VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FINITE-STATE BASED RECOGNITION NETWORKS FOR FORWARD-BACKWARD SPEECH DECODING

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Dipl.-Ing. MIRKO HANNEMANN
AUTHOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FINITE-STATE BASED RECOGNITION NETWORKS FOR FORWARD-BACKWARD SPEECH DECODING

ROZPOZNÁVACÍ SÍTĚ ZALOŽENÉ NA KONEČNÝCH STAVOVÝCH PŘEVODNÍCÍCH PRO DOPŘEDNÉ A ZPĚTNÉ DEKÓDOVÁNÍ V ROZPOZNÁVÁNÍ ŘEČI

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Dipl.-Ing. MIRKO HANNEMANN

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. LUKÁŠ BURGET, Ph.D.

BRNO 2016

Abstract

Many tasks can be formulated in the mathematical framework of weighted finite state transducers (WFST). This is also the case for automatic speech recognition (ASR). Nowadays, ASR makes extensive use of composed probabilistic models – called decoding graphs or recognition networks. They are constructed from the individual components via WFST operations like composition. Each component is a probabilistic knowledge source that constrains the search for the best path through the composed graph – called decoding. The usage of a coherent framework guarantees, that the resulting automata will be optimal in a well-defined sense. WFSTs can be optimized with the help of determinization and minimization in a given semi-ring. The application of these algorithms results in the optimal structure for search and the optimal distribution of weights is achieved by applying a weight pushing algorithm. The goal of this thesis is to further develop the recipes and algorithms for the construction of optimal recognition networks. We introduce an alternative weight pushing algorithm, that is suitable for an important class of models – language model transducers, or more generally cyclic WFSTs and WFSTs with failure (back-off) transitions. We also present a recipe to construct recognition networks, which are suitable for decoding backwards in time, and which, at the same time, are guaranteed to give exactly the same probabilities as the forward recognition network. For that purpose, we develop an algorithm for exact reversal of back-off language models and their corresponding language model transducers. We apply these backward recognition networks in an optimization technique: In a static network decoder, we use it for a two-pass decoding setup (forward search and backward search). This approach is called tracked decoding and allows to incorporate the first pass decoding into the second pass decoding by tracking hypotheses from the first pass lattice. This technique results in significant speed-ups, since it allows to decode with a variable beam width, which is most of the time much smaller than the baseline beam. We also show that it is possible to apply the algorithms in a dynamic network decoder by using the incrementally refining recognition setup. This additionally leads to a partial parallelization of the decoding.

Keywords

Automatic speech recognition, LVCSR decoding, recognition networks, weighted finite state transducers, N-gram language models, weight pushing

Bibliographic citation

Mirko Hannemann: Finite-state based recognition networks for forward-backward speech decoding, PhD thesis, Brno, Brno University of Technology, Faculty of Information Technology, 2016

Abstrakt

Pomocí matematického formalismu váhovaných konečných stavových převodníků (weighted finite state transducers WFST) může být formulována řada úloh včetně automatického rozpoznávání řeči (automatic speech recognition ASR). Dnešní ASR systémy široce využívají složených pravděpodobnostních modelů nazývaných dekódovací grafy nebo rozpoznávací sítě. Ty jsou z jednotlivých komponent konstruovány pomocí WFST operací, např. kompozice. Každá komponenta je zde zdrojem znalostí a omezuje vyhledávání nejlepší cesty ve složeném grafu v operaci zvané dekódování. Využití koherentního teoretického rámce garantuje, že výsledná struktura bude optimální podle definovaného kritéria. WFST mohou být v rámci daného polookruhu (semi-ring) optimalizovány pomocí determinizace a minimalizace. Aplikací těchto algoritmů získáme optimální strukturu pro prohledávání, optimální distribuce vah je pak získána aplikací “weight pushing” algoritmu. Cílem této práce je zdokonalit postupy a algoritmy pro konstrukci optimálních rozpoznávacích sítí. Zavádíme alternativní weight pushing algoritmus, který je vhodný pro důležitou třídu modelů - převodníky jazykového modelu (language model transducers) a obecně pro všechny cyklické WFST a WFST se záložními (back-off) přechody. Představujeme také způsob konstrukce rozpoznávací sítě vhodné pro dekódování zpětně v čase, které prokazatelně produkuje ty samé pravděpodobnosti jako dopředná síť. K tomuto účelu jsme vyvinuli algoritmus pro exaktní reverzi back-off jazykových modelů a převodníků, které je reprezentují. Pomocí zpětných rozpoznávacích sítí optimalizujeme dekódování: ve statickém dekodéru je využíváme pro dvoustupňové dekódování (dopředné a zpětné vyhledávání). Tento přístup — “sledovací” dekódování (tracked decoding) — umožnuje zahrnout výsledky vyhledávání z prvního stupně do druhého stupně tak, že se sledují hypotézy obsažené v rozpoznávacím grafu (lattice) prvního stupně. Výsledkem je podstatné zrychlení dekódování, protože tato technika umožnuje prohledávat s variabilním prohledávacím paprskem (search beam) – ten je povětšinou mnohem užší než u základního přístupu. Ukazujeme rovněž, že uvedenou techniku je možné využít v dynamickém dekodéru tím, že postupně zjemňujeme rozpoznávání. To navíc vede i k částečné paralelizaci dekódování.

Klíčová slova

Automatické rozpoznávání řeči, dekodování řeči, rozpoznávací sítě, váhované konečné stavové automaty, jazykové modely

Bibliografická citace

Mirko Hannemann: Rozpoznávací sítě založené na konečných stavových převodnících pro dopředné a zpětné dekódování v rozpoznávání řeči, disertační práce, Brno, FIT VUT v Brně, 2016

Finite-state based recognition networks for forward-backward speech decoding

Declaration of Originality

I hereby declare that this thesis and the work reported herein was composed by and originated entirely from me. The work has been supervised by Doc. Ing. Lukáš Burget, Ph.D. and Doc. Dr. Ing. Jan Černocký. Information derived from the published and unpublished work of others has been acknowledged in the text and references are given in the list of sources. Some of the used recognition systems were set-up by the members of the BUT Speech@FIT research group or in cooperation with third parties (Microsoft Research, Kaldi team, Johns Hopkins University).

....
Mirko Hannemann
17.07.

Rozpoznávací sítě založené na konečných stavových převodnících pro dopředné a zpětné dekódování v rozpoznávání řeči

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením Doc. Ing. Lukáše Burgeta, Ph.D. a Doc. Dr. Ing. Jana Černockého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal. Některé systémy použité v práci byly vytvořeny členy výzkumné skupiny BUT Speech@FIT samostatně nebo ve spolupráci s třetími stranami (Microsoft Research, Kaldi team, Johns Hopkins University).

....
Mirko Hannemann
17.07.

