for an entry such as /haUs/, there is no a-priori way of knowing whether it corresponds to the phoneme sequence /h/-/a/-/U/-/s/ or /h/-/aU/-/s/.

#### Implementation of the transcription class interface

Internally, the KAN tier is transformed into a transcription object. The transcription object offers the interface by which all other classes in the program access the transcription. It offers this access via a token-by-token iterator and a phone-by-phone iterator, which have a common abstract parent class and can therefore be used polymorphically. Thus, classes like the symbolic aligner and the language model do not need to know whether they are used in a T, TP or P chunker. Instead, whether they operate on the level of the word or on the level of the phoneme is fully governed by the iterator that they are given.

## 3.4 Language model phase

### 3.4.1 Language or phonotactic model

In many cases, the acoustic information in the audio signal is not sufficient for speech recognition. Many speech recognition systems therefore rely on language models for information about which sequences of words are more likely than others (e.g. [Fink, 2013, pp. 23-24]). Similarly, phoneme recognition engines may use phoneme-based models, which are also called phonotactic models, to bias the recognition engine towards phonotactically probable sequences. One such example would be MINNI [Kisler et al., 2016], a phoneme recognition web service that is also offered at the BAS, which uses a bigram phonotactic model for recognition.

When opting for a term-based language model, a problem that needs to be solved is the vast number of possible terms and term combinations. Even a language model trained on millions of sentences will frequently encounter unknown terms, or unknown n-grams, leading to errors (e.g. [Martin and Jurafsky, 2000, ch. 4]). In the present case, the task is made easier by the transcription, which can be considered a strong clue about the content of the audio file. [Katsamanis et al., 2011] and [Moreno et al., 1998] exploit this clue by training their language models on the transcription, assuming that while this makes their recognition engine weak at recognizing other texts, it gives it an important edge when it comes to recognizing the target. In taking this approach, [Katsamanis et al., 2011] opt for a trigram language model, while [Moreno et al., 1998] use word pair and triple models.

The chunker takes a similar approach and trains an n-gram language model (T and TP chunker) or n-gram phonotactic model (P chunker) on the transcription. Contrary to [Katsamanis et al., 2011], the n-gram degree is not fixed, but can be set via the argument **-languagemodel**. More specifically, the user can choose between three options, which can be combined with all three algorithms presented in Section 3.2:

- A zerogram model (**-languagemodel 0**), meaning that all terms or phonemes are equally probable regardless of context.

- A unigram model (**-languagemodel 1**), meaning that a term or phoneme's probability depends solely on its frequency in the transcription.
- A smoothed (interpolated) bigram model (**-languagemodel 2**), meaning that a term or phoneme's probability is a weighted sum of its probability given the previous token or phone (bigram probability) and its unigram probability. The weight of the bigram probability can be set via the argument **-bigramweight**, while the unigram probability receives the weight (1 − **bigramweight**).

In all three cases, the weight of the language model, relative to the acoustic models, can be set via the option **-languagemodelweight**. For the evaluation experiments, smoothed bigram models were used, with **-bigramweight** set to **0.5** and **-languagemodelweight** set to **4.0**.

#### The imperfect transcription issue

Of course, training a language model solely on the transcription can be an issue when the audio file and the transcription do not correspond to the desired degree. Consider the case of an interview where only the interviewee's turns have been transcribed. In this case, the recognition engine should perform poorly on the non-transcribed stretches, seeing that its model should not generalize well to them, and might not even know the contained terms. In the best-case scenario, the recognition output on these stretches will be a more or less garbled sequence of the best-fitting terms from the transcription. In the worst-case scenario, which is especially relevant when using a bigram model, the language model might coax the recognition engine into incorrectly 'finding' a longer sequence from the original transcription, which might then lead the symbolic aligner astray.

The interview scenario, where speaker turns from the audio file are missing from the transcription, was tested in the evaluation experiments. Results indeed suggested that the rate of misalignments increases on this kind of material (see Section 4.1 for details). One possible solution to this problem would be smoothing the transcription-attuned language model with a bigger background corpus [Lanchantin et al., 2015]. However, this option was rejected, as the inflation of the language model would have been likely to slow down recognition in the T and TP chunkers (see Section 3.5.1), thereby counteracting the goal of implementing a fast algorithm.

### 3.4.2 HTK lattice

All three model types are stored as HTK lattices with weighted or (in the case of the zerogram model) unweighted transitions. The HTK lattice format [Young et al., 2006, ch. 20] is a plain text list of the nodes and arcs of the language model. It can be read directly by the HVite recognition engine. The lattices are constructed according to the

schemata represented in Figures 3.1 through 3.3. Note that the nodes labeled 'NULL' are non-emitting nodes, meaning that they cannot be used to model the emission of signal frames, and that their duration is therefore always zero. The nodes labeled '<p:>' are special optional pause models. They are inserted between the units of the lattice to absorb possible silences between tokens in the audio signal.

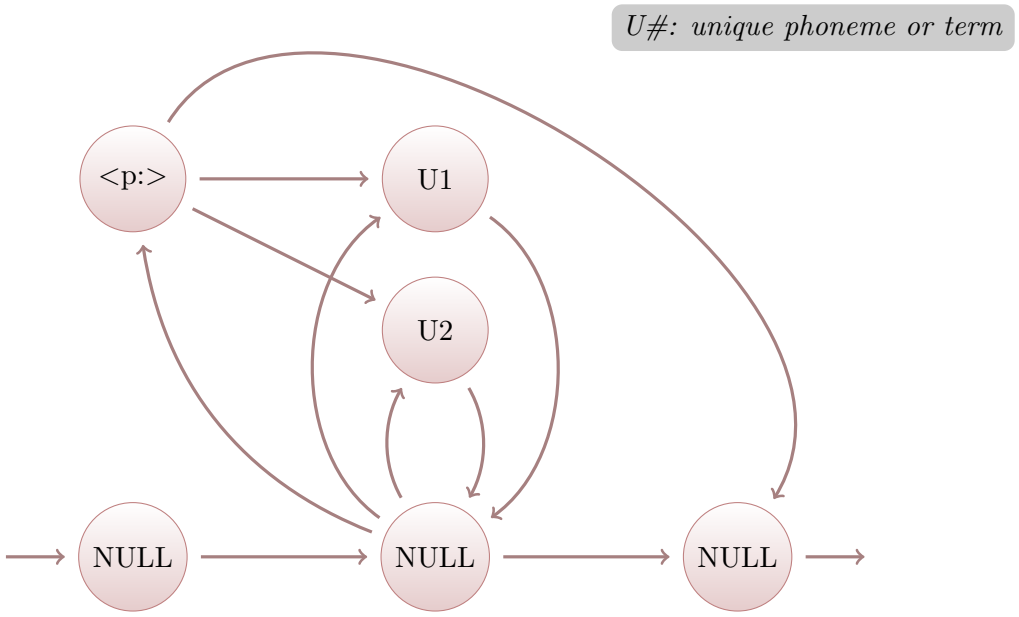



Figure 3.1: Schematic representation of the zerogram lattice

### Lattice size

Regardless of the n-gram degree, the number of nodes in the P chunker lattice is proportional to the number of unique phonemes in the transcription. Since the number of unique phonemes is bounded, this means that lattice size is effectively constant. The number of nodes in the T and TP chunker lattices on the other hand is proportional to the number of unique terms in the transcription. This has some important effects on recognition runtime, which will be explored in Section 3.5.1.

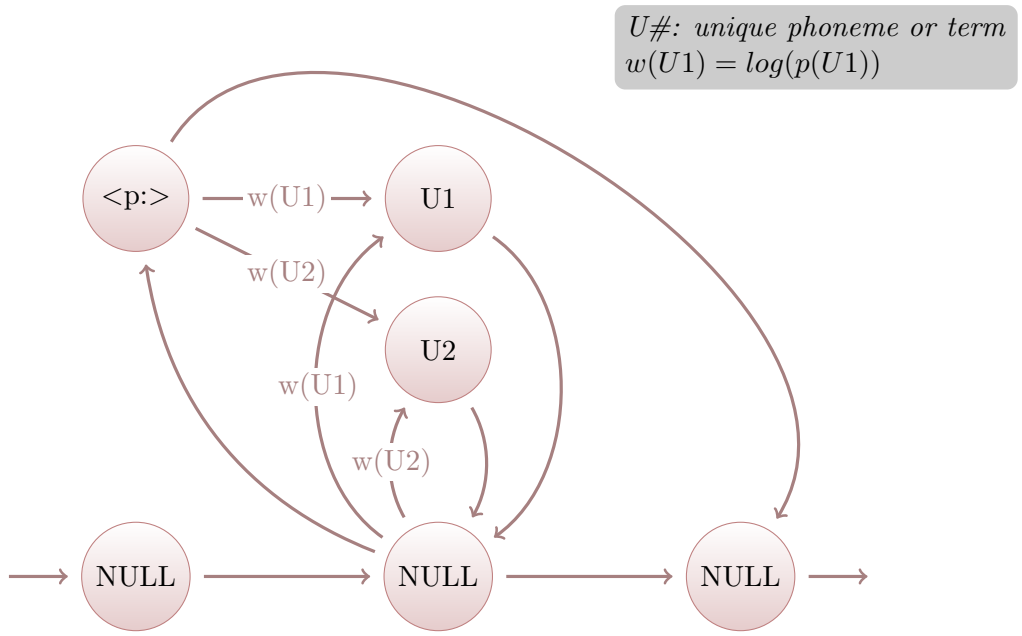



Figure 3.2: Schematic representation of the unigram lattice

### 3.4.3 Dictionary and HMM file

In addition to the lattice, HVite requires a so-called dictionary file which specifies a sequence of acoustic models for every unit (every term or phoneme) of the lattice, as well as an HMM file listing the names of all necessary acoustic models. Both files are derived from the transcription object and saved to the chunker's working directory. In both files, an additional entry is made for the optional pause symbol <p:>.

## 3.5 Recognition phase

The HTK Viterbi decoder HVite is used to derive a symbolic representation from the signal. HVite works by first turning the provided lattice, dictionary and acoustic models into a multi-level network of nodes. Once this network is compiled, the most likely path given the audio input is calculated by propagating a number of so-called 'tokens' through the network, while adding up the emission and transition log probabilities that they encounter on their way. A token residing in a node s at a point in time t represents the optimal partial path starting at time zero and ending in node s at time t. Through dynamic programming, HVite calculates the optimal path ending in the exit node at the last signal frame. This path then becomes the recognition result. For details, see [Young et al., 2006, ch. 13].

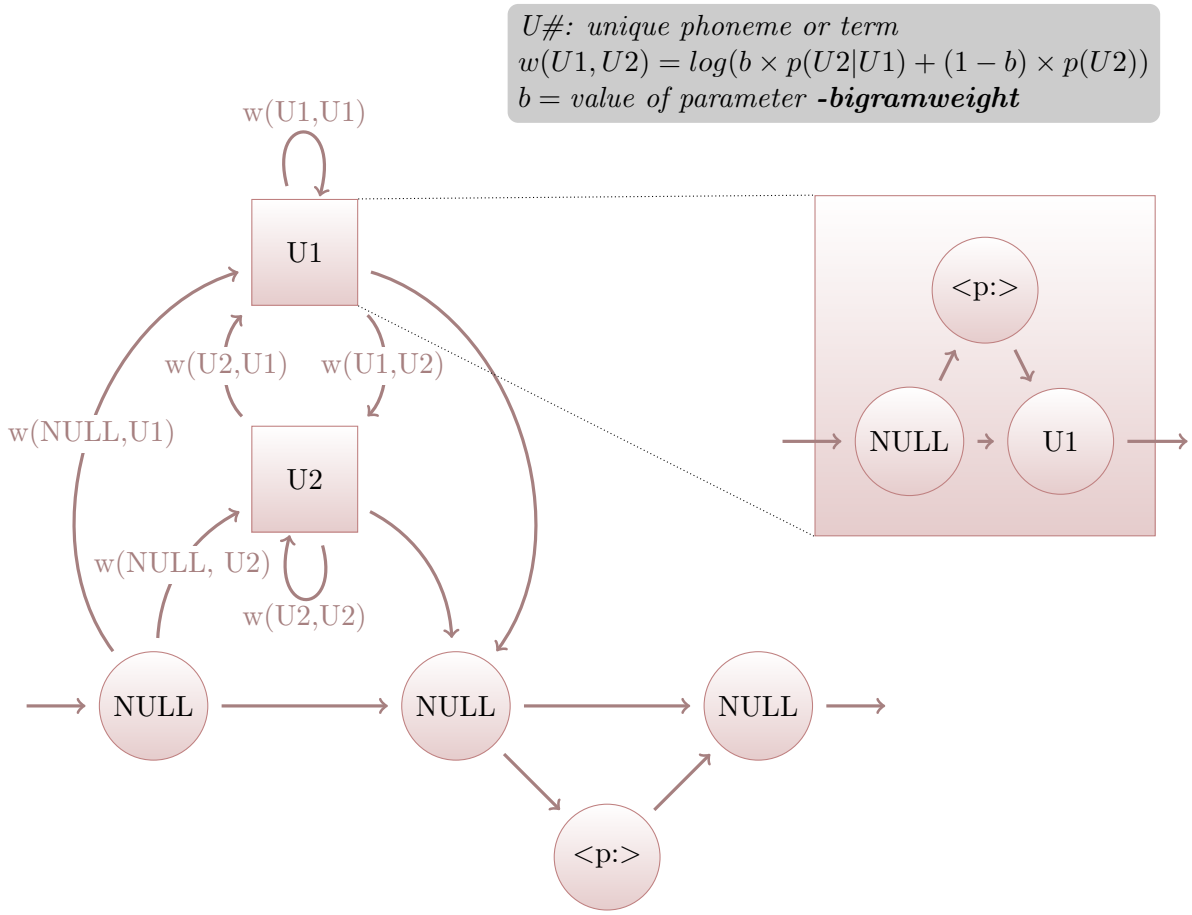



Figure 3.3: Schematic representation of the smoothed bigram lattice

### 3.5.1 HVite performance

Since there can only be one optimal path ending in a given node at a given point in time, the maximal number of tokens that need to be tracked for a single frame of the input signal is limited to the number of nodes. In other words, HVite runtime depends both on audio duration and on the size of the HTK lattice.

In the P chunker case, the number of nodes in the lattice is constant (see Section 3.4.2), meaning that HVite runtime should be linear in audio duration alone. In the T and TP chunker, the number of nodes is proportional to the number of unique terms in the transcription. If we assume that the number of transcription tokens is proportional to audio duration, then the deciding factor for HVite's performance is the ratio of unique terms to transcription tokens. To demonstrate the effect of this factor, consider the following extreme cases:

### Fast case

First, consider the case where the transcription underlying the lattice consists of the repetition of a single sentence to the point where, no matter how long the recording, the number of unique terms never increases. In this case, the number of nodes in the HTK lattice would be constant, leading to HVite's runtime being linear in audio duration.

### Slow case

Now consider the other extreme case, which might occur in a recording of someone reciting a dictionary. In this scenario, every token is a new unique term, meaning that lattice size grows linearly with transcription length. In fact, the number of nodes in this lattice would be of the same order as the number of nodes in a lattice used for forced alignment of the same transcription. Therefore, recognition would unlikely be faster than forced alignment.

The position of a given transcription on the axis between these two extremes will necessarily depend on a number of factors such as the elicitation method, the register and the topic of the recorded text, all of which are beyond the scope of this discussion. In the material used for runtime evaluation in Section 4.3, the correlation between the number of tokens and the number of terms was found to be sub-linear, with the number of terms growing much more slowly than the number of tokens, and the growth rate decreasing with transcription length (see Figure 3.4). Therefore, the number of nodes in a term-based recognition HTK lattice, like in Figures 3.1 through 3.3, should normally be inferior to the number of nodes necessary for the corresponding forced alignment task.