Acknowledgements

I want to thank Lukáš Burget, who was my thesis supervisor, for his feedback and for the discussions, especially for his efforts to advance our general machine learning knowledge. I especially want to thank Daniel Povey, who was my first supervisor at Microsoft Research and who had the original idea of decoding forwards and backwards in time and to perform an iterative weight pushing. I also want to thank him and his family for their hospitality. Thanks go to Honza Černocký for reviewing this thesis and for never being tired in coming up with new projects and funding and to all colleagues from Speech@FIT for their help and good atmosphere at work. I also want to thank Jasha Droppo, who was my second supervisor at MSR, for the discussions on parallelizing the decoding algorithm and reversing the language model. I thank Geoffrey Zweig for hosting me at MSR and for useful discussions on decoding in general. Thanks go to Sanjeev Khudanpur for his input on the weight-pushing algorithm and on language modeling. I want to thank my wife Tereza for the patience she had with the inconveniences caused by the process of writing this thesis, and I want to thank God for providing me with the time and health necessary to achieve it. Some of the work described here was done when the author was summer intern at Microsoft Research, Redmond, WA. This work was partly supported by the Intelligence Advanced Research Projects Activity (IARPA) BABEL program, the European Union's Horizon 2020 project No. 645523 BISON and Czech Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project „IT4Innovations excellence in science – LQ1602“.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation: search graphs and decoding networks

The application I had in mind while writing this thesis was the search for the best path through a composed probabilistic model, represented as weighted finite state acceptor (WFSA) or transducer (WFST). The task can be for example the decoding of the most probable sequence of words in large vocabulary automatic speech recognition (LVCSR). However, the approach presented here can also be used in other tasks, which can be formulated in the WFST framework, as e.g. in finding the most probable sentence in statistical machine translation and finding the most probable pronunciation of a spelled word in grapheme-to-phoneme conversion.

Automatic speech recognition (ASR) can be formulated in the WFST framework [Allauzen et al.(2004)], [Mohri et al.(2008)]. Nowadays, ASR makes extensive use of composed WFSTs, called decoding graphs or recognition networks. WFSTs are used to represent the language model (LM), the pronunciation lexicon and the Hidden Markov Models (HMM) in a unified framework. These component WFSTs are integrated into a single WFST by the composition operation. Each component is a probabilistic knowledge source that constrains the search for the best path through the composed graph. This search is called decoding. The usage of a coherent framework guarantees, that the resulting automata will be optimal in a well defined sense. WFST can be optimized by operations like determinization and minimization in a given semi-ring. The application of these algorithms results in the optimal (deterministic and minimal) structure for search.

An optimized recognition network can contain up to millions of states, and the resulting search state space (trellis) is even several orders of magnitude larger. Given the complexity of most of the tasks, the resulting huge search spaces cannot be explored exhaustively. It is necessary to use heuristic pruning techniques. In this case, we have to distinguish *search errors*, which are due to the incomplete exploration of the search space (e.g. through search beams and other pruning techniques), from *modeling errors*, which are due to insufficient (or bad) training data or due to inaccurate models (independence assumptions, choice of distribution, smoothing, . . .). In general, the goal is to reduce the amount of search errors at given run-time requirements (decoding speed). This can be achieved by operations like weight pushing, which aim to distribute the weights along the path in a way that is optimal for pruned search.

The goal of this thesis is to further develop the recipes and algorithms for the construction of optimal recognition networks. We aim to find the optimal trade-off between improving search speed and reducing search errors.

1.2 Claims of the thesis

The focus of this thesis is the construction of optimal forward and backward recognition networks and the development of decoding techniques that combine forward and backward decoding to achieve speed-ups. We introduce the idea of symmetrically decoding forwards and backwards in time. For some tasks, the pruned backward search can be more efficient than the forward search. Moreover, we show, that the search errors of forward and backward search are mutually independent. To concentrate on search errors rather than on modeling errors, we require both decoding passes to be symmetric – i.e. both models are equally powerful and are constructed to assign exactly the same probabilities to hypotheses. This guarantees that each difference in comparing the results of forward and backward decoding corresponds to a search error. For most of the time frames in beam search decoding, a very narrow beam is sufficient. Therefore, we decode with a variable beam width – using a small baseline beam and only increasing it in places, where the forward and backward searches disagree. Decoding with a variable beam width results in significant speed-ups.

The main contributions of this thesis can be summarized in the following points:

- **Symmetric forward and backward decoding:** To speed-up the decoding, as opposed to multi-pass recognition techniques [Nguyen et al.(1993)], we use forward and backward recognition passes which are equally powerful. Equally powerful forward and backward decoding has been used before for the purpose of system combination [Li et al.(2009)] and confidence estimation [Jouvet and Fohr(2014)]. However, we require that the forward and backward recognition networks assign exactly the same probability scores, which allows us to detect search errors, to recombine partial paths and to incorporate the first pass into the second pass.

- **WFSTs resulting from back-off and interpolated language models:** We show, that the common practice to convert interpolated LMs into back-off LMs, when storing them in the ARPA file format, leads to problems in the construction of the recognition network in the log-probability semi-ring. We give details about the approximation and the correct handling of back-off arcs and explain "missing" N-grams.

- **Alternative weight pushing algorithm:** We give the theoretical justification and explain details of the alternative weight pushing algorithm, that is suitable for an important class of models – language model transducers, or more generally cyclic WFSTs and WFSTs with failure (back-off) transitions.

- **Construction of symmetric backward recognition networks:** We present a recipe to construct recognition networks, which are suitable for decoding backwards in time, fulfill the criteria of determinism and similar size, and which, at the same time, are guaranteed to give exactly the same probabilities as the forward recognition network.

- **Exact back-off language model reversal:** For the purpose of constructing backward recognition networks, we develop an algorithm for exact reversal of back-off language models and their corresponding language model transducers, which is valid for both types of approximations: using epsilon arcs or using failure arcs. We show the derivation of the formulas by a series of steps guaranteeing WFST equivalence, as well as the derivation from Bayes' rule.

- **Tracked decoding and variable beam width:** We develop a two-pass decoding setup (forward search and backward search), that allows to incorporate the first pass decoding into the second pass decoding by tracking hypotheses from the first pass lattice. This technique allows to decode with a variable beam width, which is most of the time much smaller than the smallest single-pass beam at the same word error rate. The beam is only increased in areas, where forward and backward decoding disagree.

- **Speed-up and parallelization:** We have implemented the backward recognition networks for both static and dynamic network decoders and show experiments that demonstrate significant speed-ups in both cases. Applying the incrementally refining recognition setup of [Nolden et al.(2013)] additionally leads to a partial parallelization of the decoding.