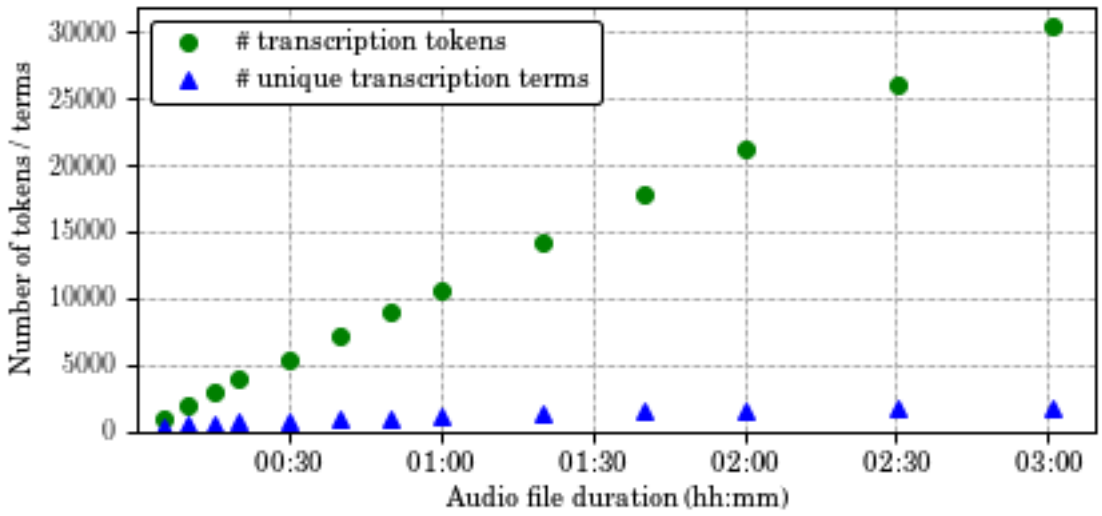



Figure 3.4: Number of tokens and unique terms in transcriptions of audio files of varying durations from Section 4.3

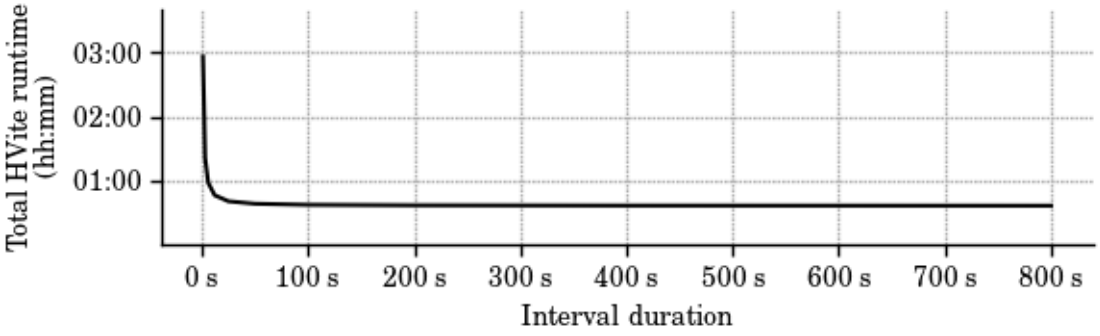



Figure 3.5: HVite runtime on spliced one-hour signal (term-based smoothed bigram lattice) at different interval durations without multithreading

### 3.5.2 Speeding up HVite

#### Parameter file splicing and parallelization

The idea to splice the HTK parameter file was inspired by [Katsamanis et al., 2011], who report having to cut their signal files into intervals of about 10 to 15 seconds due to 'computational limits of the speech recognition engine'. While this particular computational limit did not appear to affect the recognition engine used in the present project, it was found that HVite would show undefined behavior in form of a segmentation fault when performing phoneme recognition on some long audio files, starting at an audio duration of about 40 minutes. Therefore, and because of the additional benefit of multithreading as described below, it was decided to cut the signal into regular intervals before passing them to HVite one by one[3]. The duration of these intervals can be set via the parameter **-splitduration**.

It is important to note that since, at this point, the content of the intervals is unknown, every interval has to be recognized using the complete lattice. This means that while operations are distributed over several HVite calls, the total number of operations remains the same, with the additional overhead of having to compile the lattice multiple times. Tests run during development suggested that interval durations below the 100 second mark are harmful to runtime, whereas changes above that mark do not have a noticeable effect (see Figure 3.5). Therefore, an interval duration of 120 seconds was used in the evaluation experiments.

[3] While [Katsamanis et al., 2011] use voice activity detection (VAD) to ensure that the signal is not cut inside words, no such precautions are taken in the chunker. However, this does not appear to be overly harmful to the results. In an experiment performed on a single one-hour audio file, it appeared that acceptable results could be achieved with interval durations as short as one second, with a median chunk boundary error of 19 ms (range 0 - 1.5 seconds), and a maximal chunk duration of 1 minute.

While performance may not benefit from signal splicing alone, it is easy to see that splitting the recognition task into many independent subtasks opens up the possibility of a multithreaded approach. Since multithreading is only productive when the hardware is capable of handling the desired number of threads, its extent is controlled by a lock-protected internal counter; and new threads are only opened when that counter is below the value of the **-maxnumthreads** parameter. See Section 4.3 for an overview of the impact of parallelization on the different chunker types at one, three, six and nine threads.

#### Beam search

Apart from parallelization, there is also the option of speeding up HVite using beam search. Beam search is a variation on the traditional Viterbi algorithm where tokens whose summed log probabily is more than a given value (the so-called 'beam width') below the most likely token are deactivated, or 'pruned'. In the evaluation experiment, a beam width of 140 was used. It can be customized via the parameter **-hvitebeamwidth**.

### 3.5.3 Recognition result

The recognition result, which HVite writes into a so-called label file, is parsed. Optional inter-word pauses (<p:>) are ignored to reflect the fact that they do not appear in the transcription either. Internally, the recognition result is stored in a recognition result object. Like the transcription class, the recognition result class provides an iterator for token-by-token or phone-by-phone access (see Section 3.3.3). It has the same abstract parent class as the transcription iterators, and can therefore also be used polymorphically during alignment.

## 3.6 Alignment phase

The transcription and the recognition result can be represented and thus aligned as two sequences of symbols. For this purpose, naive Levenshtein edit costs [Levenshtein, 1966] are used, meaning that zero substitutions have cost 0, while all other edit operations (insertions, deletions and non-zero substitutions) have cost 1.

### 3.6.1 Wagner-Fischer algorithm

The most basic solution to the symbolic alignment problem, the Wagner-Fischer algorithm [Wagner and Fischer, 1974], represents the search space between the two sequences as a matrix. Every field in the matrix pairs one prefix of the first sequence with one prefix of

the second sequence. Starting at the top left corner, the matrix can be filled using dynamic programming such that every field contains the minimum total edit costs required to turn one of its prefixes into the other, using only the edit operations described above. Once all fields have been filled, the optimal alignment sequence, or path, can be discovered by tracing the fields with the minimum edit costs backwards, starting at the bottom right corner (see Figure 3.6).

| | | g | r | y: | n | l | I | C |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | | | | | |
| f | 1 | 1 | 2 | | | | | |
| r | 2 | 2 | 1 | | | | | |
| 2: | 3 | 3 | | | | | | |
| l | 4 | 4 | | | | | | |
| I | 5 | 5 | | | | | | |
| C | 6 | 6 | | | | | | |

Filling the matrix

| | | g | r | y: | n | l | I | C |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| f | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| r | 2 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2: | 3 | 3 | 2 | 2 | 3 | 4 | 5 | 6 |
| l | 4 | 4 | 3 | 3 | 3 | 3 | 4 | 5 |
| I | 5 | 5 | 4 | 4 | 4 | 4 | 3 | 4 |
| C | 6 | 6 | 5 | 5 | 5 | 5 | 4 | 3 |

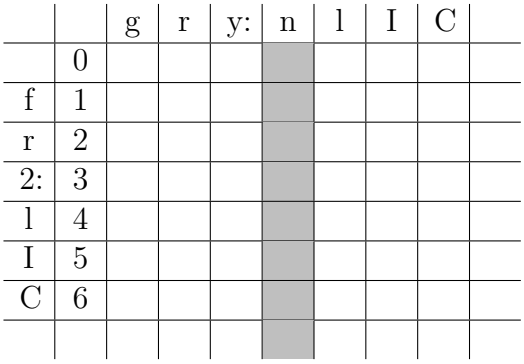
Backtracing the optimal path

Figure 3.6: Wagner-Fischer algorithm

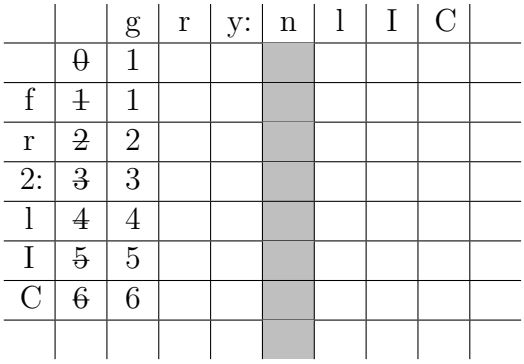
Since all matrix fields have to be calculated once, the Wagner-Fischer algorithm has runtime complexity $O(m \times n)$, with $m$ and $n$ representing the lengths of the symbolic sequences. If the alignment path is of interest (as it is in this case), memory requirements are also $O(m \times n)$, as all matrix fields have to be stored until the end, so that the optimal path can be traced back. This is an issue, especially in the case of the TP and P chunkers, whose symbol sequences are expressed in phones instead of tokens and therefore longer. At a speaking rate of ten phones per second, the resulting matrix for a three-hour recording would have $(10 \times 3 \times 3600)^2 \approx 1.2 \times 10^{10}$ fields. If we assume that every field contains a four byte unsigned integer, this leaves us with memory requirements of around 44 GB, which exceeds the main memory capacities of most computers.

### 3.6.2 Hirschberg algorithm

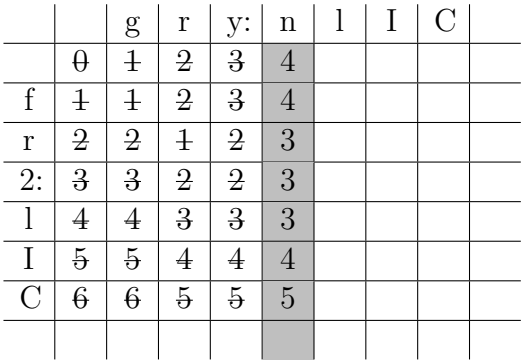
A solution to the memory issue is the Hirschberg algorithm [Hirschberg, 1975], a divide-and-conquer method that is also used by [Bordel et al., 2012]. The Hirschberg algorithm takes advantage of the fact that the minimum edit cost of two sequences (as opposed to their optimal alignment path) can be calculated with memory requirements $O(n)$. This is because, if the optimal path is not of interest, the Wagner-Fischer algorithm described above can 'forget' the first column of its matrix once the second one has been calculated, the second one once the third one has been calculated, and so on.



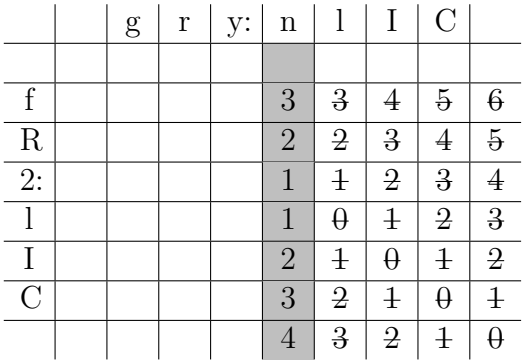
Calculating edit costs ...



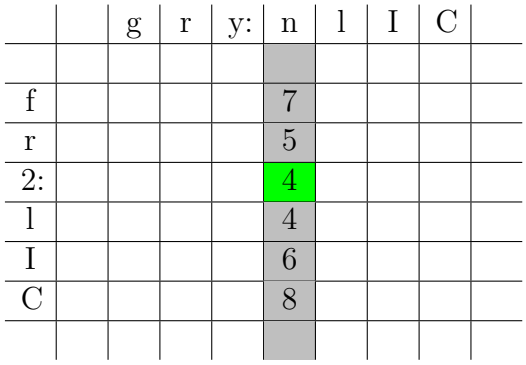
... with memory requirements $O(n)$



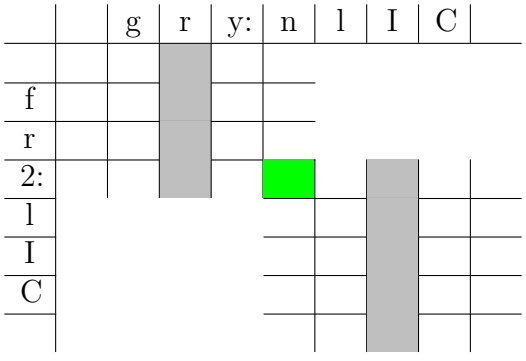
Middle column forward edit costs



Middle column reverse edit costs



Summing to find the optimal field



Splitting and recursion

Figure 3.7: Hirschberg algorithm

The Hirschberg algorithm uses this approach to calculate the edit costs for all fields in the middle column of the matrix (the column that cuts the longer of the two symbolic sequences in half). The second piece of information needed for every field in that middle column is its reverse minimum edit cost, i.e. the minimum cost incurred when performing the same procedure, but starting in the bottom-right corner and reading both symbolic sequences from the end. The field in the middle column that has the smallest sum of forward and reverse minimum edit costs is guaranteed to lie on the optimal path.

At this point, the problem can be reduced to two sub-problems: finding the optimal path that leads from the top left corner of the matrix to the discovered field, and the optimal path that leads from the discovered field to the bottom right corner. For each problem, only part of the matrix has to be considered. This is because, as the alignment path can only go down or to the right, areas to the bottom left and top right of the discovered field are ruled out. The Hirschberg algorithm can be applied recursively to the top-left and bottom-right submatrices to find more fields for the optimal path, until it is complete. It is obvious that every recursion requires fewer resources than the previous one (see Figure 3.7).

#### Parallelization

Due to the fact that every field of the matrix has to be considered at least once, half the fields at least twice, a quarter of the fields at least four times, and so on, the Hirschberg algorithm requires twice as many operations as the Wagner-Fischer algorithm. However, this factor can be canceled out by a multithreaded approach: Firstly, the forward pass and the backward pass towards the middle column of the initial matrix are independent of one another and can therefore be computed in parallel. Secondly, the submatrices to the top left and bottom right of the discovered field are independent of one another, and therefore also parallelizable.

The impact of parallelization is limited, though. Once the forward and reverse edit costs of the initial middle column have been computed, $n \times m$ operations have already been performed. In other words, the $n \times m$ operations can be distributed over no more than two threads. This means that the impact of an increased number of CPUs on the alignment phase is very limited, in contrast to the recognition phase, where all additional CPUs have the same impact (as long as the value of the **-splitduration** parameter is small enough).

### 3.6.3 Token-by-token versus phone-by-phone alignment

Whether the symbols of the aligned sequences represent tokens (T chunker) or phones (TP and P chunker) is formally irrelevant for both the Wagner-Fischer and the Hirschberg algorithm. It does however make a difference in granularity. Consider the case where the token sequence 'the others and' has mistakenly been recognized as 'three other sand'

by HVite. This phrase can be considered a phonetic near-miss, due to the segment-level similarities. The chunker should therefore have an incentive to assume that 'the others and' is located in the area where 'three other sand' has been recognized.

However, when using a token-by-token approach to alignment, there are no possible zero-substitutions between these two sequences, meaning that the aligner does not gain anything from aligning them (see Figure 3.8). And even if the sequences did end up being aligned, the chunk boundary selection module (see Section 3.7) would consider them just as bad an anchor as the alignment of two completely dissimilar three-word phrases.

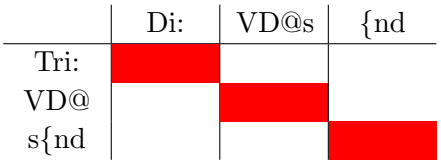



Figure 3.8: Granularity of token-by-token alignment (T chunker)

When using phone-by-phone alignment on the other hand, phonetic similarities are taken into account. In this case, the above-mentioned scenario would make for eight zero-substitutions (see Figure 3.9). This does not only increase the chance of this alignment being successful in the first place, it also means that the chunk boundary selection module will consider it a better candidate anchor than the alignment of two less similar sequences. This is exactly the intended behavior, and this is why it was hoped that the TP chunker would be more successful than the T chunker at finding chunk boundaries.

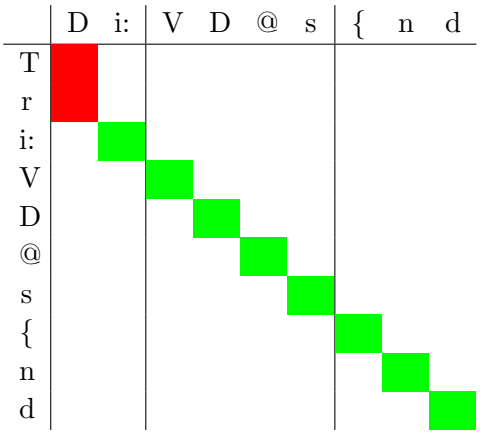



Figure 3.9: Granularity of phone-by-phone alignment (TP and P chunkers)

### 3.6.4 Implementation of the alignment path

So far, alignment has been treated as an operation on sequences of symbols. Internally, however, the alignment module operates on pointers to segments of the transcription (NULL for a deletion) and pointers to segments of the recognition result (NULL for an insertion). The object that is returned (the alignment path) is also made up of pairs of these pointers. This is to ensure that additional information, such as the time in the recognized signal or the position in the transcription, are available to the chunk boundary selection module.

## 3.7 Chunk boundary selection phase

### 3.7.1 Path traversal

In the chunk boundary selection phase, the path outputted by the alignment module is traversed, checking at every step whether the current alignment operation is the beginning of a candidate anchor, i.e. a sequence of alignment operations that satisfies the following criteria:

- The sequence's length is at least the value of the parameter **-minanchorlength**.
- The edit costs of the contained alignment operations sum to no more than the value of the parameter **-maxanchorcost**.
- The sequence contains at least n singletons, i.e. transcription tokens that occur exactly once in the chunker's transcription[4], with n being the value of the parameter **-minanchorsingletons**. This is done because it was feared that frequently recurring token sequences (e.g. from backchanneling behavior) would trick the alignment module into mapping non-identical sequences with the same content onto one another.

Every sequence of alignment operations that satisfies the criteria is added to a set of candidate anchors.

### 3.7.2 Candidate anchor set reduction

At this point, the set of candidate anchors may contain anchors that are too close to one another to obey the value of the **-minchunkduration** parameter, or even overlapping anchors. Therefore, some or even most must be rejected in the next step. To do so, the candidate anchors are first sorted according to the following criteria:

[4]As the TP and P chunkers perform phone-by-phone alignment, their anchors may contain transcription tokens partially. In this case, containing part of a singleton is considered sufficient for incrementing the singleton count by one.