1.2.1 Contribution and authorship

My work on the topic of this thesis started at the Kaldi workshop 2010, where I was part of the team implementing a WFST based speech decoder. Several implementation designs were tested, finally the one from Daniel Povey was the simplest and fastest and was used further on as the main Kaldi decoder. The outcome of these efforts is described in the common paper [Povey et al.(2011)]. In the Kaldi workshop 2011, I was part of the team, whose task was to add lattice generation to the Kaldi decoder. Again, several approaches were discussed and implemented, and the final format of the lattices was decided. I conducted the experiments exploring the properties of the new Kaldi lattice generation [Povey et al.(2012)].

While I was at an internship at Microsoft Research (MSR), under supervision of Daniel Povey, we developed the technique of forward-backward decoding. The original idea came up in discussions with Daniel Povey and Geoffrey Zweig. I developed a recipe for the generation of backward decoding networks (section 5.2), together with an initial reversal algorithm for bi-gram language models and I ran a series of experiments exploring the properties of forward and backward decoding and an analysis of the pruning behavior of the Kaldi decoder.

During the internship, my task was to come up with a method for guiding the second pass decoding with the results from the fist pass. For that purpose, I developed the graph-arc lattices and the algorithm to construct them from the Kaldi lattices (section 5.4.3). I also designed the tracked decoding algorithm (section 5.4), implemented it and ran the experimental validation of tracked decoding on a smaller LM.

Afterwards, I implemented the tracked decoding into the Kaldi toolkit and continued the experiments by switching to a larger LM. Together with Daniel Povey, we developed the initial reversal algorithm for WFSTs resulting from ARPA LMs with epsilon arcs. I derived the formulas for this approach in the first part of section 4.4. I also explained the presence of missing N-grams in the LM (section 4.3) and showed how to correctly deal with them in the forward and backward models. This approach for LM reversal was used in a more thorough evaluation of the tracked decoding, which we published in [Hannemann et al.(2013)]. In this thesis, I also analyze the contribution of the different pruning parameters of the technique (section 5.4.5).

Since the standard weight pushing algorithm was failing for higher order LMs, together with Daniel Povey, we discussed several approaches to weight pushing. The final idea of using the matrix power method was suggested by Daniel Povey and Sanjeev Khudanpur and first implemented into the Kaldi toolkit by Ehsan Variani and Pegah Ghahrmani. However, we only mentioned it very briefly in [Hannemann et al.(2013)] without giving much explanation. In chapter 3, I provide a theoretical justification of the algorithm from the theory of Markov chains and non-negative matrices and explain the derivation of the algorithm in detail. I generalized the algorithm to be able to push towards the final state and show the relation between the test for stochasticity and the propagation in the matrix-vector multiplication.

After we published the forward-backward decoding, the authors of [Nolden et al.(2013)] showed that it is possible to use our recipe for generating the exactly matching backward models [Hannemann et al.(2013)] in an incremental forward-backward decoding setup. While I was at my second internship at MSR under the supervision of Jasha Droppo, together with the authors of [Maleki et al.(2014)], we were looking for a way to implement the approach of [Maleki et al.(2014)] in a parallel speech decoder for LVCSR. Realizing the analogy of the approach of decoding mis-matching portions of speech [Nolden et al.(2013)] and the approach of decoding chunks [Maleki et al.(2014)], I implemented the incremental forward-backward decoding into the MSR recognizer and ran experiments to analyze the potential of using the incremental decoding for the parallelization of the decoding (section 5.3).

Since the initial algorithm for the reversal of ARPA LMs was only for transducers using the epsilon-arc approximation for back-off arcs, together with Jasha Droppo, I developed the constructive approach for LM reversal described in section 4.2 and showed that it is also valid for failure arcs. Afterwards, I also derived the proof for correctness in section 4.4 by a series of pushing operations and the derivation of the LM reversal from Bayes' rule in section 4.5.

1.2.2 Structure of the thesis

This thesis is organized as follows:

- **Chapter 2** introduces the basic concepts and necessary definitions for automatic speech recognition, weighted finite state transducers, language models and the construction of recognition networks.
- **Chapter 3** provides the theoretical framework of the alternative weight pushing algorithm by deriving it from the theory of Markov chains and non-negatives matrices.
- **Chapter 4** describes the constructive approach for the exact reversal of back-off language models as well as the formal proof and derivation.
- **Chapter 5** Explains the application of the forward and backward recognition networks in speed-up techniques.
  - **Section 5.2** explains the construction of the symmetric backward recognition network from its components.
  - **Section 5.3** shows experiments with the incremental decoding in a dynamic network decoder.
  - **Section 5.4** explains the tracked decoding and shows experiments exploring its parameters.
- **Chapter 6** Summarizes the findings in this thesis.

Chapter 2

Weighted finite state transducers and LVCSR decoding

2.1 Automatic speech recognition

The task in automatic speech recognition (ASR) is to recognize the words uttered in a segment of recorded audio and to correctly transcribe them to their corresponding textual word form. $\mathbf{W} = w_1, \ldots\ldots, w_n$ is the (unknown) uttered sequence of words (usually from a fixed vocabulary V). The encoding and transmission of the audio signal introduces errors due acoustic deviations of the channel (microphone, telephone network, etc.). The resulting audio will be recorded, and then some acoustic analysis (feature extraction) is performed, resulting in the sequence of acoustic vectors $\mathbf{X} = x_1, \ldots, x_m$, called acoustic observation. The task in ASR is to decode the observation $\mathbf{X}$ to the (possibly wrong) word sequence $\hat{\mathbf{W}}$.

The dominant approach to ASR is statistical pattern matching - i.e. to learn patterns from training examples and for the recognition, the observation is compared against the trained patterns and classified according to the goodness of match. To decode an utterance, we search for the word sequence $\hat{\mathbf{W}}$ with the maximum a-posteriori probability (MAP) given the acoustics. This results in the fundamental equation of speech recognition:

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{W}} P(\mathbf{W}|\mathbf{X}) = \arg\max_{\mathbf{W}} \frac{P(\mathbf{W})P(\mathbf{X}|\mathbf{W})}{P(\mathbf{X})} \tag{2.1}$$

Since the posterior probability $P(\mathbf{W}|\mathbf{X})$ is difficult to model, we applied Bayes' rule. The probability of the observation sequence $P(\mathbf{X})$ for a particular utterance doesn't depend on the hypothesized word sequence and doesn't influence the $\arg\max$. $P(\mathbf{X}|\mathbf{W})$ is called the acoustic model, which computes the likelihood that the observation $\mathbf{X}$ will be produced, when the speaker utters the words $\mathbf{W}$ and $P(\mathbf{W})$ is the language model, which is the prior probability that the speaker utters the word sequence $\mathbf{W}$.