- Long anchors come before short anchors
- Anchors containing long inter-word pauses come before anchors containing no or short inter-word pauses, with the longest pause within each anchor acting as the sorting key[5].

After that, the highest-ranked anchor from the ordered set is examined. Recall that every anchor is a subsequence of the alignment path, i.e. a sequence of alignment operations that map transcription segments onto recognition result segments. An anchor containing m operations thus contains up to m-1 segment boundaries. All segment boundaries that satisfy the following criteria are added to a separate set of word boundaries:

- The boundary is situated between two zero substitutions
- The segments to the left and right of the boundary are associated with different BPF KAN tier token indices (i.e. the current boundary is a word boundary)

The resulting word boundary set is sorted, with the duration of the word boundary's inter-word pause being the sorting key. This sorted set is then iterated over, asking for every word boundary whether it is at least the minimum chunk duration away from all other chunk boundaries found so far, including the start and end of the current chunker's audio signal. If this is the case, the word boundary or, if applicable, the mid-point of the inter-word pause, is selected as a chunk boundary and added to a result set. Anchors for which no eligible word boundary is found are rejected.

After this, the next anchor from the candidate anchor set is examined. This step is repeated until there are no more candidate anchors left to consider.

### 3.7.3 The -silenceonly parameter

If the **-silenceonly** parameter is set to a value larger than **0**, only word boundaries that span across an inter-word pause of at least that duration (in ms) are considered as output chunk boundaries. Else, word boundaries with inter-word pauses are prioritized (see above), but not to the exclusion of pause-less word boundaries. Setting this option makes most sense on clean material that contains many pauses, such as read speech. With spontaneous speech, or in the presence of background noises, the risk of reducing the number of discovered chunk boundaries was found to be too big. Therefore, the **-silenceonly** option was set to **0** in all evaluation experiments with the exception of the audio book segmentation (see Chapter 5).

[5]While optional inter-word pauses (<p:>) are ignored when parsing the recognition result, their durations can be deduced by comparing the start and end times of neighboring segments.

## 3.8 Recursion phase

If the **-recursion** option is set to **1**, at least one chunk boundary has been found, and the current chunker's depth in the recursion tree is smaller than the value supplied via the **-maxrecursiondepth** argument, the chunker enters the recursion phase. In this phase, a child chunker is started for every interval between two neighboring chunk boundaries (including the current signal's start and end time), provided that the interval has a duration of at least twice the value of the **-minchunkduration** parameter.

More specifically, the corresponding slice from the parent's HTK parameter file is cut out using HTK's HCopy tool, and transferred to the child chunker's working directory. Similarly, the child chunker constructs its transcription object by cutting a slice from the parent's transcription object. The child chunker then enters the language model phase, meaning that it creates its own HTK lattice, dictionary and HMM list based on its transcription slice. Since child chunkers operate on subsections of their parents' transcriptions and HTK parameter files, it can be assumed that their language models are more strongly attuned to the content of their signals. This in turn can lead to them being able to find chunk boundaries that their parent could not discover.

The chunk boundaries found by a child chunker are propagated into the parent chunker's result set. If that chunker is also a child chunker, its result set is again propagated to the next-highest chunker in the recursion tree. This way, all discovered chunk boundaries end up at the topmost chunker.

### 3.8.1 Recursion runtime

The impact of the recursion phase on overall runtime is harder to predict than that of the other phases, as it does not only depend on audio duration and transcription length, but also on the number and position of the chunk boundaries found at every iteration. Two extreme cases can be singled out:

#### Fast case

If the topmost chunker in the recursion tree does not find any chunk boundaries at all, or if it finds a sufficient number of chunk boundaries to fully cover the signal in chunks of duration less than twice the value of the **-minchunkduration** parameter, the recursion phase is skipped. In other words, the recursion phase does not affect total runtime at all in this case.

### Slow case

Now consider the case where the topmost chunker finds a single chunk boundary right next to its beginning. In this case, two child chunkers would be spawned: a fast one for the chunk spanning from the beginning to the boundary, and a slow one for the remainder, which would take practically as long as its parent. If that slow child again finds a single chunk boundary right at its edge, and so on, this process would be repeated again and again, with runtimes becoming only marginally faster between iterations.

In practice, this pattern would be stopped at the value of the **-maxrecursiondepth** parameter, which acts as an 'emergency brake' in the recursion tree. This means that the worst case runtime of the chunker would be approximately **-maxrecursiondepth** times the runtime of the topmost chunker.

To see where the chunker algorithms lie between the fast and the slow case, log files from the tests run in Chapters 4 and 5 were screened. It was found that the T and TP chunkers spend on average more time running the topmost chunker than running all child chunkers combined, while the P chunker spends between 1.5 and 2 times as much time running all child chunkers combined, compared to the topmost chunker. The **-maxrecursiondepth** value of 10 was reached in three out of a total of 1686 chunker calls, suggesting that the 'emergency brake' is mostly unnecessary.

### Parallelization

It is, in principle, possible to parallelize the runtime of sibling child chunkers. In practice however, this proved difficult to achieve, as threads used for child chunkers, HVite and the Hirschberg algorithm can create dead-locks or near-dead-locks, effectively increasing instead of decreasing runtime. Therefore, and due to the fact that the impact of recursion on global runtime proved smaller than feared, these efforts were abandoned. At present, child chunkers are therefore run sequentially.

## 3.9 Output phase

The output function is called by the topmost chunker only. Once this chunker has collected the chunk boundaries of all its children, it groups the tokens contained in its transcription accordingly and writes the result to a TRN tier. This TRN tier may be written in isolation, or it may be appended to the original BPF file, based on the value of the parameter **-outtype**.

If the **-maxchunkduration** option is set to **0**, all chunks are written to the TRN tier regardless of their duration. While this guarantees full coverage of the audio file, it also introduces the risk of including chunks that are too long for immediate processing with

MAUS. In some special cases, users might therefore wish to exclude chunks that are longer than a certain threshold. In this case, the **-maxchunkduration** option should be set to that threshold, leading to the output function ignoring all chunks that are longer. As MAUS ignores any material that is not included in a TRN tier segment (provided that TRN tier processing is enabled, of course), this would obviously result in an incomplete phonetic segmentation. Therefore, the option should only be used in scenarios where this is acceptable.

# Chapter 4

# Evaluation

In Section 1.3.1, the following benchmarks were formulated for the chunker:

- **Benchmark I**: At least 95 % of chunk boundaries are no more than 110 ms from their ground truth location.
- **Benchmark II**: At least 95 % of tokens end up in chunks of duration less than 5 minutes.
- **Benchmark III**: The chunker should not significantly deteriorate MAUS's word segmentation accuracy.
- **Benchmark IV**: The phonetic segmentation of audio files up to three hours – meaning the combination of chunk segmentation and subsequent MAUS segmentation – should be possible within less than real-time.

In the following, Section 4.1 deals with benchmarks I and II. Section 4.2 deals with benchmark III, and Section 4.3 with benchmark IV.

## 4.1 Chunk boundary errors and token accessibility

### 4.1.1 Material

Since the chunker was developed for long transcribed recordings, a corpus of long audio files with a reliable ground truth segmentation at the word level was needed. Since no such corpus was found, it was artificially created by concatenating data from the German subsets of the Verbmobil 1 (VM1)[1] and Verbmobil 2 (VM2)[2] corpora [Burger et al., 2000] (recording sessions prefixed k*, l*, m*, g*, z*, j* and w*).

[1] <hdl.handle.net/11858/00-1779-0000-001D-BFEE-3> (last accessed 02/07/2016)
[2] <hdl.handle.net/11858/00-1779-0000-001F-7D95-0> (last accessed 02/07/2016)

### The corpora

The Verbmobil corpora contain dialog turns from speakers negotiating their timetables in an imaginary business scenario. In total, 13900 German VM1 turns and 15208 German VM2 turns were found. For every turn, there is one mono NIST SPHERE file (sampling rate 16000 Hz, sample width 2 bytes, close microphone)[3] and one BAS Partitur Format file. The BPF files contain - among others - orthographic (ORT) and canonical pronunciation (KAN) tiers as well as at least one type of phonetic segmentation (see below). The average turn duration is 6.8 seconds, with a standard deviation of 7.0 seconds and a maximum turn duration of 85 seconds.

### Ground truth segmentation

A small subset of the turns contain a manual phonetic segmentation, called the Sampa (SAP) tier, while the majority of turns only contain a phonetic segmentation derived by MAUS (MAU tier). It is of course problematic to test the output of one computer program (the chunker) against the output of another computer program (MAUS). Nonetheless, the fact that the MAUS segmentation of the Verbmobil material had been positively evaluated [Schiel et al., 2011] was taken as an indicator that it would make an acceptable ground truth. Note also that the MAUS segmentation had been performed on the original short turns. This means that, even in the worst case, a given MAUS-segmented ground truth segment could never be further off its target than by the duration of the turn it had originally been contained in.

The manually segmented SAP tier was used for the evaluation of the effects of chunker preprocessing on MAUS (see Section 4.2).

### Data preparation

First, the NIST SPHERE audio files were converted into WAVE format. The BPF files were manipulated as follows, using a Python module implemented for this purpose:

- Any tiers other than ORT (orthography), KAN (canonical pronunciation) and MAU (MAUS segmentation) were deleted.
- The ORT and KAN tiers were used as is.
- The MAU tier was used for the creation of an automatic word segmentation tier, called AWO[4], and then deleted.

[3] Some sessions in Verbmobil 2 contain additional audio files recorded via telephone or room microphone, but these were not used.

[4] Note that this tier is not from the official BPF standard.

### Concatenation

In a next step, the turns were concatenated to derive audio files of duration one hour. For this, they were ordered alphabetically according to their file names. They were then concatenated by iterating over the sorted list and appending the WAVE files to one another and the BPF files to one another, making sure that indices and start times in the BPF files were correctly incremented. Once the target duration of one hour had been passed, the concatenated recording-BPF pair was saved as VM1####.wav/par or VM2####.wav/par, with #### being a running index starting at 0000 for both corpora. Then, the process started anew, beginning with the next turn. This resulted in a total of 53 one-hour recordings[5].

### Development and evaluation data

To ensure a clean evaluation, parameter fine tuning and practice tests were run exclusively on concatenated recordings with odd indices (starting with VM10001.wav/par), while the final evaluation experiments were run on file pairs with even indices (starting with VM10000.wav/par).

### Manipulations

The Verbmobil transcriptions and audio material are high quality, and therefore not necessarily representative of all types of material that may require chunk segmentation. Therefore, three types of manipulation were performed on the data: omission of single transcription tokens, omission of transcription speaker turns, and artificial cross-talk. Every type comes in five levels of severity, which are detailed in Table 4.1.

The first type of manipulation is supposed to model omission errors in the transcription process. To achieve this effect, single tokens were deleted from the source BPFs at regular intervals, taking care to decrease the indices of subsequent tokens accordingly. The WAVE files were kept as is. See Table 4.1 for the percentages of tokens missing from the transcriptions at every level.

For the second type of manipulation, speaker turns were omitted from the source BPFs to mimic the effect of an interview where only the interviewee's turns are transcribed. More specifically, the concatenation step was repeated, with the difference that at every n'th turn, the BPF was ignored but the WAVE file was appended, as if the latter had not been transcribed. See Table 4.1 for the percentage of speaker turns missing at every level.

[5] Due to a bug in the concatenation script, the last batch of turns from each corpus was accidentally ignored. In Verbmobil 1, this affects turns alphabetically lower than 'n198kxx0_008_OS9', amounting to 2.3 % of the material. In Verbmobil 2, it affects turns alphabetically lower than 'g626bch2_011_BHP' amounting to 3.8 % of the material.

The third type of manipulation, artificial cross-talk, was created by overlaying every recording with a background recording. The background recording was made up of the second half of the original recording, followed by the first half. Thus, it contained the same speakers and the same material as the original, but in the wrong locations. The transcribed audio file (the signal) and the background recording (the 'noise') were overlayed at the signal-to-noise ratios specified in Table 4.1, using [SoX, 2015]. The appended USB drive contains example recordings for all signal-to-noise ratios.

| Condition | Level 0 | Level 1 | Level 2 | Level 3 | Level 4 | Level 5 |
|---|---|---|---|---|---|---|
| Deletion of tokens | none | 2% | 5% | 10% | 20% | 25% |
| Deletion of turns | none | 5% | 10% | 20% | 25% | 50% |
| Cross-talk (SNR) | none | 20 dB | 15 dB | 10 dB | 5 dB | 3 dB |

Table 4.1: Degree of distortion at different levels of manipulation

### 4.1.2 Metric

#### Chunk boundary errors

A chunk boundary error is defined as the distance of a predicted chunk boundary from either the ground truth end of the last token of the previous chunk, or the ground truth beginning of the first token of the next chunk, whichever is closer. In the case where there is a pause between the two tokens in the ground truth segmentation, any position within that pause is considered a zero error.

#### Token accessibility

The chunker's goal is to make material accessible to MAUS. Since material in short chunks is more accessible than material in long chunks, a useful accessibility metric is chunk duration by token, where every token in the transcription reports the duration of the chunk that it is contained in. This way, chunks containing a lot of material weigh more heavily in the resulting token accessibility distribution than chunks containing little material.

It would of course also be possible to use chunk duration by individual chunk, as opposed to chunk duration by token. However, it was decided that this metric could be misleading. Consider the hypothetical case where the first 1000 tokens of a 10000 token transcription have been split into chunks of duration 10 seconds, while no chunk boundaries have been found among the remaining 9000 tokens. When using chunk duration by individual chunk as a metric, this result might look favorable: after all, there are many short chunks of duration 10 seconds, and only one long outlier. However, this metric would be overlooking the fact that 90 % of the original material is still inaccessible, due to being contained in

the still-too-long last chunk. The chunk duration by token metric on the other hand avoids this kind of oversight.

### 4.1.3 Experiment

The clean and manipulated data were inputted into the T, TP and P chunkers with the parameters specified in Table 4.2. The resulting BPF files were examined using a Python script, to measure token accessibility and chunk boundary errors as described in Section 4.1.2.

### 4.1.4 Results

Figure 4.1 shows token accessibility and chunk boundary error distributions as boxplots on the 27 clean one-hour recordings included in the evaluation set. Figures 4.2, 4.3 and 4.4 show token accessibility and chunk boundary error distributions on the same recordings at different levels of manipulation, as described in Table 4.1. Note that the boxplots' whiskers represent 5 % and 95 % percentiles, and that plots labeled 'zoom' are zoomed-in versions of the plots above or next to them. Note also that the yellow horizontal lines correspond to benchmarks I and II.

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| **-minchunkduration** | 6 | **-maxchunkduration** | 0 |
| **-languagemodel** | 2 | **-recursion** | 1 |
| **-silenceonly** | 0 | **-maxnumthreads** | 3 |
| **-outtype** | bpf | **-splitduration** | 120 |
| **-hvitebeamwidth** | 140 | **-bigramweight** | 0.5 |
| **-languagemodelweight** | 4.0 | **-maxrecursiondepth** | 10 |
| **-aligner** | hirschberg | **-minanchorsingletons** | 1 |

**Algorithm-specific parameters fine-tuned on development set**

| | **T chunker** | **TP chunker** | **P chunker** |
|---|---|---|---|
| **-maxanchorcost** | 0 | 1 | 1 |
| **-minanchorlength** | 3 (tokens) | 14 (phones) | 6 (phones) |

Table 4.2: Parameters used in evaluation experiments (see Appendix A for a manual)

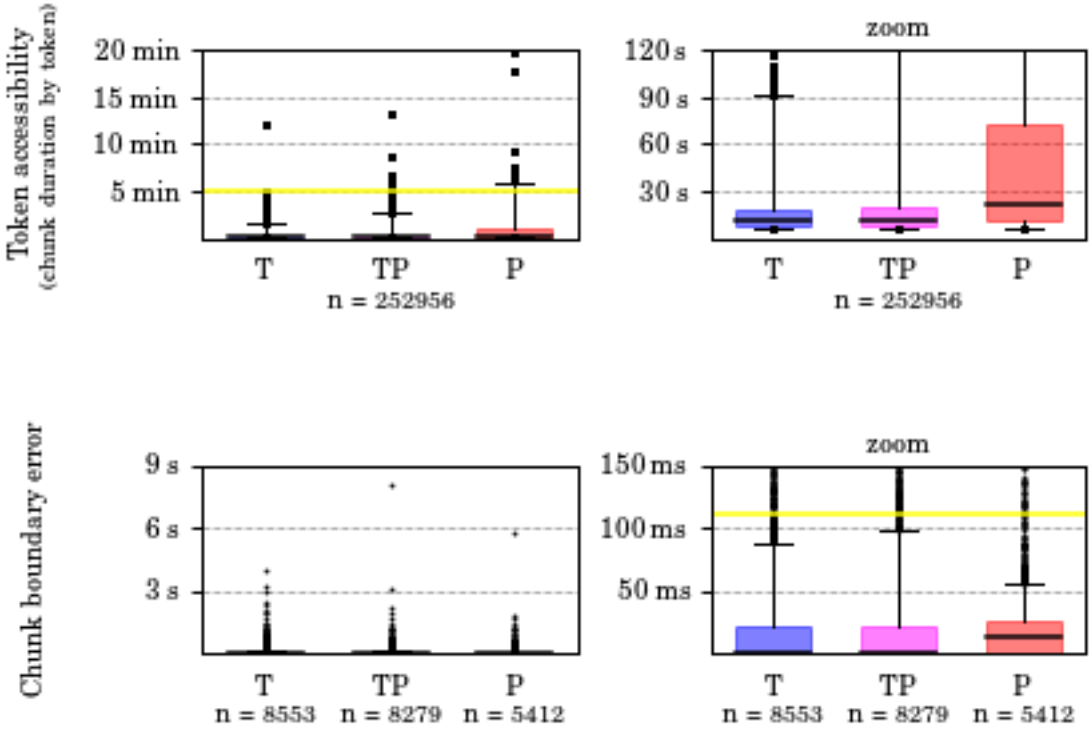



Figure 4.1: Token accessibility (chunk duration by token) and chunk boundary error distributions on clean data

### 4.1.5 Discussion

#### Clean data

In the output of the T and TP chunkers, at least 95 % of tokens are contained in chunks of duration below five minutes, meaning that they meet benchmark II. The P chunker misses the benchmark narrowly, with 95 % of tokens in chunks of duration below six minutes. Generally, the T chunker appears to be most successful at making material accessible, with only two chunks outside the 5 minute range.

As for chunk boundary errors, all three chunker algorithms meet benchmark I, with more than 95 % of chunk boundaries less than 110 ms (in fact less than 100 ms) off their target. The fact that the T and TP chunkers' medians are at zero suggests that at least 50 % of chunk boundaries correctly landed in an inter-token pause. The fact that there are fewer such zero errors in the P chunker output is probably due to its anchors being shorter, which means that there is less space for inter-token pauses to begin with. Generally, it is difficult to say which chunker is more accurate: while the P chunker makes fewer 'big' errors, the T and TP chunkers are more successful at correctly targeting inter-token pauses.

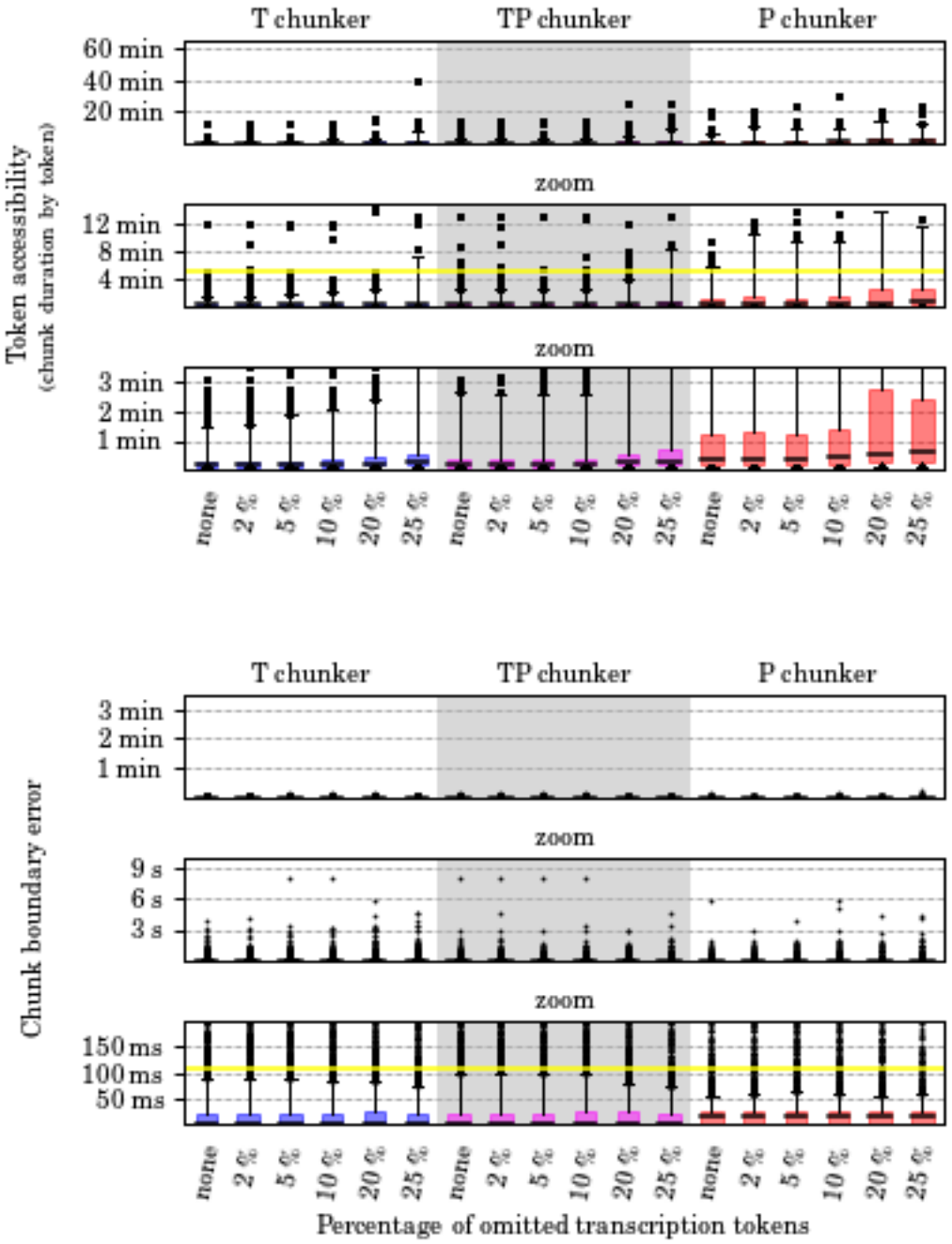



Figure 4.2: Token accessibility (chunk duration by token) and chunk boundary error distributions on data with omitted transcription tokens

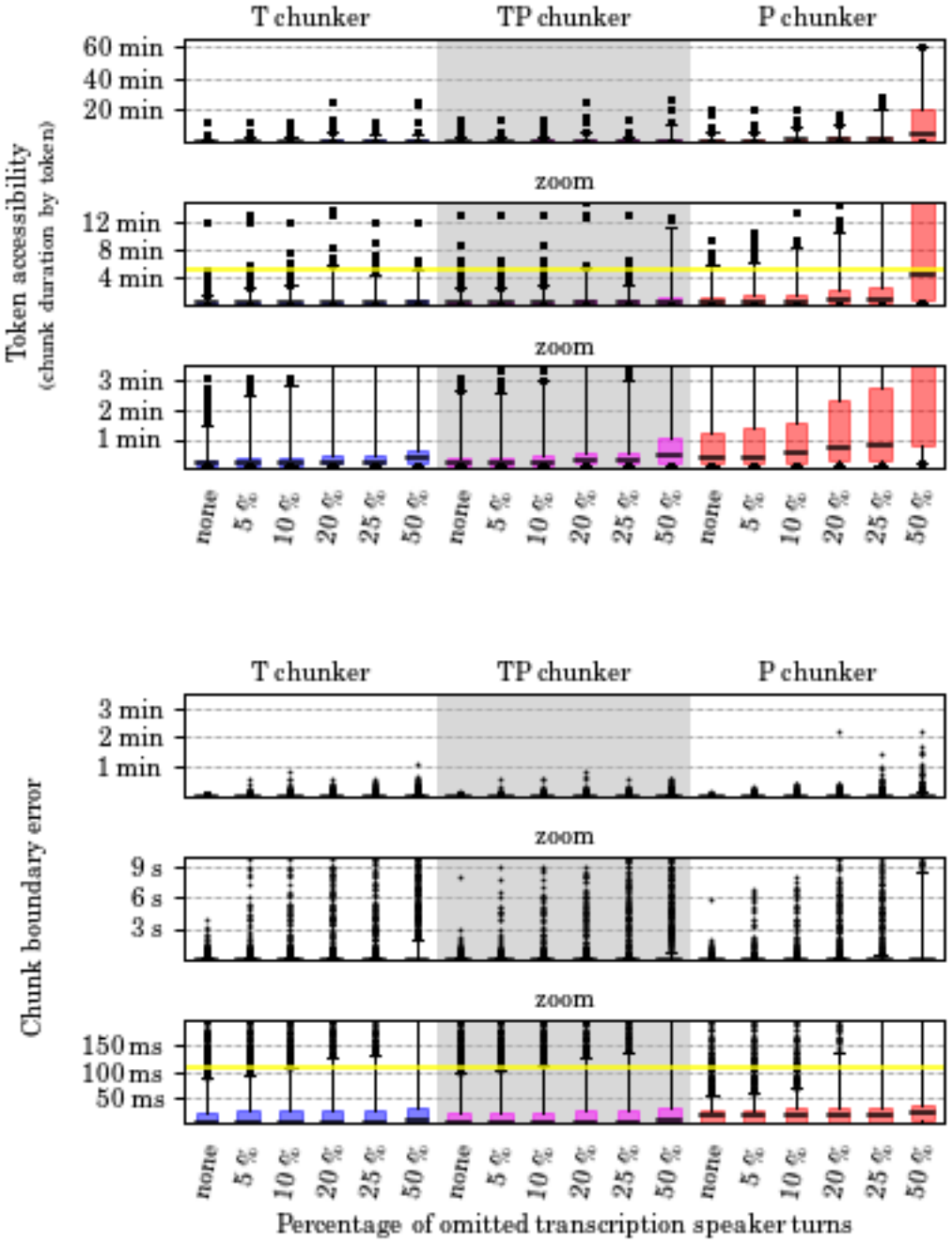



Figure 4.3: Token accessibility (chunk duration by token) and chunk boundary error distributions on data with omitted transcription speaker turns

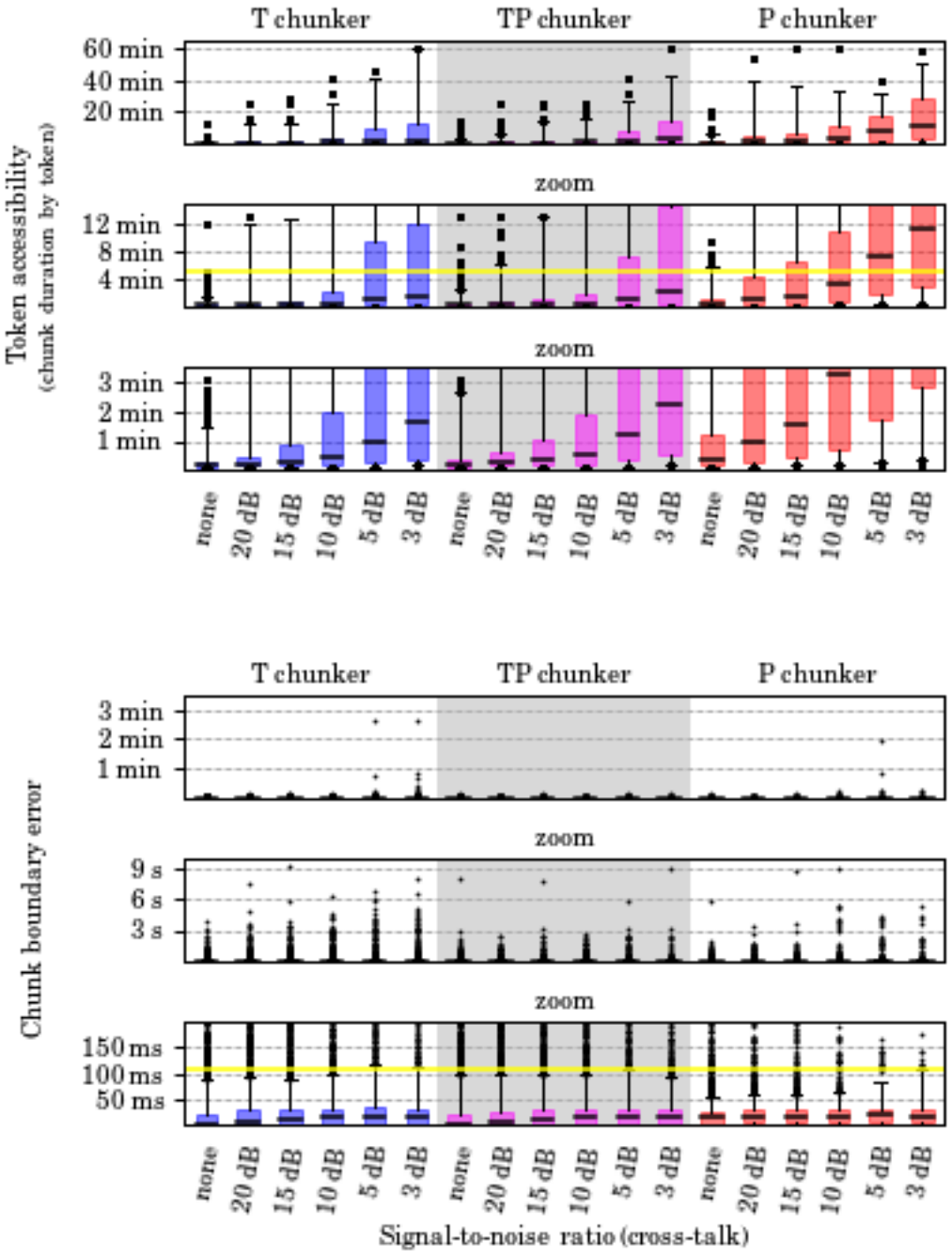



Figure 4.4: Token accessibility (chunk duration by token) and chunk boundary error distributions on data with cross-talk

### Omitted tokens

It appears that chunk boundary errors remain relatively stable in the presence of omitted transcription tokens, with all three chunkers meeting benchmark I in all conditions, even at the highest token omission rate of 25 %. In fact, there is a slight decrease in errors. However, this is likely to be an artefact, as the missing tokens are treated as pauses, making more room for zero-errors.

While errors remain relatively stable, token accessibility deteriorates a little. The T and TP chunkers meet benchmark II at token omission rates up to 20 %, but fail to do so at a token omission rate of 25 %. The P chunker continues to fail to meet the benchmark. However, its 95 % percentiles remain below 15 minutes, suggesting that most material would still be MAUS-segmentable in a reasonable amount of time, although not in real-time.

All in all, it appears that the T and TP chunkers are robust in the face of single-word omissions, and that the P chunker does not perform badly either.

### Omitted speaker turns

Contrary to the omission of single tokens, the omission of complete speaker turns from the transcription results in an increase of chunk boundary errors. Firstly, the magnitude of outliers grows considerably for all three chunkers, from the 10 second mark up to approximately 1 minute (T / TP) or over 2 minutes (P). This suggests that longer stretches of missing transcription material increase the risk of major misalignments. The T and TP chunker meet benchmark I at a turn omission rate of 5 %, and only miss it narrowly at turn omission rates up to 25 %. At a turn omission rate of 50 %, however, the 95 % error percentile grows to almost 3 seconds for the T chunker and to 9 seconds for the P chunker. This means that more than 5 % of chunk boundaries are off target by several seconds, which clearly does not count as a good chunk segmentation anymore.

As for token accessibility, the T chunker remains relatively stable and meets or narrowly misses benchmark II at all turn omission rates. The same is true for the TP chunker, with the exception of the turn omission rate of 50 %. The P chunker's ability to return short chunks on the other hand is severely affected, with the 95 % percentile for token accessibility rising to 20 minutes at a turn omission rate of 25 % and to 60 minutes at a turn omission rate of 50 %. The latter figure suggests that there were a few recordings where the P chunker did not find any chunk boundaries at all.

To summarize, it seems that the chunker's performance is negatively affected by large-scale transcription errors such as non-transcribed turns, with larger errors and less accessible material, especially in the case of the P chunker and at high turn omission rates.

### Cross-talk

Cross-talk appears to affect the chunker's chunk boundary error rates less strongly than turn omission. While the T chunker does not meet benchmark I at a signal-to-noise ratio of 5 dB or 3 dB, and the TP chunker does not at 5 dB, they miss it by a very narrow margin. However, the magnitude of outliers grows for the T and P chunkers, suggesting that grave misalignments, while rare, are possible in this kind of scenario. The TP chunker on the other hand does not appear to suffer from misalignments on this scale, as its maximum outliers remain at the level they had on clean data.

On the other hand, the chunkers become much less effective at finding short chunks, which is marked by the fact that none of the three algorithms meet benchmark II at any signal-to-noise ratio. The effect is especially prominent in the P chunker and at severe degrees of cross-talk.

Of course, this does not mean that the chunk segmentations are completely unsuccessful. In the T and TP chunker output, even at a signal-to-noise ratio of 3 dB, more than 50 % of the material is still MAUS-segmentable in real-time, and 75 % of the material is contained in chunks of duration up to 15 minutes, which means that they are MAUS-segmentable in up to approximately one hour per chunk (see Figure 4.10). It would, however, take manual chunk segmentation, or some other method, to deal with the rest.

Alternatively, there is of course the option of rerunning the chunker with less conservative values for the **-minanchorlength**, **-maxanchorcost** and **-minanchorsingletons** parameters. Whether this does, in fact, lead to better token accessibility, or whether it leads to unacceptably high error rates, remains to be tested.

### Summary

To summarize, the chunker is successful on both clean data, and on data where individual tokens are missing from the transcription. In the presence of missing speaker turns, the number of chunk boundary errors increases, especially if a lot of material is missing. In the presence of cross-talk, error rates remain mostly low, but the chunker becomes less successful at returning short chunks.