Commonly, the acoustic model is a hidden Markov model (HMM, example in figure 2.1). A first-order HMM is defined by a finite set of states $s_i \in \mathcal{Q}$, the state transition probabilities $a_{ij} = P(s_j|s_i)$, a set of emission symbols $x \in \mathcal{X}$ (in our case continuous, but can be also discrete) and the emission probabilities $b_i(x_t) = p(x_t|s_i)$. The state transitions $P(s_{i+1}|s_i)$ model the temporal structure of speech. The sequence of states is not observed. The emission probabilities model the acoustic observations $P(x_i|s_i)$, i.e. the sequence of emission symbols is observed. In the HMM framework, a common approximation is to

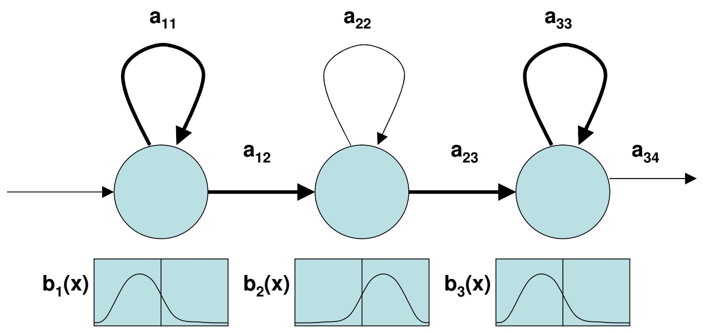


Figure 2.1: *Three-state left-to-right Hidden Markov Model with depicted one-dimensional continuous emission probabilities and a selected path through it (figure from Lukáš Burget).*

search for the optimal state sequence **S** instead of searching for the optimal word sequence:

$$\hat{\mathbf{W}} \approx \arg\max_{\mathbf{S}} \prod_{i=0}^{m-1} P(x_i|s_i)P(s_{i+1}|s_i) \tag{2.2}$$

In the most simple case of isolated word recognition, the HMMs model whole words and in decoding, the Viterbi algorithm searches for best path for each HMM separately, and then the scores of the best paths are compared. In connected speech recognition, we construct a composite model of the word models and the language model (LM) functions as a grammar, which constrains which words can follow each other. This is the most simple form of a recognition network.

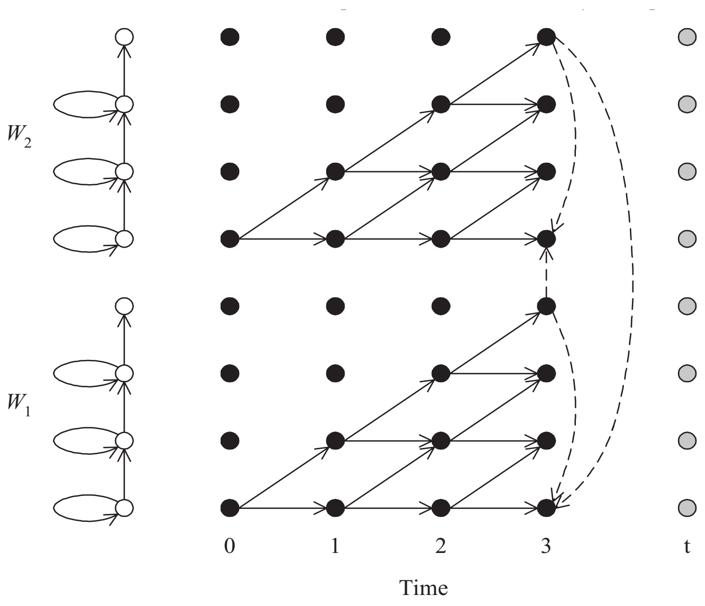


Figure 2.2: *Viterbi search in composite model [Young et al.(2006)]. Each word is represented with a left-to-right HMM, and the final states of words are connected to the initial states according to the LM. The initial and final states are not tied to an observation (non-emitting). Therefore, during decoding, the word connections are followed within the same time frame.*

If there are too many words in the vocabulary to reliably estimate all word models on the training data, it is necessary to use HMMs that model sub-word units (e.g. phonemes) instead of whole words. In this case, we need a mapping from words to phonemes, which is given in the form of a pronunciation lexicon. We give a small example:

```
ONE   w ah n
TWO   t uw
THREE th r iy
```

Given this example pronunciation lexicon, figure 2.3 shows the corresponding simple recognition network for connected speech recognition with phoneme sub-word units.

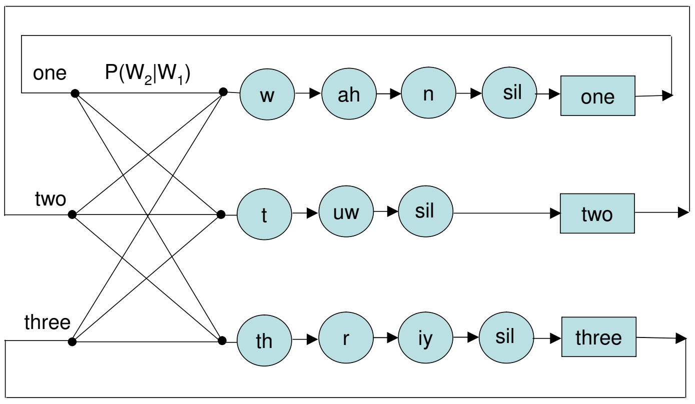


Figure 2.3: *Simple phoneme-based recognition network. Words are modeled by phonemes (sil: silence) and bi-gram probabilities are applied at word transitions (figure from Lukáš Burget). Each phoneme is modeled by a three-state HMM, thus in the figure, we show only the transitions connecting the phoneme models and words.*

In figure 2.4, we summarize the basic structure of an ASR system. The recognition network is a composition of the LM (accepting word sequences), the pronunciation lexicon (mapping the words to phonemes), and the HMM structure, modeling the temporal structure of the phonemes. We conceptually split the HMMs into the HMM structure (transition probabilities a_{ij}), which are considered as part of the static recognition network and emission probabilities $b_i(o_t)$, which produce the acoustic likelihoods scores $p(x_i|s_i)$ for each frame, and are usually applied dynamically during the recognition.