## 4.2 Effects on MAUS's word segmentation accuracy

### 4.2.1 Material

The effects of chunker preprocessing on MAUS's word segmentation accuracy were tested on a manually segmented subset of the Verbmobil 1 turns, which had served as a benchmark for MAUS before (recording sessions m112 through m117, m119, m222, m224, m230 and m232). When concatenated, the material amounts to just under one hour of speech. Due to the sparsity of manually segmented data, there is no split into a development and evaluation set.

Preprocessing and concatenation were performed in the manner described in Section 4.1.1. The only difference was that the manually segmented SAP tier was used instead of the MAU tier for the creation of the ground truth word segmentation[6]. Due to the expected long runtime of MAUS on long recordings, the input for the MAUS-only tests was presegmented into ten-minute chunks. The chunker on the other hand worked with the one-hour input, which effectively meant a head start for MAUS. In addition to the clean data, the manipulations described in Section 4.1.1 were applied to the data.

### 4.2.2 Metric

Segmentation accuracy was evaluated by calculating the distance between predicted word beginnings and ground truth word beginnings. Contrary to the evaluation of chunk boundary errors, there was no possibility for zero-errors (apart, of course, from directly hitting the correct ground truth word onset sample).

### 4.2.3 Experiment

The one-hour recording was inputted into the three chunkers, with the parameters listed in Table 4.2. Their output then served as input to MAUS, with the **USETRN** option set to **yes**, to enable TRN tier processing. The ten-minute presegmented input was fed to MAUS directly.

To test the statistical significance of the effect of chunker preprocessing, paired t-tests were conducted using Python-SciPy's ttest_rel function [Jones et al., 2001]. In every test, the word boundary errors made by the MAUS-only segmentation (the baseline) were tested against the word boundary errors made after using one of the three chunkers as a preprocessor. The tokens (or rather their indices in the concatenated BPF file) acted as the t-tests' pairing factor.

[6] SAP segments whose labels ended with a minus sign, indicating that they were elided, were ignored.

### 4.2.4 Results

Figure 4.5 shows the distribution of word boundary errors on clean data, while Figures 4.6, 4.7 and 4.8 show the distribution on manipulated data. For ease of inspection, the t-test results are printed below the boxplots of the chunker-preprocessed results, in the notation presented in Table 4.3. Like in Section 4.1.4, the boxplots' whiskers represent the 5 % and 95 % percentiles of the distribution, while plots labeled 'zoom' are zoomed-in versions of the plots above them.

| Symbol | Directionality of effect | P value |
|---|---|---|
| (o) | Improvement relative to MAUS-only segmentation | $p \geq 0.05$ |
| o | | $p < 0.05$ |
| oo | | $p < 0.01$ |
| ooo | | $p < 0.001$ |
| (+) | Deterioration relative to MAUS-only segmentation | $p \geq 0.05$ |
| + | | $p < 0.05$ |
| ++ | | $p < 0.01$ |
| +++ | | $p < 0.001$ |

Table 4.3: Notation for paired t-test results in Figures 4.5, 4.6, 4.7 and 4.8

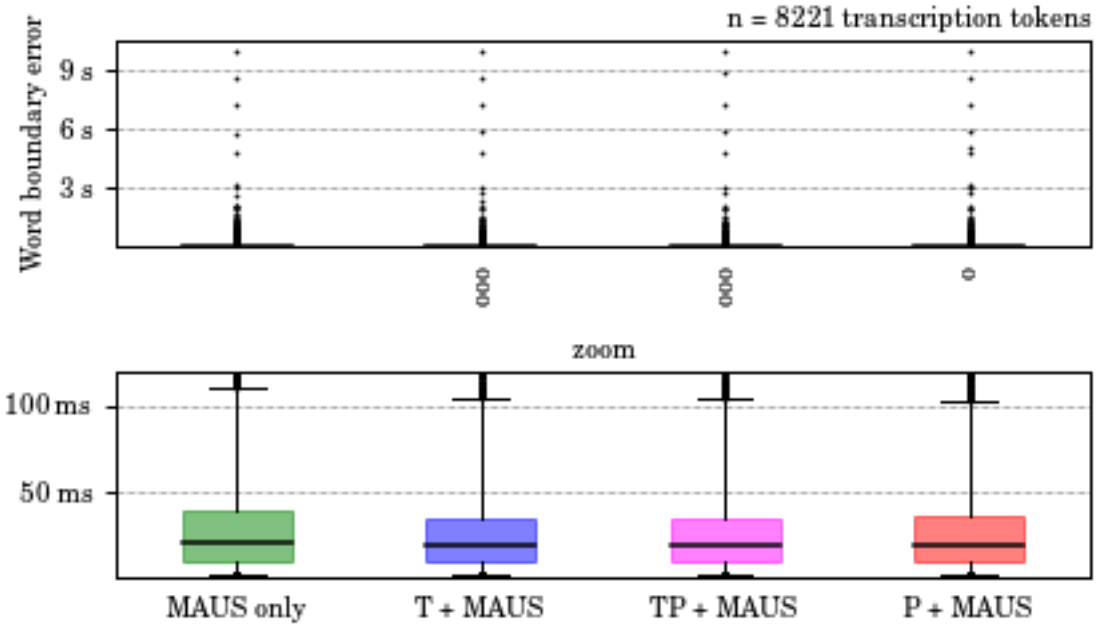



Figure 4.5: MAUS word boundary error distributions on clean data with and without chunker preprocessing

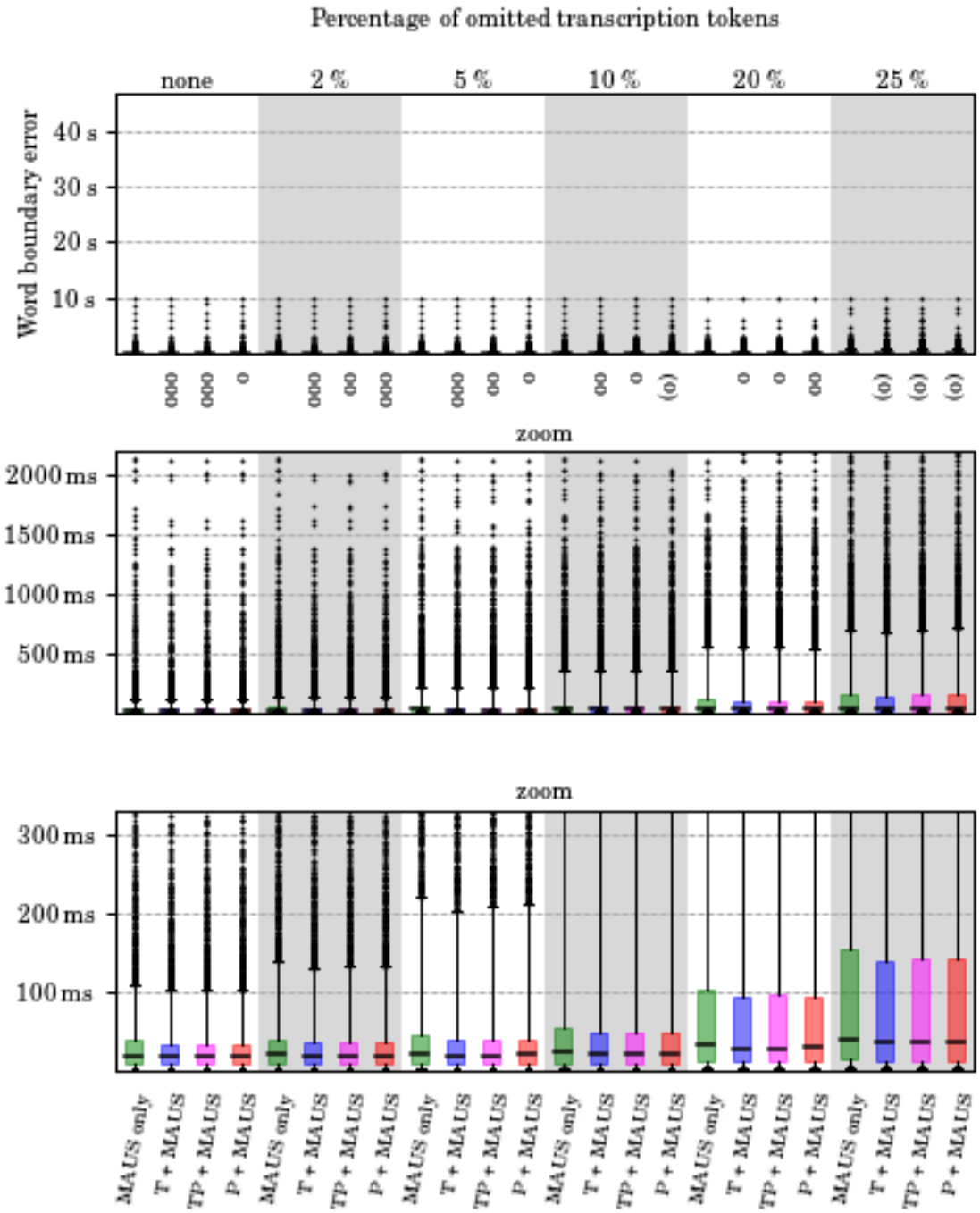



Figure 4.6: MAUS word boundary error distributions on data with omitted transcription tokens with and without chunker preprocessing

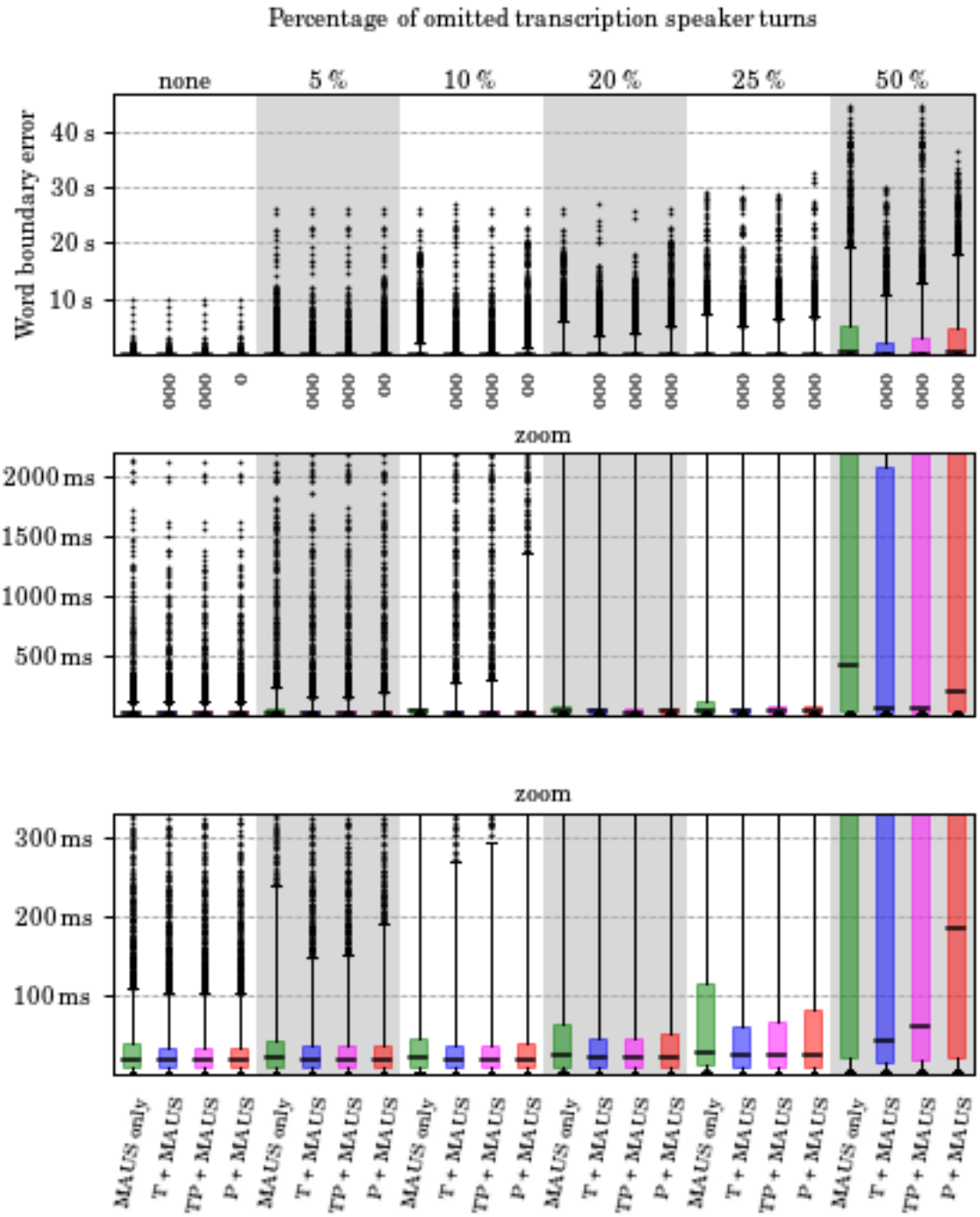



Figure 4.7: MAUS word boundary error distributions on data with omitted transcription speaker turns with and without chunker preprocessing

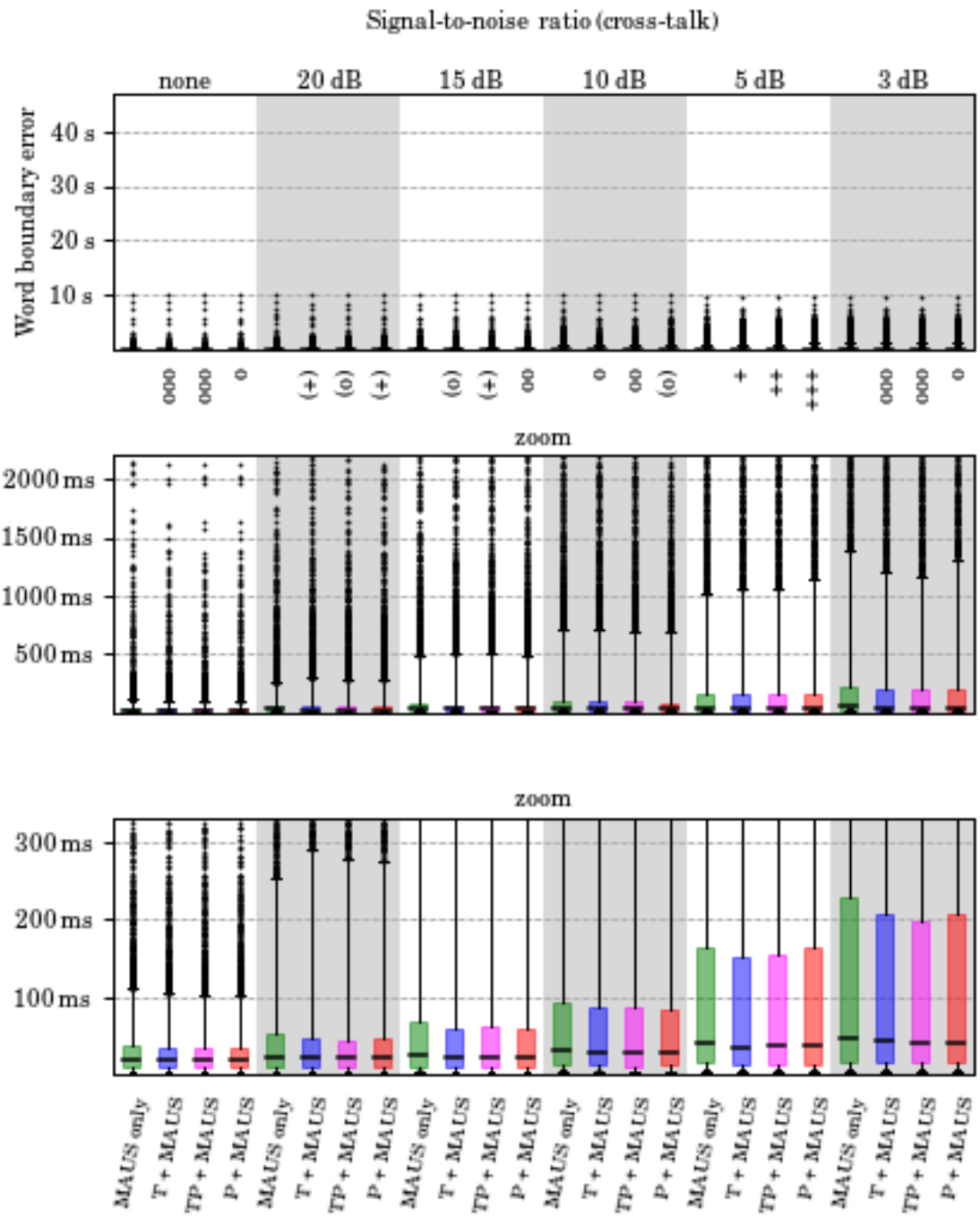



Figure 4.8: MAUS word boundary error distributions on data with cross-talk with and without chunker preprocessing

### 4.2.5 Discussion

#### Clean data

On the clean data, MAUS's word segmentation accuracy is significantly improved by chunker preprocessing, regardless of which chunker is used. The effect is however relatively small. This means that all three chunkers meet benchmark III on clean data, as they do not significantly decrease MAUS's word segmentation accuracy. It is interesting to note that the chunker does not appear to keep outlier word segmentation errors from happening, as the pattern of errors above the 3 second mark is identical for all four boxplots. This is somewhat surprising, as it had been expected that the chunker, by cutting MAUS input into small windows, would be most effective against big segmentation errors. However, at a minimum chunk duration of 6 seconds, it is possible that chunk granularity was simply too low to prevent them.

#### Omitted tokens

In the presence of missing transcription tokens, word segmentation accuracy is mostly significantly improved by the chunkers at lower token omission rates, while the effect is insignificant at 25 %. The chunker therefore meets benchmark III at all five levels of token omission. However, like in the clean data condition, effects are rather small.

#### Omitted speaker turns

In the case of speaker turn omissions, all three chunker algorithms significantly improve MAUS's word segmentation accuracy on all levels of manipulation, meaning that they meet benchmark III. This means that while the chunker is increasingly prone to errors in the turn omission scenario (see Figure 4.3), the errors made by MAUS on its own are even more severe. Nonetheless, it appears that MAUS generally finds it difficult to segment this kind of data, and that chunker preprocessing, while helpful, cannot prevent error rates skyrocketing as the rate of missing turns increases. Therefore, the take-home message would probably have to be not to use data with missing transcription turns in the first place.

#### Cross-talk

In the cross-talk scenario, there is a mix between significant improvements, significant deteriorations and insignificant effects of chunker preprocessing on MAUS's word segmentation accuracy. The significant deteriorations affect all three chunkers at a signal-to-noise ratio of 5 dB, meaning that benchmark III is not met in this condition. Interestingly, all three

algorithms significantly improve MAUS's word segmentation accuracy at the next-lowest signal-to-noise ratio of 3 dB, making for a somewhat unclear pattern. Generally, effects are relatively small, compared to the turn omission scenario.

#### Summary

To summarize, chunker preprocessing mostly does not significantly deteriorate MAUS's word segmentation accuracy, with the exception of material with cross-talk at a signal-to-noise ratio of 5 dB. Instead, chunker preprocessing often significantly improves MAUS's segmentation accuracy, especially on clean data and on data with omitted transcription turns.

## 4.3 Performance

### 4.3.1 Material

Chunker performance was tested on clean concatenated recordings of durations 5, 10, 15, 20, 30, 40, 50, 60, 80, 100, 120, 150 and 180 minutes. The performance of MAUS was tested on audio-transcription pairs of duration smaller or equal to 50 minutes, as tests on the longer material would have taken too long. Concatenation was done as described in Section 4.1.1, setting the temporal cutoff to the desired duration. Due to limitations on computational power and time, only the first concatenated recording (VM10000.wav/par) could be tested for each audio duration.

### 4.3.2 Metric

The first performance metric is the time taken by every chunker call as a function of input duration. The second performance metric is the combined runtime of chunker calls and subsequent MAUS calls, when working on the chunker output. This is done to compare the runtime of MAUS-only segmentation to the runtime of chunker-plus-MAUS segmentation.

### 4.3.3 Experiment

Each of the three chunker algorithms was evaluated on one, three, six and nine threads, by setting the **-maxnumthreads** option accordingly. All other parameters were set according to Table 4.2. The output BPFs were passed to MAUS, with the parameter **USETRN** set to **yes**. Tests were run on a 12 core HP ProLiant DL160 G6 server, which is also used by the MAUS web service.

### 4.3.4 Results

See Figure 4.9 for runtimes of the chunker alone, and Figure 4.10 for the summed runtimes of the chunker and the subsequent MAUS calls, on different numbers of threads.

### 4.3.5 Discussion

#### Chunker alone

The runtimes of all three chunkers grow super-linearly with input duration. This is consistent with the fact that all three algorithms have at least one super-linear component (the Hirschberg algorithm and, in the case of the T and TP chunkers, HVite). Regardless of the number of threads, the P chunker is fastest, followed by the T chunker and then the TP chunker. This is probably because phoneme-based recognition is faster than term-based recognition, due to the constant number of states in the HTK lattice. The extra time taken by the TP chunker, relative to the T chunker, is due to the fact that the symbolic alignment task involves a bigger matrix.

The differences between the three algorithms are most pronounced when running on a single thread, but shrink as more threads are added. This suggests that the advantage of using the P chunker diminishes as computational power grows. Generally, it appears that the P chunker benefits less from parallelization than the T and TP chunkers. This is consistent with the insight that the part of the runtime that is most easily parallelized – the recognition phase – is less prominent in the P chunker than it is in the T and TP chunkers. Generally, the effect of parallelization appears to decrease with the number of threads, with the step from one to three threads having the greatest effect.

#### Chunker plus MAUS

It is obvious, from Figure 4.10, that chunker preprocessing decreases total segmentation time. The difference between the MAUS-only segmentation and the chunker-plus-MAUS segmentations becomes especially pronounced at longer input durations, above the ten minute mark, where segmentation with MAUS alone becomes increasingly impractical. The P chunker meets benchmark IV regardless of the number of threads. The T and TP chunkers do not meet the benchmark on one thread, but they do on three threads or more.

It is however important to point out that the runtime of MAUS, when working on chunker output, is dependent on the duration of the chunks discovered. While these durations were rather short on the clean material used here, they might be longer in the presence of disturbances such as cross-talk (see Figure 4.4). In the worst-case scenario, where the chunker is unable to find any chunk boundaries, MAUS working on the output should take as long as MAUS on its own, plus the overhead of the unsuccessful previous chunker run.

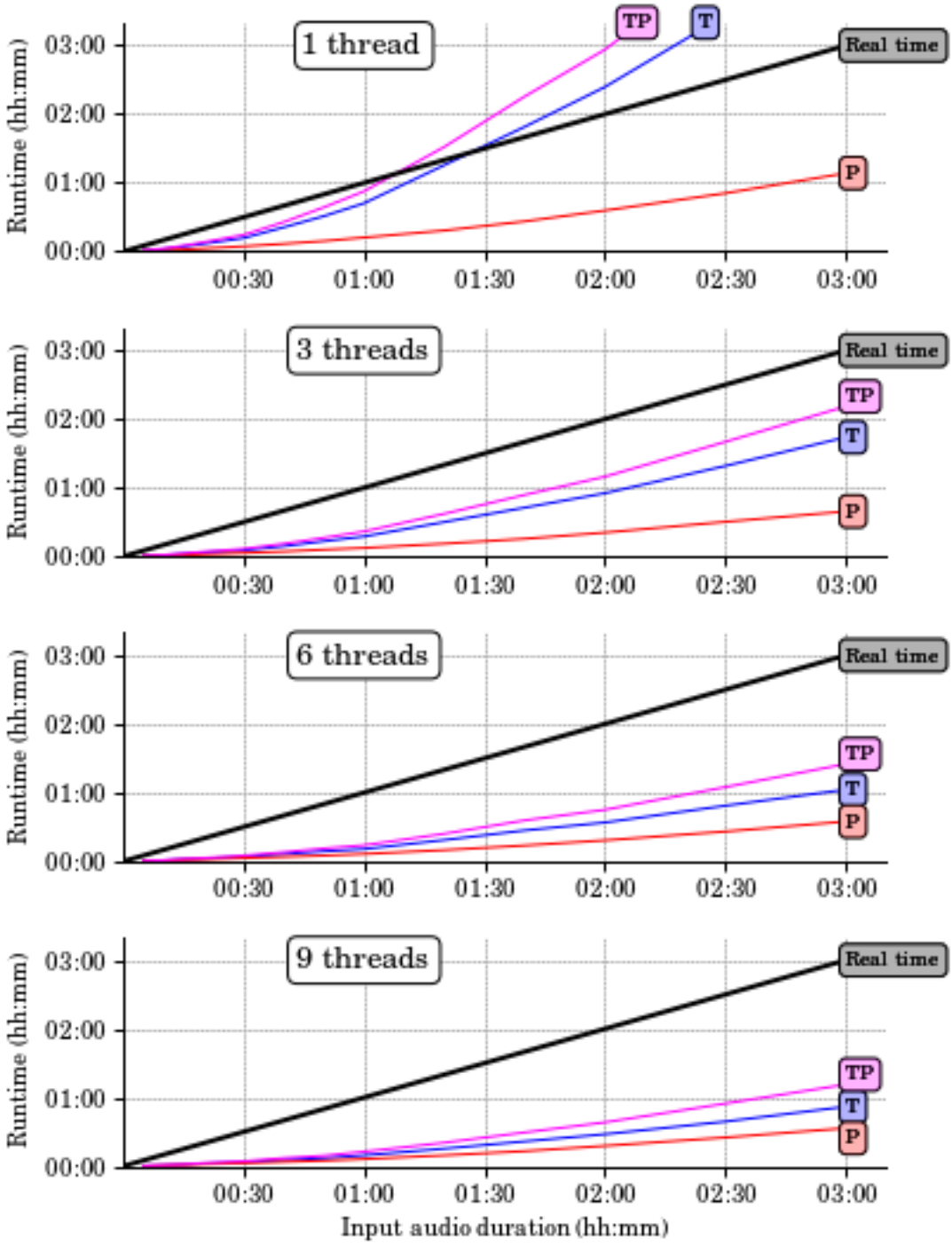



Figure 4.9: Runtime of chunker algorithms as a function of input audio duration, at different levels of multithreading

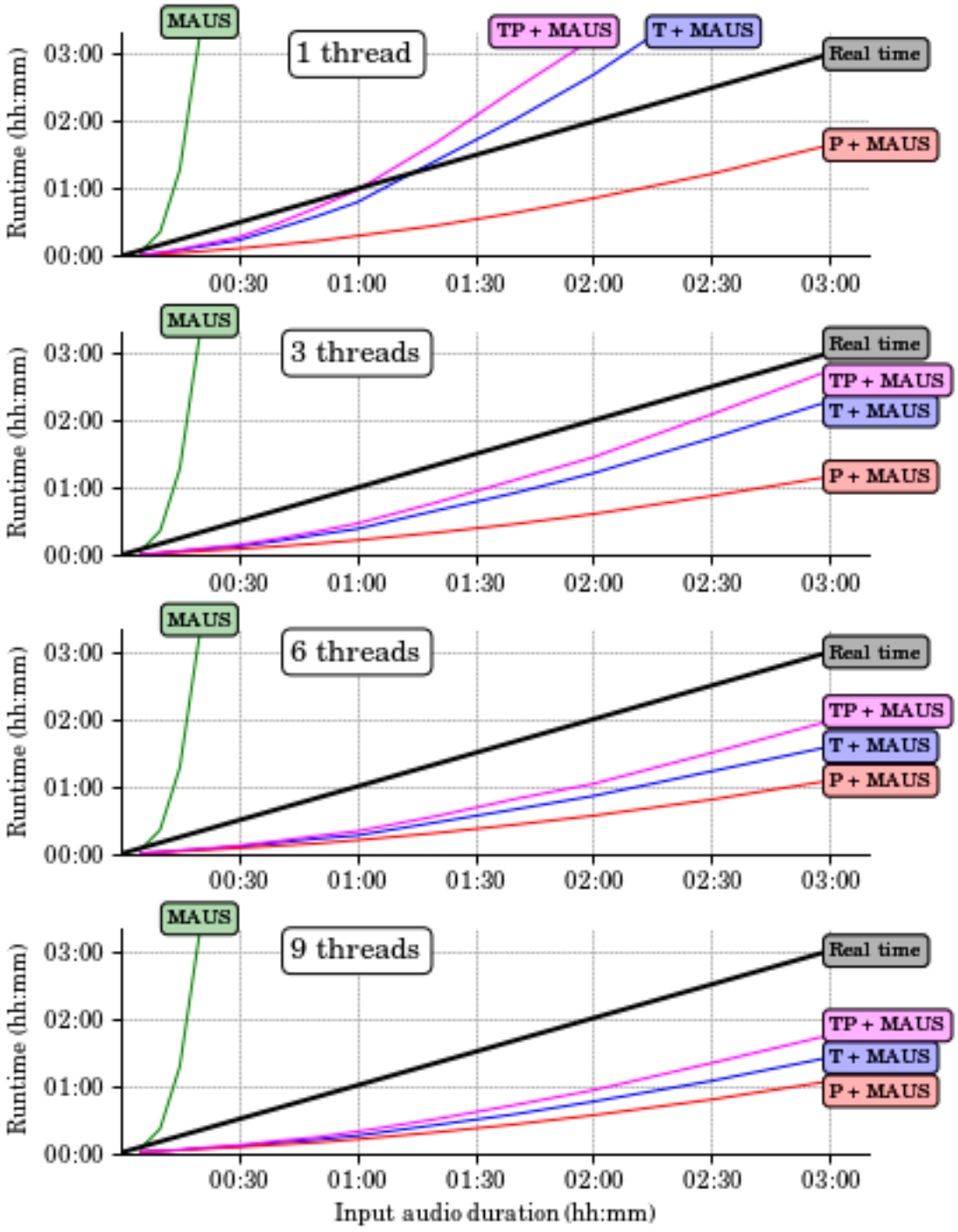



Figure 4.10: Runtime of chunker-plus-MAUS and MAUS-only phonetic segmentation as a function of input audio duration, at different levels of multithreading

# Chapter 5

# Audio book segmentation

The experiment described in this chapter is meant to reflect a more realistic use case for the chunker. The three chunker algorithms are used in a tool chain from raw data to phonetic segmentation on material that is not derived from a phonetic production experiment, and therefore more likely to require chunk segmentation in the first place.

## 5.1 Material

Text data was taken from a German translation of 'Madame Bovary' [Flaubert, 1857], which is publicly available from the Gutenberg project[1]. The audio data is taken from a corresponding audio book, read by a male German speaker, which is publicly available from the libriVox project[2]. The audio book comes in chapters of durations up to 56 minutes, amounting to a total of over 14 hours of speech. This means that segmentation with MAUS on its own could be expected to take several days or even weeks. In fact, an attempt to perform a MAUS-only segmentation on the 56 minute chapter ended with an error after two days.

## 5.2 Experiment

The e-book text was split into chapters, which were then inputted into the grapheme to phoneme converter G2P [Reichel and Kisler, 2014], to receive a German canonical transcription in form of a BPF file. The audio book chapters were downloaded as mp3 files, converted to WAVE format and downsampled to 16000 Hz using [SoX, 2015]. The audio-BPF pairs were inputted into all three chunkers with the parameters described in Table

[1] <www.gutenberg.org/ebooks/15711> (last accessed 02/07/2016)

[2] <www.librivox.org/frau-bovary-by-gustave-flaubert> (last accessed 02/07/2016)

4.2. The only exception was that the **-silenceonly** parameter was set to 50, meaning that chunk boundaries were limited to areas where HVite had found a pause of duration above 50 ms. This was done as the read material contained plenty of pauses, and because targeting these pauses could potentially increase the number of zero-errors. The resulting TRN-tier-augmented BPF files were inputted into MAUS along with their corresponding WAVE files, with the **USETRN** option set to **yes**. All programs were run on the same server used in Section 4.3.

## 5.3 Evaluation

### 5.3.1 Performance

Runtime was evaluated for the 'Madame Bovary' material as a whole, by adding up the durations of the individual calls to G2P[3], the chunker and MAUS. Figure 5.1 therefore shows the time necessary to get from the raw audio-text collection to a phonetic segmentation, when using one of the three chunker algorithms and MAUS.

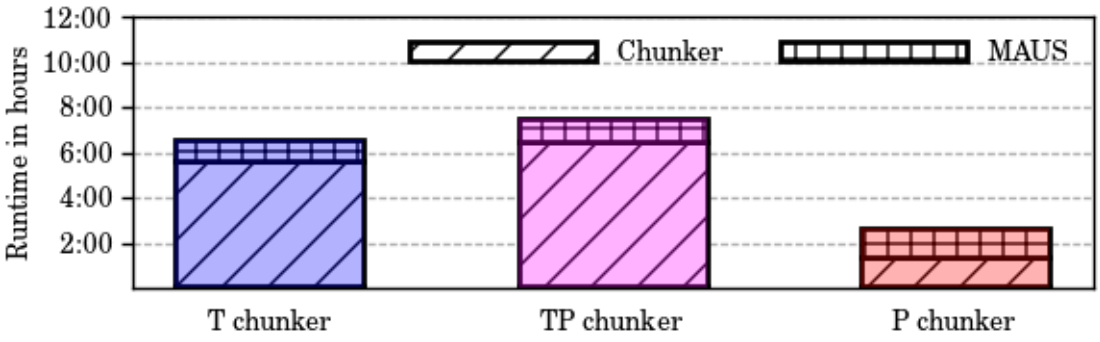



Figure 5.1: Segmentation runtime on full 'Frau Bovary' collection on three threads

### 5.3.2 Token accessibility

Token accessibility was evaluated as described in Section 4.1.2. In the case of the T and TP chunkers, the maximum chunk duration for the whole collection was below one minute. In the case of the P chunker, all tokens ended up being contained in chunks of duration below 4 minutes (see Figure 5.2). This improvement over the token accessibility achieved on the clean Verbmobil data is probably due to the fact that this was read instead of spontaneous speech.

[3]Note that total G2P runtime was just over 2 minutes, which is why it does not show in the bar chart.