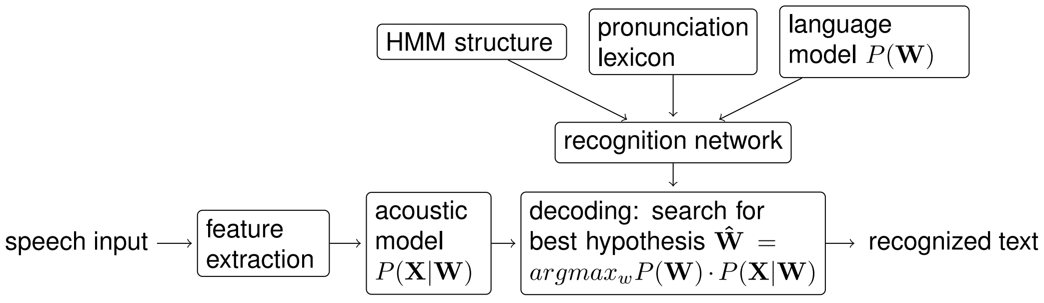
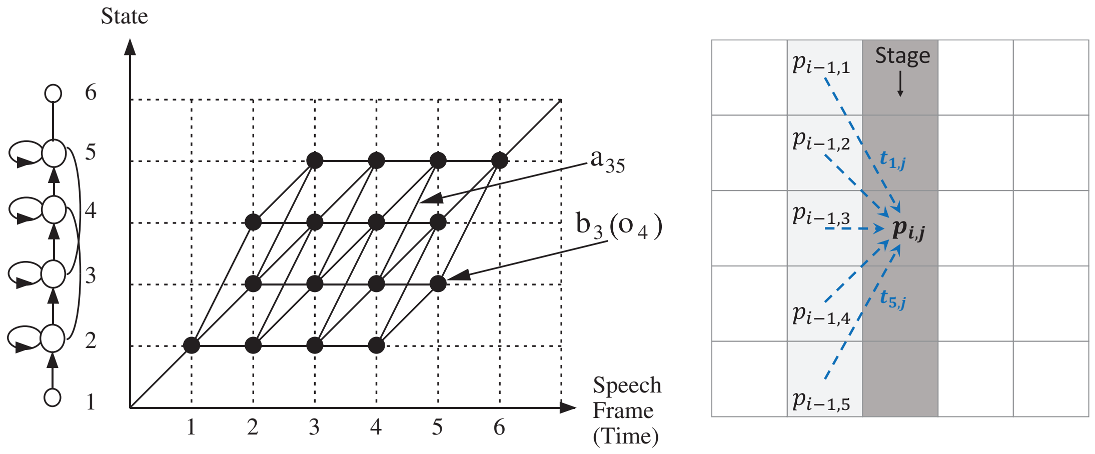

HMM structure
pronunciation lexicon
language model $P(\mathbf{W})$
recognition network
speech input
feature extraction
acoustic model $P(\mathbf{X}|\mathbf{W})$
decoding: search for best hypothesis $\hat{\mathbf{W}} = argmax_w P(\mathbf{W}) \cdot P(\mathbf{X}|\mathbf{W})$
recognized text

Figure 2.4: *Components of automatic speech recognition*

2.2 Speech recognition decoding

The algorithm [Viterbi(1967)] that is used for the search for the best path through the recognition network belongs to the class of algorithms called dynamic programming [Bellman(1952)]. In this class of algorithms, we can reduce the global task of finding the best path, to the task of recursively solving the sub-problem of choosing the predecessor with the best partial path up to this time. Figure 2.5 shows this for the Viterbi algorithm.



$$p_{i,j} = \max_k (p_{i-1,k} * t_{k,j})$$

Figure 2.5: *Dependencies and parallelism in the Viterbi algorithm. Left: Viterbi algorithm applied to an HMM in isolated word recognition [Young et al.(2006)] Right: Dependencies in time-synchronous Viterbi search: The sub-problem of finding the max token in the current time step depends on all incoming arcs from the previous time step (stage) [Maleki et al.(2014)].*

The Viterbi algorithm [Viterbi(1967)] is a special form of the single source shortest path problem (SSSP), which has been extensively studied [Gibbons(1985)], [Cormen et al.(2009)]. As seen in the left part of figure 2.5, the search graph (in the example the HMM out-most left) unfolds to the trellis structure of search states, where each HMM state is copied for each time step. Often, the Viterbi algorithm is implement as a token passing algorithm [Young et al.(1989)]. We think of a token as a record of a particular state in the HMM that is active on a particular time frame and contains the accumulated score of the partial path explored so far (as well as a back-pointer).

As opposed to figure 2.5, the search graphs used in LVCSR (seen as composite HMM) are usually huge and the resulting number of search states in the trellis is an order of magnitude higher. Therefore, we would not construct the full trellis, but build it frame by frame. Every graph state in a particular time frame can only be reached by states from the previous time frame (figure 2.5). This special dependency structure is used in the time-synchronous Viterbi algorithms, which are applied in the majority of speech decoders. The advantage is, that only scores of paths of the same length need to be compared, and only the states of the current frame need to be kept in memory.

Alternatively, we can represent the state transitions in a matrix **P**, where each entry p_{ij} represents the sum of all transitions from state i to state j. Given a vector of forward (Viterbi) probabilities for each state at a certain time, the vector of forward probabilities for the next time step is obtained by "multiplying" the matrix P_{ij} (tropical semi-ring, explained in next section 2.3). The weights of the matrix P_{ij} need to be computed at each time step – the transition probabilities a_{ij} are fixed, but we need to evaluate the emission probabilities $b_j(o_t)$ given the observation at the current time step. Usually, the probabilities are computed on-the-fly by combining the so called acoustic scores $(b_j(o_t))$ with the so called graph scores (a_{ij}, containing LM and HMM transitions).

Due to the huge search spaces, it is not efficient or not possible at all to perform an exhaustive search. In this case, pruning needs to be applied (e.g. beam search [Lowerre(1976)]). The choice of an efficient pruning strategy is the dominant factor in determining the recognizer speed. Typically, only a percent of the states of the search graph are active at each time frame and as a result, for the straight forward implementation, their data structures are scattered in memory, which leads to cache failures and slow memory access. Thus, the application of pruning changes the properties of the basic SSSP fundamentally and the algorithm design becomes more complex.

2.3 Weighted finite state transducers

Throughout this work, we think of weighted finite state acceptors/transducers (WFSA/WFST) as having a set of states with one distinguished start state[1]. Each state has a final weight (or $\bar{0}$ (infinite cost) for non-final states) and there is a set of arcs between the states, where each arc has an input label (for WFST also an output label), and a weight. Formally, we introduce a WFSA [Mohri(1997)], [Mohri and Riley(2001)] as:

$$A = (\Sigma, Q, i, F, E, \lambda, \rho) \text{ over a semi-ring } (\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1}) \tag{2.3}$$

A semi-ring [Kuich and Salomaa(1986)] is an algebraic structure – it is a ring that may lack negation. It has two associative operations $\oplus$ and $\otimes$ that are closed over the set $\mathbb{K}$, they have identities $\bar{0}$ and $\bar{1}$, respectively. $\otimes$ distributes over $\oplus$ and $\bar{0}$ is an annihilator. When the weights represent probabilities, the appropriate semi-ring is the probability semi-ring $(\mathbb{R}_+, +, \times, 0, 1)$ [2]. For numerical stability, often log-probabilities are used, which results in the log semi-ring $(\mathbb{R} \cup \infty, \oplus_{log}, +, \infty, 0)$ with $\forall a, b \in \mathbb{R} \cup \infty, a \oplus_{log} b = -\log(\exp(-a) + \exp(-b))$. When we use the Viterbi approximation, we replace the $\oplus_{log}$ with the minimum and the resulting semi-ring is the tropical semi-ring $(\mathbb{R}_+ \cup \infty, \min, +, \infty, 0)$. A special class of semi-rings are divisible semi-rings, i.e. $\forall a, b \in \mathbb{K}, a \oplus b \neq \bar{0} : \exists a_1 \in \mathbb{K} : a = (a \oplus b) \otimes a_1$. In other words, a_1 is the remainder of the division of a by $a \oplus b$ and we introduce the inversion operation: $a_1 = (a \oplus b)^{-1} \otimes a$.

A WFSA is given by:

- an alphabet or label set Σ
- a finite set of states Q
- an *initial state* $i \in Q$
- a set of *final states* $F \subseteq Q$
- a finite set of *transitions* $E \subseteq Q \times (\Sigma \cup \epsilon) \times \mathbb{K} \times Q$
- an *initial weight* λ
- and a *final weight* $\rho(q)$

[1] As used in the OpenFST toolkit: `www.openfst.org`

[2] Sometimes, it is used in a more general sense, not limiting the numbers to between zero and one.

A transition $t = (p[t], l[t], w[t], n[t]) \in E$ can be represented as an arc from the *source state* $p[t]$ to the *destination state* $n[t]$, with the *label* $l[t]$ and *weight* $w[t]$, which is typically a probability (or log-probability). Transitions labeled with the *empty string* ϵ consume no input. For each state $q \in Q$, $E[q]$ denotes the set of transitions leaving q. The transition weights can be also represented in form of the transition matrix $\mathbf{P}_{i,j} \in |Q| \times |Q|$, where each entry $p_{ij} = w[t]$ contains the sum of weights of all transitions t from state $i = p[t]$ to state $j = n[t]$. If no corresponding transition exists, the entry is $\bar{0}$.

A *path* in A is a sequence of consecutive transitions $\pi = t_1 \dots t_n$ with $n[t_i] = p[t_{i+1}], i = 1, \dots, n-1$. A *successful path* is a path from the initial state i to one of the final states $f \in F$. The label of a path π is the concatenation of the labels of its constituent transitions: $l[\pi] = l[t_1] \dots l[t_n]$ and the weight associated to π is the $\otimes$-product of the initial weight, the weights of its constituent transitions and the final weight $\rho(n[t_n])$ of the state $f = n[t_n]$ reached by π:

$$w[\pi] = \lambda \otimes w[t_1] \otimes \dots \otimes w[t_n] \otimes \rho(n[t_n]) \tag{2.4}$$

The *total weight* of an WFSA is the sum of all successful paths from the initial state i to all of the final states F:

$$w_{tot} = \bigoplus_{\forall \pi, p[\pi]=i, n[\pi]=f \in F} w[\pi] \tag{2.5}$$

A symbol sequence is accepted by A if there exists at least one successful path π labeled with $x = l[\pi]$. The weight associated by A to the sequence x is then the $\oplus$-sum of the weights of all the successful paths π labeled with x. In the same way, the weight of a set of paths is the $\oplus$-sum of the weights of the individual paths. A state q is *accessible* if there is a path from the initial state i to q. A state q is *co-accessible* if there is a path from q to a final state $f \in F$. A WFSA is *trim* or *connected* if it contains neither inaccessible nor co-inaccessible states. A WFSA is *stochastic*, if the transitions out of each state q (and the final-probability) "sum to one" in the given semi-ring:

$$\forall q \in Q, \left(\bigoplus_{e \in E[q]} w[e] \right) \oplus \rho[q] = \bar{1} \tag{2.6}$$

It is only possible to make the WFSA stochastic if the total weight of the entire WFSA is $\bar{1}$. Otherwise, there is a *left-over weight* that must be handled. In practice this may be discarded, or put on the initial or final states of the WFSA.

WFSTs generalize WFSAs by replacing the single transition label by a pair (i, o) of an input label and an output label:

$$A = (\mathcal{A}, \mathcal{B}, Q, i, F, E, \lambda, \rho) \text{ over a semi-ring } (\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1}) \tag{2.7}$$

, where $\mathcal{A}$ is the finite input alphabet, $\mathcal{B}$ is the final output alphabet, and a transition $t = (p[t], i[t], o[t], w[t], n[t]) \in E$ has an input label $i[t]$ and an output label $o[t]$.

A WFST associates pairs of symbol sequences and weights, i.e. it represents a weighted binary relation between symbol sequences. Two WFSAs are equivalent if they associate the same weight to each input string, i.e. weights may be distributed differently along the paths of two equivalent acceptors [Mohri and Riley(2001)]. Two WFSTs are equivalent if they associate the same output sequence and weights to each input sequence, i.e. the distribution of the weight or output labels along paths need not be the same in the two WFSTs.

There might be some confusion about the terms "weight" and "cost". Usually, by mentioning "weights" in this thesis, we refer to a general probability representation in any semi-ring, and by mentioning "costs", we refer to probabilities in the log- or tropical semi-rings, where a cost is a floating point number that typically represents a negated log-probability.

2.4 Weighted finite state transducer based decoding

Also LVCSR can be formulated in the framework of weighted finite state acceptors/transducers (WFSA/WFST) [Allauzen et al.(2004)], [Mohri et al.(2008)]. As seen in figure 2.6, the HMM search graph can be represented as WFST. We refer to this WFST as the HMM structure transducer H. The input labels are identifiers of probability density functions (PDF-ids, often context-dependent HMM states).

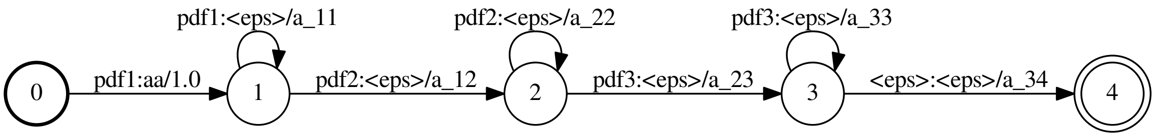


Figure 2.6: *WFST H corresponding to three-state left-to-right HMM from figure 2.1. The notation of an arc is „input:output/weight", where „<eps>" stands for the ϵ (no symbol). Instead of attaching the emission probabilities to the state, we attach them to the incoming arcs of a state. Thus, the input labels correspond to identifiers of probability density functions (PDF-ids). During decoding, the PDF-ids are used to evaluate the emission probabilities assigned to the destination state of the arc. The acoustic likelihood score is combined ($\otimes$) with the weight of the arc, corresponding to the transition probability a_ij. The output label is the identity of the phoneme (aa). The final arc is non-emitting (<eps> input). In composite models, it can serve to interconnect the individual (sub-word HMM) models according to the pronunciation lexicon and the LM.*