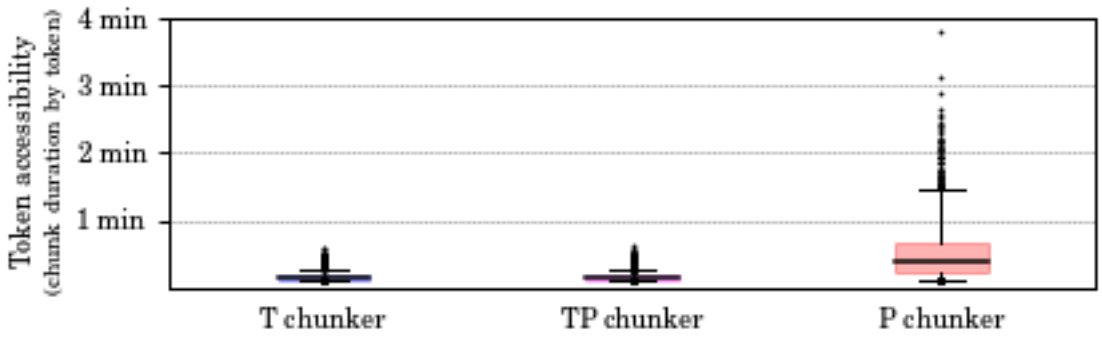



Figure 5.2: Token accessibility (chunk duration by token) on 'Frau Bovary' material (with **-silenceonly 50**)

### 5.3.3 Chunk boundary errors

Since no ground truth segmentation is available for the 'Frau Bovary' material, chunk boundary errors could not be formally evaluated. However, an informal manual check was performed on the first chapter using the emu webApp [Winkelmann and Raess, 2014]. It revealed promising results, with no grave errors appearing in any of the three chunk segmentations.

### 5.3.4 Placing boundaries in pauses

Finally, the chunker's effectiveness at placing chunk boundaries in inter-word pauses was evaluated. Since there was no ground truth segmentation, this could not be done directly. Instead, the auxiliary hypothesis was made that pauses often coincide with certain punctuation marks (period, comma, question mark, exclamation mark, colon and semicolon) in read speech. Therefore, these were taken as ground truth pause indicators. The ratio of the number of pause punctuation marks to the number of tokens in the full e-book was 17.4 %. This suggests that, with chunk boundary placement at random word boundaries, 17.4 % of chunk boundaries would be expected to coincide with a pause punctuation mark.

First, the audio book segmentation was re-run with the **-silenceonly** option set to **0**, in order to be able to compare the results to those already obtained for a **-silenceonly** value of **50**. In a next step, the frequency of chunk boundaries coinciding with pause punctuation marks was calculated for both conditions and all three chunker algorithms. This was done by finding the contexts of all chunk boundaries in the original text with a regular expression, and checking whether the thus located inter-word space contained a period, comma, question mark, exclamation mark, semicolon or colon. See Table 5.1 for the results.

| Boundary type | With punctuation | Total | Percentage with punctuation |
|---|---|---|---|
| Word | 20008 | 114809 | 17.4 % |
| **-silenceonly 0** | | | |
| Chunk (T) | 3960 | 5740 | 69.0 % |
| Chunk (TP) | 3984 | 5769 | 69.1 % |
| Chunk (P) | 981 | 3748 | 26.2 % |
| **-silenceonly 50** | | | |
| Chunk (T) | 4722 | 4941 | 95.6 % |
| Chunk (TP) | 4827 | 5026 | 96.0 % |
| Chunk (P) | 1416 | 1562 | 90.7 % |

Table 5.1: Frequency of boundaries coinciding with pause punctuation marks, at different values of the **-silenceonly** parameter

Assuming that the auxiliary hypothesis formulated above is correct, Table 5.1 suggests that the chunker is more effective at placing chunk boundaries in inter-word pauses than it would be expected to be with random placement, even when the **-silenceonly** parameter is set to **0**. The effect becomes stronger when **-silenceonly** is set to **50**, with ratios between 90.7 % and 96 % of chunk boundaries coinciding with pause punctuation marks, depending on the algorithm.

It is however worth noting that the total number of discovered chunk boundaries falls from the **-silenceonly 0** condition to the **-silenceonly 50** condition for all three chunker algorithms, with the decrease being a lot more dramatic in the P chunker than in the T and TP chunkers. This may be one of the reasons why the P chunker is less effective at finding short chunks than the other two algorithms in Figure 5.2.

## 5.4 Summary

To summarize, the chunker makes it possible to perform phonetic MAUS-segmentation on the chapters of the 14-hour 'Frau Bovary' audio book within less than real-time. All three algorithms manage to place all material in chunks below four minutes, with most material being contained in chunks of duration under one minute. When the **-silenceonly** option is set to **50**, all three algorithms are also very successful at placing chunk boundaries in places where they correspond to pause punctuation marks in the e-book. All in all, this suggests that the chunker is well-suited for the segmentation of e-book / audio book pairs.

# Chapter 6

# Discussion

## 6.1 Opportunities

In Chapter 1, it was argued that many types of non-scientific transcription-recording pairs are difficult to segment with MAUS, as their durations often exceed the limits of what MAUS can process in a reasonable amount of time. The audio book used in Chapter 5 is a good example of this: While the material is a forced aligner's dream in many respects – it is studio quality, clearly articulated, slow speech – much of it would simply be inaccessible to MAUS, and therefore to many types of phonetic analysis, without manual presegmentation.

Chunker preprocessing has the potential for changing this situation. It extends the possibility of real-time, fully automatic MAUS segmentation to recordings of duration up to three hours, with the potential for pushing this limit even further with additional multithreading. Therefore, it is hoped that the chunker will make a contribution towards the inclusion of a wider range of recording types into phonetic research.

## 6.2 Limitations

In Section 4.2, we have seen that, with one exception, the effect of chunker preprocessing on MAUS's word segmentation accuracy is neutral or positive even in the face of noisy recordings and faulty transcriptions. Therefore, there is little reason to advise against its use at this point. It is however worth pointing out that while the chunker mostly did not decrease MAUS's segmentation accuracy, it also did not solve the problem posed by corrupted and noisy material. Therefore, users should probably be advised that, concerns over audio duration aside, they should not input data into the chunker that they would not have inputted into MAUS in the first place.

## 6.3 Comparison with other segmentation tools

When comparing the word segmentation error distributions presented in Section 4.2 with the alignment accuracy scores reported by [Moreno et al., 1998], [Katsamanis et al., 2011] and [Bordel et al., 2012] (see Section 2.1), the combination of the chunker and MAUS appears to produce competitive results on long recordings. Of course, the comparison is not reliable, as experiments were conducted on different languages, different corpora, and with different kinds of signal and transcription manipulations on the material. In the case of [Katsamanis et al., 2011]'s SailAlign, a direct comparison may be possible in the future, as the tool is publicly available[1]. For this purpose, the chunker will have to be extended to English first.

## 6.4 Comparison of the chunker algorithms

As we have seen in Section 4.3, the P chunker is faster than the T and TP chunkers, with an especially big margin on few threads. However, it is mostly less successful than the others at returning short chunks, at placing chunk boundaries in inter-word pauses, and at improving MAUS's segmentation accuracy.

While the T chunker is slower than the P chunker, the difference becomes smaller as the number of threads grows. In most conditions, it is the most successful algorithm at returning short chunks and at improving MAUS's segmentation accuracy. It also has a good record of placing boundaries in inter-word pauses.

The TP chunker usually performs similar to the T chunker, with slightly worse runtime performance. One selling point, opposed to the T and P chunkers, would be that its chunk boundary error outliers were found to be a lot smaller in the cross-talk scenario, suggesting that the TP chunker is less prone to grave misalignments on noisy audio material. Apart from that however, it appears that combining term-based recognition with the granularity of phone-based symbolic alignment did not pay off.

To conclude, it appears that the T chunker is the strongest candidate among the three chunker algorithms. In cases where runtime is an issue, and multithreading is not possible, it might be preferable to use the P chunker, especially if the material is clean enough to expect many chunk boundaries.

[1] <www.github.com/nassosoassos/sail_align> (last accessed 06.07.16)

## 6.5 Possible future developments

### 6.5.1 Integration into the BAS web services

The chunker is due to be integrated into the BAS web services. However, there are a number of practicality issues that need to be resolved before then.

#### The upload problem

At present, users wishing to use the chunker and then MAUS as a BAS web service would have to upload their audio file twice. This is inefficient, as the exact same audio file is used in both cases. Potential solutions would include offering a wrapper service chaining both services together, although this would run counter to the BAS's philosophy of modular services. Alternatively, it might be possible to implement a scheme where users can opt to upload their file to the server for a limited period of time, be given an access code in return, and then use that access code as an input to the chunker and then to MAUS.

#### The initial and final pause problem

At present, MAUS has two options for handling initial and final pauses, which are governed by the option **NOINITIALFINALSILENCE**. If the option is set to **no**, MAUS assumes that there is a mandatory pause of at least 30 ms at the beginning and end of every chunk. If it is set to **yes**, initial and final pauses are forbidden, meaning that the HTK lattice forces the first and last segments of the segmentation to touch the edges of the chunk.

This is a problem when working on chunker output, especially when the **-silenceonly** option is set to **0**. In this case, the chunker does not guarantee a homogeneous output with respect to initial and final pauses: There may be chunks that start or end with a pause, as well as others that start or end right in the middle of a phrase.

In the first case, setting **NOINITIALFINALSILENCE** to **yes** results in the first and last segments being incorrectly drawn into the pause. In the second case, setting the option to **no** results in the introduction of a forced pause where there is none. (In the evaluation experiments, the option was set to **no** throughout.)

The problem could be solved by introducing a third option to make initial and final silences optional, just as they are in the n-gram models used by the chunker. An informal test suggested that placing optional pauses at the beginning and end of MAUS lattices is feasible and produces the expected results.

#### Parameters

Additionally, it has to be decided which of the chunker input parameters (see Appendix A) will be set internally at the BAS, and which will be passed on to the web service interface. Especially, there is a conflict between the wish to give users as much control as possible, and the danger of confusing those without sufficient insight into the chunker's method. A compromise solution would be setting default values that can be accepted by novice users and changed by experts. Potential user interface parameters include:

- **-minchunkduration**, default value **6**
- **-maxchunkduration**, default value **0** (meaning no maximum chunk duration)
- **-chunkertype**, default value **t**
- **-silenceonly**, default value **0**
- **-minanchorsingletons**, default value **1**
- **-minanchorlength**, default value according to the chunker type (see Table 4.2)
- **-maxanchorcost**, default value according to the chunker type (see Table 4.2)

#### The no output problem

As we have seen in Section 4.1, there may be cases, especially in the presence of cross-talk noise, where the chunker fails to find any chunk boundaries. As this leads to long MAUS runtimes, it will be necessary to warn users that this has happened, and to advise them of their options. The BAS web service interface allows to print error messages and warnings to a messaging window in the browser. Apart from a simple warning, there is also the possibility of including the closest **-minanchorlength** parameter that would have led to a better chunk segmentation, along with a warning that low values on this parameter may increase error rates.

### 6.5.2 Combining the chunker algorithms

Recall from Section 6.4 that the P chunker is faster than the T chunker, while the T chunker is more effective at finding short chunks and at targeting inter-word pauses. Instead of simply weighing these advantages against one another, it might be possible to combine them. Recall from Section 3.8.1 that the T chunker spends most of its runtime on the topmost chunker in the recursion tree, while child chunkers lower down are relatively fast. Consequently, it might be advantageous to run that topmost chunker in P chunker mode,

possibly with a relatively high **-minchunkduration** value, before running the child chunkers in T chunker mode. Ideally, this should give us the best of both worlds: the fast algorithm where speed is important, and the thorough algorithm at the later stages.

It would, of course, be necessary to take precautions against the case where the initial P chunker does not find any chunk boundaries at all. One solution might be to try and redo the initial segmentation with the T chunker in this case.

### 6.5.3 Factor automaton language model

At present, the chunker uses n-gram language or phonotactic models for recognition. An alternative would be factor automaton language models which, as described in Section 2.2.2, accept all substrings of the target transcription. Therefore, they are more restrictive than n-gram models, and thus more likely to return a usable recognition result, which in turn might lead to an easier symbolic alignment. Preliminary tests suggested that substituting the n-gram models for factor automata increases HVite runtime, meaning that multithreading would become even more important. What is more, factor automata are likely to be risky when there are longer untranscribed audio stretches, like in the turn omission scenario described in Section 4.1.1. Nonetheless, the addition might prove beneficial, and would likely be possible with relatively few modifications to the chunker's source code.

### 6.5.4 MAUS knowledge for the chunker

At present, the chunker does not have the ability to model pronunciation variation in the way that MAUS does. Instead, it assumes that a term's acoustic realization will resemble its canonical pronunciation. In Section 1.1.1, it was claimed that the recognition-based segmentation approach used by the chunker does not support this feature. However, this is not completely true. The T chunker could, in principle, be extended to model pronunciation variation within terms (see Appendix B).

There are two potential benefits of this step: Firstly, it might improve recognition accuracy, as HVite would be better able to recognize non-canonical realizations of terms. More accurate recognition results would likely make symbolic alignment, and thus the search for anchors and ultimately chunk boundaries, more successful. Secondly, it would even be conceivable to implement a 'Fast MAUS' mode that directly returns a full MAUS-style phonetic segmentation instead of a chunk segmentation. See Appendix B for a detailed outline of that plan.

# Appendix A

# Chunker Manual

## A.1 Requirements

The chunker is implemented using C++ and the 2011 C++ standard library. It requires access to the following HTK tools as executables:

- HVite
- HCopy
- HList

For details on HTK, see [Young et al., 2006].

## A.2 Options and parameters

**-v, -verbose (0|1|2|3)**

Controls how much information is printed.

- **0** = no output
- **1** = summary
- **2** = summary and recursion tree
- **3** = summary and detailed recursion tree

### -t, -type, -chunkertype (t|tp|p)

Type of algorithm used.

- **t** = term-based recognition and token-by-token symbolic alignment
- **tp** = term-based recognition and phone-by-phone symbolic alignment
- **p** = phoneme-based recognition and phone-by-phone symbolic alignment

### -lm, -languagemodel (0|1|2)

Degree of n-gram model to be used for the HTK recognition lattice.

- **0** = loop without weights
- **1** = unigram language model
- **2** = smoothed bigram language model

### -lmweight, -languagemodelweight

Weight of the language model relative to the acoustic models.

### -bigramweight

Smoothing parameter (weight of bigram language model relative to the unigram language model). Only useful when **-languagemodel** is set to **2**. Should be a value between **0** (no weight for bigram probabilities) and **1** (full weight for bigram probabilities).

### -r, -recursion (0|1)

Enables recursive application of the algorithm to chunks found in previous iterations. Highly recommended.

### -maxrecursiondepth

Recursion depth at which recursion is disabled.

**-minchunkduration, -minduration**

Minimum chunk duration in seconds. Potential chunk boundaries that would lead to chunks below this threshold are rejected.

**-maxchunkduration, -maxduration**

Maximum chunk duration in seconds.

- **0** = all discovered chunks are written to the output TRN tier. This guarantees that all transcription tokens end up in a chunk.
- greater than **0** = only chunks with a duration below the threshold are written to the output TRN tier. This means that some tokens may not be included in any chunk, leading to MAUS ignoring them.

**-silenceonly**

- **0** = prioritize inter-word pauses, but also consider word boundaries without an intervening pause as chunk boundaries.
- greater than **0** = place chunk boundaries only in inter-word pauses with duration (in ms) above this threshold.

Set to **0** for audio files that do not contain many pauses, if audio quality is poor (e.g. in the presence of cross-talk), or if there are other reasons for assuming that the chunker will struggle to find boundaries.

**-minanchorlength**

Minimum length of anchors on the alignment path in token (T chunker) or phone (TP and P chunkers) alignment operations. Setting this to a high value leads to conservative choices by the chunker, while increasing the risk of not finding the desired number of chunk boundaries.

**-maxanchorcost**

Maximum summed Levenshtein edit costs of anchors on the alignment path. Setting this to a low value leads to conservative choices by the chunker, while increasing the risk of not finding the desired number of chunk boundaries. The value of **-maxanchorcost** should always be smaller than the value of **-minanchorlength**.

**-minanchorsingletons**

Minimum number of singleton tokens in anchors on the alignment path. Setting this to a high value leads to conservative choices by the chunker, while increasing the risk of not finding the desired number of chunk boundaries. The value of **-minanchorsingletons** should always be smaller than or equal to the value of **-minanchorlength**.

**-maxnumthreads**

Maximum number of threads that the chunker may run in parallel. Set to **1** to disable multithreading.

**-splitduration**

Duration, in seconds, of intervals that the signal is split into before being passed to the recognition engine. Small values (below approximately 100 seconds) may increase runtime and lead to errors. Very large values (above audio duration divided by the value of the **-maxnumthreads** parameter) may also increase runtime if multithreading is used.

**-a, -audio, -audiofile**

Input audio file. Must be in WAVE format.

**-b, -bpf, -bpffile**

Input file in BAS Partitur Format. Must at least contain:

- a sample rate entry of the form **SAM: <samplerate>**
- a canonical pronunciation tier whose entries have the form **KAN: <index> <token>**

**-c, -config, -configfile**

Path of configuration file.

**-o, -outfile**

Path of the future output file.

**-outtype (trn-append|bpf)**

Desired output type.

- **trn-append** = isolated TRN tier
- **bpf** = original BPF file with appended TRN tier

**-workdir**

Directory into which the chunker can write its intermediate files.

**-mmffile**

Path of HTK compatible master model file. The labels of the models in this file must match the model names from the second column of the **-labelmap** file.

**-labelmap**

File containing a two-column table mapping the phonemes found in the input KAN tier onto the corresponding acoustic models in the master model file.