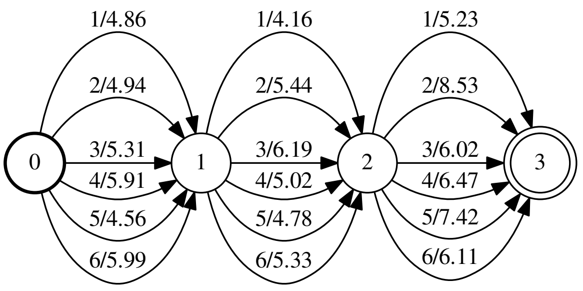


Figure 2.7: *Acceptor U describing the acoustic scores of the utterance [Povey et al.(2012)].*

To decode an utterance of T frames in the WFST framework, i.e. to find the most likely state sequence through the trellis, we construct an acceptor (WFSA) U, as in figure 2.7. It has $T{+}1$ states, with an arc for each combination of (time, PDF-id). The weights on these arcs correspond to negated and scaled acoustic log-likelihoods[3]. We construct the trellis S

[3]In figure 2.7, we represented the acoustic likelihoods (which can be very small numbers) in the negative log-semi-ring, while in figure 2.6, we showed the weights the probability semi-ring for illustrational purposes. Of course, the composition must be done with both weights in the same semi-ring.

with WFST composition [Povey et al.(2012)]:

$$S \equiv U \circ H. \tag{2.8}$$

The trellis has approximately $T+1$ times more states than H. The decoding problem is equivalent to finding the best path through S, which can be done with the shortest path algorithm in the corresponding semi-ring. The best path is represented as a linear WFST. The output symbol sequence of this best path represents the decoding result, i.e. the recognized sequence of phonemes (and words)[4]. The input symbol sequence of the best path represents the sequence of PDF-ids used for each time frame. If there is a direct correspondence between the PDF-id and the state in the H transducer (for example if pdf1 $= 1$, pdf2 $= 2$, pdf3 $= 3$ in figure 2.6), we obtain the sequence of states as well. This is called the state-level alignment. As done in the Kaldi toolkit [Povey et al.(2011)], the input labels can be constructed in such a way, that they represent the PDF-id, the graph state and the transition number, so that all this information will be available in the state-level alignment. We refer to these identifiers as Kaldi *transition-ids*.

In practice, S is not searched exhaustively, but beam pruning is used. Let B be the searched subset of S, containing a subset of the states and arcs of S obtained by some heuristic pruning procedure. When we do Viterbi decoding with beam-pruning, we are finding the best path through B. Since a LVCSR system can have up to ten-thousands of PDF-ids and there are typically hundreds to thousands of frames in an utterance, it is not very practical to construct U in advance. Also, due to pruning, just a subset of PDF-ids needs to be evaluated for each frame. Therefore, we are dynamically composing U during decoding. This corresponds to combining ($\otimes$) the acoustic likelihood score with the arc weights (transition probabilities, called graph score) on-the-fly.

Conceptually, we split HMMs (containing transition probabilities and emission probabilities) into the HMM structure transducer H (figure 2.6) and the acoustic model (the emission probabilities, which produce the acoustic likelihoods scores $p(x_i|s_i)$ for each frame). The HMM structure transducer H represents the transitions, i.e. the part of the HMM network, that is fixed for all time steps. It maps from a sequence of acoustic unit identifiers (e.g. one PDF-id per frame) to a sequence of phonemes.

Nowadays, instead of phonemes as sub-word acoustic units, usually we use context-dependent phonemes. Most often, tri-phones are used (including the current phoneme and one to the left and to the right). Thus, the basic building block of the graph (figure 2.6) are tri-phone HMMs. In this case, an additional component is needed: the phoneme-to-context-phoneme mapping. Due to data sparsity not all possible context-dependent phonemes can be observed sufficiently often in training, usually, a clustering algorithm (e.g. decision tree) is applied to reduce the number of units to train.

2.4.1 Decoding graph construction in the Kaldi toolkit

Already in figure 2.4, we indicated the building blocks of a recognition network: the HMM structure, the pronunciation lexicon and the LM. Instead of the simple three-state HMM in figure 2.6, now, we use a recognition network composed of thousands of sub-word HMMs, connected according to the phoneme-to-context-phoneme mapping, the pronunciation lexicon and the LM. We represent each component as WFST. The standard recipe for the decoding graph construction is [Mohri et al.(2008)]:

$$HCLG = \min(\det(H \circ C \circ L \circ G)), \tag{2.9}$$

[4]In our example just “aa”, but for composite models, we obtain the sequence of phoneme HMMs used.

here, H, C, L and G are the components, which are created separately and are integrated into a single WFST($HCLG$) (called decoding graph) with WFST composition (denoted as $\circ$). H, C, L and G represent the HMM structure, the phonetic context-dependency transducer, the lexicon transducer and the LM (grammar), respectively. As in figure 2.6, the result is a "fully expanded" graph, where the arcs correspond to HMM transitions, the input labels are the identifiers of PDF-ids (context-dependent HMM states), and the output labels represent words as accepted by the LM. For both the input and output labels, the special symbol ϵ may appear, meaning "no label is present." In the following, we briefly describe the H, C and L transducers. The G transducer is described in detail in section 2.5. Unless otherwise mentioned, the experiments in this work were conducted with the Kaldi toolkit [Povey et al.(2011)][5].

The HMM structure transducer H was already described in figure 2.6, but here, we extend our model to an ergodic loop of many sub-word HMMs. An example will be given in the upper part of figure 5.6. As a particularity of the Kaldi toolkit [Povey et al.(2011)], the HMM structure transducer is created without self-loops (called H_a) to reduce the size of the model. The self-loops are added in a final step.

The context-dependency transducer C is a mapping from context-dependent phonemes to phonemes. Figure 2.8 explains how to construct a deterministic mapping and figure 2.9 shows the full context WFST C for the toy example with only two phonemes. From this example, it is clear, that C is huge, and therefore it is often constructed and composed on-the-fly.