**-exceptionfile**

File containing a newline-separated list of all symbols that the chunker should ignore when parsing the transcription.

**-hvitebeamwidth**

Beam width to be used by the recognition engine.

**-aligner (matrix|hirschberg)**

Algorithm for symbolic alignment.

- **matrix** = Wagner-Fischer algorithm
- **hirschberg** = Hirschberg algorithm (recommended for long segmentation tasks)

# Appendix B

# MAUS knowledge for the chunker – an outline

The following is a technical outline of a plan for enhancing the T chunker with MAUS-style knowledge about pronunciation variation within tokens[1]. This would have two potential benefits: Firstly, knowledge about non-canonical pronunciation might increase HVite's recognition accuracy within the chunker, which might then lead to an easier symbolic alignment and thus to more chunk boundaries. Secondly, it would also be conceivable to implement a 'Fast MAUS' mode where the chunker produces a MAUS-style phonetic segmentation instead of an intermediate chunk segmentation, thereby making it unnecessary to invoke MAUS afterwards.

## B.1 Implementation outline

At present, the HTK lattices constructed by the T chunker represent terms as atomic units, which are mapped onto linear sequences of acoustic models via the HTK dictionary file. MAUS on the other hand models terms as directed, forking graphs of phonemes (see Figure B.1 for an example), with weighted arcs to model the probabilities of different pronunciation variants. The graphs, which are stored in the same HTK lattice format that the chunker uses, are created by a module called word_var [Kipp et al., 1997], which is independent of MAUS. It is conceivable to replace the chunker's atomic terms with the lattices outputted by word_var (see Figure B.2)[2].

[1] While MAUS is in principle able to model variation that depends on processes across token boundaries, an implementation based on the T chunker would be confined to processes within tokens. However, for many languages (e.g. French and British English), MAUS does not model inter-token processes either.

[2] The T chunker's HTK dictionary would need to be replaced with a two-column table where every phoneme from the word_var lattices is mapped onto an acoustic model, similar to the way that MAUS does this.

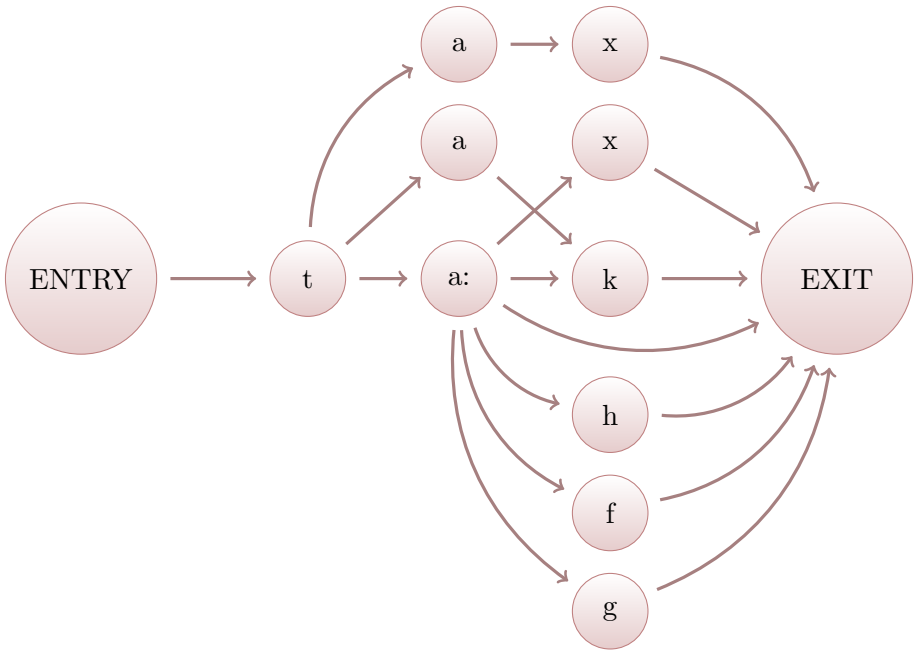



Figure B.1: word_var lattice for the German term /ta:k/

After these changes, the recognition engine would output a string of phonemes instead of a string of tokens. Within that string, there would be no way of telling where tokens begin and end, or which tokens are realizations of which terms. These two problems obviously need to be solved if we want to perform token-by-token symbolic alignment.

The first problem – locating the beginning and end of tokens – can be solved in the way that MAUS does. MAUS precedes every token in its transcription with an optional inter-word pause model to absorb possible silences between tokens. This inter-word pause model is mapped onto the acoustic model <p:>. The acoustic model can be skipped by an internal 'jump-over' arc, meaning that it can be traversed in zero milliseconds. However, the model still surfaces in the label file, even when its duration is zero. This is why MAUS output can be segmented into tokens by simply cutting at the pauses.

Within the chunker lattices, a similar construction is already in place (see Figures 3.1, 3.2 and 3.3). However, this construction does not guarantee that pauses surface in the label file, as they may also be skipped entirely by an external 'jump-over' arc. If the external 'jump-over' arcs were deleted, optional pauses would always be printed to the label file, just as they are in the case of MAUS. Consequently, the function parsing the label file would be able to tell where tokens begin and end.

The second issue remains however. Even if we can segment the string into tokens, there is no a-priori way of telling, from the tokens' phoneme sequences alone, which terms the tokens represent. In MAUS, the solution is trivial, as the intervals between pauses can be mapped onto the transcription tokens in a one-on-one fashion, as both sequences are

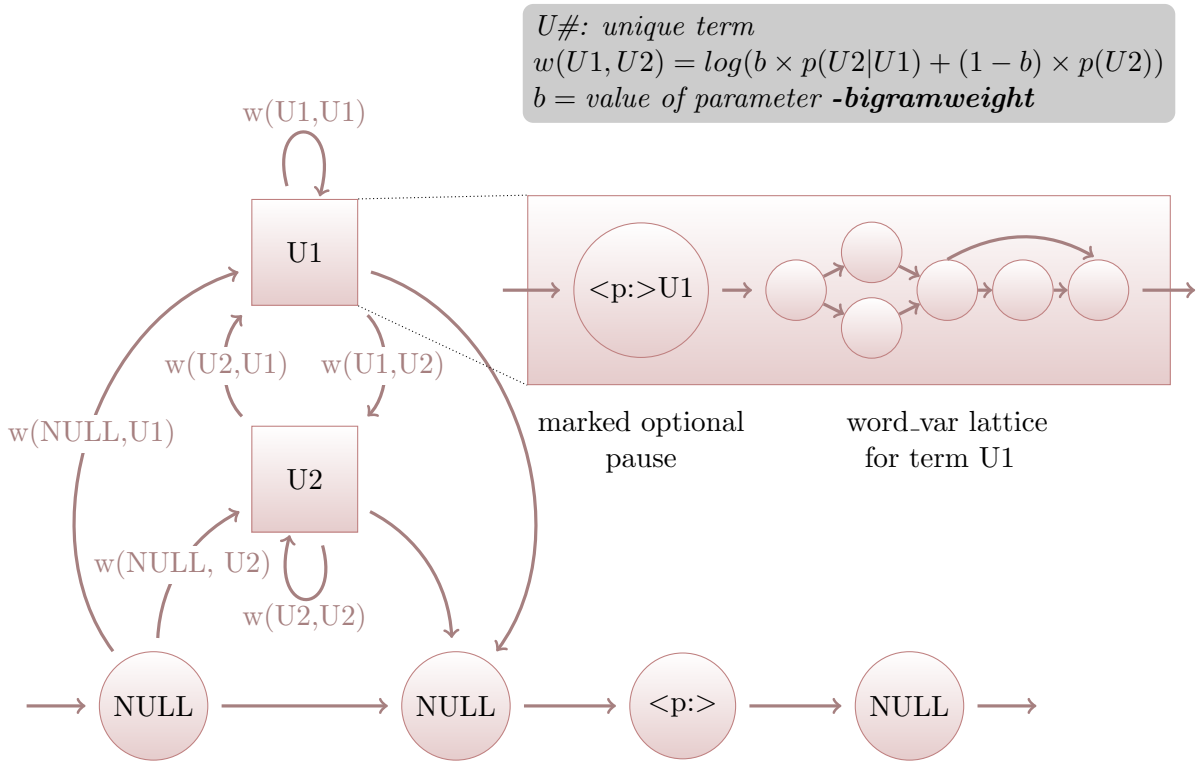



Figure B.2: Smoothed bigram lattice with word_var pronunciation models as terms, and marked optional pauses

ordered identically. In our case on the other hand, this is obviously impossible. There is a solution, however, which also heavily depends on the optional inter-word pause models described above.

Recall that, at present, every pre-term optional pause in the chunker's smoothed bigram lattice has a single outgoing arc to a specific term[3]. The pause before the term /ta:k/, for instance, could therefore be characterized as /ta:k/'s own personal pause. If the personal pause's label contained a 'marker' referring to /ta:k/ (resulting in a label such as <p:>ta:k), then that marker would also appear in the label file. Thus, we would know that any series of phonemes recognized after <p:>ta:k, up until the next pause, is a realization of /ta:k/[4] (see Figure B.3).

Internally, the recognition result could therefore be represented as a sequence of tokens, in canonical pronunciation, according to the markers found on the optional pauses. Each token would in turn be associated with the non-canonical sequence of phonemes outputted by HVite after that marked pause, with associated start and end times. Token-by-token alignment as well as the search for candidate anchors could then be performed as described

[3] This is not true for the zerogram and unigram lattices, but this can be easily fixed.

[4] Note that the number of acoustic models is not increased by this measure, as the marked optional pauses can simply be mapped onto the generic optional pause acoustic model <p:> via the dictionary file.

```
0          1600000   <p:>gu:t@n    # marked optional pause with non-zero duration
1600000    2600000   g
2600000    3400000   u:            # non-canonical pronunciation of /gu:t@n/
3400000    4500000   t
4500000    5400000   @
5400000    5400000   <p:>ta:k      # marked optional pause with zero duration
5400000    6300000   t
6300000    7200000   a:
```

Figure B.3: Label file excerpt from HVite recognition test with word_var-enhanced lattice

in Sections 3.6 and 3.7. This would be possible because the canonical pronunciation tokens from the transcription are comparable to the canonical pronunciation tokens found in the pause markers of the recognition result. Once the anchors have been found, there are two possibilities. If word_var is simply used to enhance the chunker's ability to recognize non-canonical pronunciations, then the chunk boundary selection phase continues as described in Section 3.7.

If, on the other hand, the goal is a MAUS-style phonetic segmentation ('Fast MAUS' mode), then the internal non-canonical segmentation would not be used for chunk boundary selection and then forgotten, but instead remembered for the final result. In this case, recursion would, like in [Moreno et al., 1998] and [Katsamanis et al., 2011], only apply to segments that are not part of an anchor. Of course, this kind of algorithm requires a plan for cases in which areas remain unaligned even after recursive application of the algorithm. A combination of [Moreno et al., 1998]'s and [Katsamanis et al., 2011]'s solutions is conceivable, where the criteria for anchor selection are loosened at later iterations, with a final MAUS segmentation on any stretches that are not yet aligned at the end.

## B.2 Expected performance

Like the T chunker, the 'Fast MAUS' mode's lattice size would depend on the number of unique terms instead of the number of tokens in the transcription. This in turn means that HVite should run a lot faster on them than on comparable MAUS lattices (see Section 3.5.1). What is more, the recognition task would be spliceable[5] and therefore parallelizable. Thus, 'Fast MAUS' performance should be comparable to that of the T chunker and not to that of MAUS on long transcribed recordings.

[5] In this case, voice activity detection, as used by [Katsamanis et al., 2011], would probably be necessary, as cutting inside words is problematic when the goal is an actual phonetic segmentation.

# Bibliography


[Anguera et al., 2011] Anguera, X., Perez, N., Urruela, A., and Oliver, N. (2011). Automatic synchronization of electronic and audio books via TTS alignment and silence filtering. In *2011 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE.

[Bordel et al., 2012] Bordel, G., Peagarikano, M., Rodrguez-Fuentes, L. J., and Varona, A. (2012). A simple and efficient method to align very long speech signals to acoustically imperfect transcriptions. In *INTERSPEECH*, pages 1840–1843.

[Burger et al., 2000] Burger, S., Weilhammer, K., Schiel, F., and Tillmann, H. G. (2000). Verbmobil data collection and annotation. In *Verbmobil: Foundations of Speech-to-Speech Translation*, pages 537–549. Springer.

[Fink, 2013] Fink, G. A. (2013). *Mustererkennung mit Markov-Modellen: Theorie - Praxis - Anwendungsgebiete*. Springer-Verlag.

[Flaubert, 1857] Flaubert, G. (1857). *Frau Bovary (translation by Arthur Schurig)*.

[Harrington, 2010] Harrington, J. (2010). *Phonetic analysis of speech corpora*. John Wiley & Sons.

[Hirschberg, 1975] Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343.

[Huang et al., 2003] Huang, C.-W., Hsu, W., and Chang, S.-F. (2003). Automatic closed caption alignment based on speech recognition transcripts. *Rapport technique, Columbia*.

[Jang and Hauptmann, 1999] Jang, P. J. and Hauptmann, A. G. (1999). Improving acoustic models with captioned multimedia speech. In *Multimedia Computing and Systems, 1999. IEEE International Conference on*, volume 2, pages 767–771. IEEE.

[Jones et al., 2001] Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. (v. 0.13.03).

[Katsamanis et al., 2011] Katsamanis, A., Black, M., Georgiou, P. G., Goldstein, L., and Narayanan, S. (2011). SailAlign: Robust long speech-text alignment. In *Proc. of Workshop on New Tools and Methods for Very-Large Scale Phonetics Research*.

[Kipp et al., 1997] Kipp, A., Wesenick, M.-B., and Schiel, F. (1997). Pronuncation modeling applied to automatic segmentation of spontaneous speech. In *EUROSPEECH*.

[Kisler et al., 2016] Kisler, T., Reichel, U. D., Schiel, F., Draxler, C., Jackl, B., and Poerner, N. (2016). BAS speech science web services – an update on current developments. In *LREC*.

[Kisler et al., 2015] Kisler, T., Schiel, F., Reichel, U. D., and Draxler, C. (2015). Phonetic/linguistic web services at BAS. In *INTERSPEECH*.

[Knight and Almeroth, 2012] Knight, A. and Almeroth, K. (2012). Fast caption alignment for automatic indexing of audio. In *Methods and Innovations for Multimedia Database Content Management*, pages 204–220. IGI Global.

[Lanchantin et al., 2015] Lanchantin, P., Gales, M., Karanasou, P., Liu, X., Qian, Y., Wang, L., Woodland, P., and Zhang, C. (2015). The development of the Cambridge University alignment systems for the multi-genre broadcast challenge. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 647–653. IEEE.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[Martin and Jurafsky, 2000] Martin, J. H. and Jurafsky, D. (2000). *Speech and Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson.

[Martone et al., 2003] Martone, A. F., Taskiran, C. M., and Delp, E. J. (2003). Automated closed-captioning using text alignment. In Yeung, M. M., Lienhart, R. W., and Li, C.-S., editors, *Storage and Retrieval Methods and Applications for Multimedia*, pages 108–116.

[Moreno and Alberti, 2009] Moreno, P. J. and Alberti, C. (2009). A factor automaton approach for the forced alignment of long speech recordings. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4869–4872. IEEE.

[Moreno and Alberti, 2012] Moreno, P. J. and Alberti, C. (2012). *Aligning a transcript to audio data*. Google Patents.

[Moreno et al., 1998] Moreno, P. J., Joerg, C. F., Van Thong, J.-M., and Glickman, O. (1998). A recursive algorithm for the forced alignment of very long audio segments. In *ICSLP*, volume 98, pages 2711–2714.

[Reichel and Kisler, 2014] Reichel, U. D. and Kisler, T. (2014). Language-independent grapheme-phoneme conversion and word stress assignment as a web service. *Elektronische Sprachverarbeitung. Studientexte zur Sprachkommunikation*, 71:42–49.

[Schiel, 1999] Schiel, F. (1999). Automatic phonetic transcription of non-prompted speech. In *Proc. of the ICPhS*, pages 607–610.

[Schiel, 2015] Schiel, F. (2015). A statistical model for predicting pronunciation. In *Proc. of the ICPhS*.

[Schiel et al., 1998] Schiel, F., Burger, S., Geumann, A., and Weilhammer, K. (1998). The partitur format at BAS. In *Proceedings of the First International Conference on Language Resources and Evaluation*, pages 1295–1301.

[Schiel et al., 2011] Schiel, F., Draxler, C., and Harrington, J. (2011). Phonemic segmentation and labelling using the MAUS technique. In *Workshop New Tools and Methods for Very-Large-Scale Phonetics Research*.

[SoX, 2015] SoX (2015). SoX - Sound eXchange (v. 14.4.1). `www.sox.sourceforge.net`.

[Viterbi, 1967] Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269.

[Wagner and Fischer, 1974] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

[Winkelmann and Raess, 2014] Winkelmann, R. and Raess, G. (2014). Introducing a web application for labeling, visualizing speech and correcting derived speech signals. In *LREC*, pages 4129–4133.

[Young et al., 2006] Young, S., Evermann, G., Gales, M., Hain, T., Kershaw, D., Liu, X., Moore, G., Odell, J., Ollason, D., and Povey, D. (2006). *The HTK book*. Cambridge University.