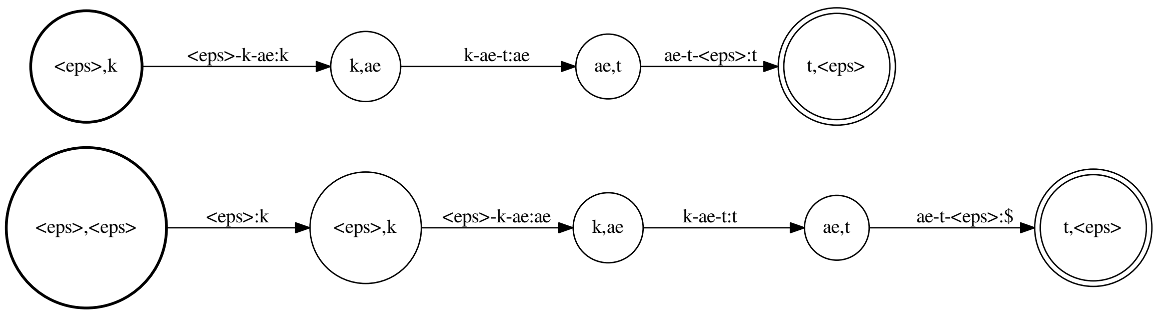


Figure 2.8: *One path of the context-dependency transducer C, mapping from context-dependent phonemes to phonemes. We compose it with the lexicon WFST L from the right, i.e. we think of the phonemes being generated from the lexicon. Therefore, the output symbols are actually the input of the mapping, which might be confusing. Upper sequence: Given the word 'cat' and its pronunciation 'k ae t', the naive implementation would be to have one arc for each phoneme (output symbol) and put the corresponding tri-phone on the input label. The tri-phone encoding 'k-ae-t' means that 'ae' is the center phoneme, with 'k' and 't' as left and right context. This naive implementation results in a FST, that is not deterministic (given the output symbols). Lower part: The deterministic solution [Mohri et al.(2008)] is to delay the tri-phone symbols until all its constituting phonemes have been observed. To compensate the delay, we introduce a special end-of-sequence symbol '$' on the last arc.*

The lexicon WFST is a mapping from words to phoneme sequences. We give an example taken from Vasil Panayotov's blog[6]. We are given the following pronunciation lexicon:

[5]The graph construction is only described very briefly in [Povey et al.(2011)], so whenever we cite this paper, we also refer to the official Kaldi documentation `http://kaldi-asr.org/doc/graph.html`.

[6]`http://vpanayotov.blogspot.cz/2012/06/kaldi-decoding-graph-construction.html`

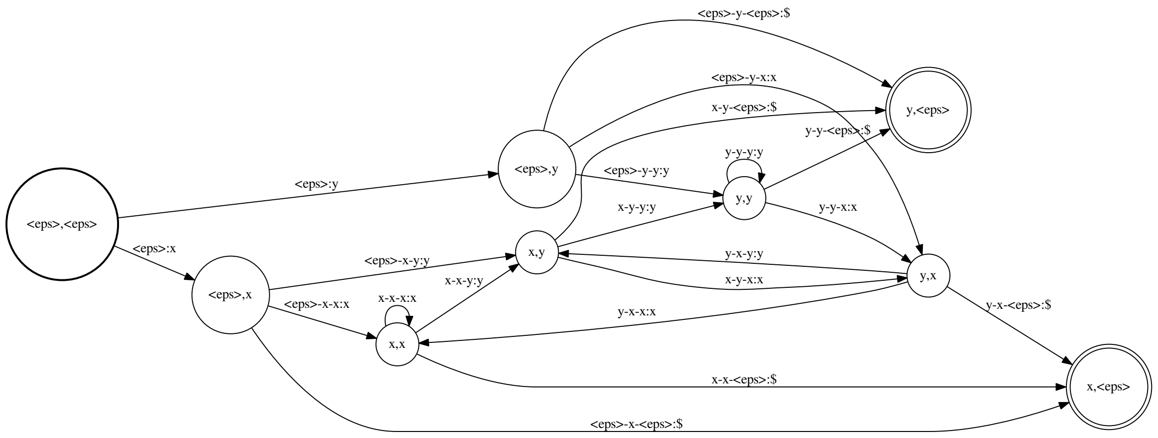


Figure 2.9: *Deterministic context-dependency transducer C, mapping from context-dependent phonemes to phonemes, shown for only two phonemes 'x' and 'y'. We compose it with the lexicon WFST L from the right and with the HMM transducer H from the left. The tri-phone encoding 'x-y-z' means that 'y' is the center phoneme, with 'x' and 'z' as left and right context. '\$' is the end-of-sequence symbol.*

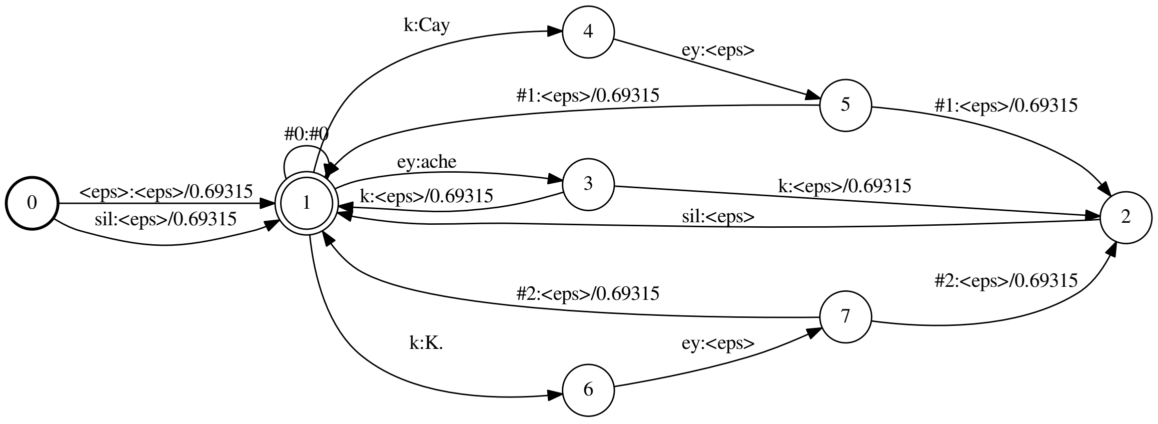


Figure 2.10: *Pronunciation lexicon transducer L. The first arc for each word (starting from state 1) outputs the word identifier. The last arc of a word is looped back to the word initial state 1, so that all possible words can follow. There is an optional silence (sil) at the begin of the sentence and in between words. Thus, we 'split' the word-final arc, either looping back to state 1 with log-probability -log(0.5) or going over state 2 and producing a silence (sil). The '#0' is a disambiguation symbol forwarded from the G transducer.*

```
ache   ey k
Cay    k ey #1
K.     k ey #2
```

The symbols `#1` and `#2` are disambiguation symbols [Mohri(1997)]. Without adding them, the resulting WFST would not be determinizable, since the phoneme sequence 'k ey' can result in two different words. Therefore, we insert auxiliary phone symbols disambiguating the two possible homophones before the determinization[7]. We also need to add

[7]Disambiguation symbols need to be passed through the C and H transducers by adding self-loops at each state, and after determinizing the final WFST $HCLG$, we replace them by ϵ.

disambiguation symbols if a phoneme sequence can be a prefix of another. The resulting lexicon WFST is shown in figure 2.10.

So far, we have introduced the WFST components H_a, C and L. The remaining WFST G is explained in the following section. The final formula for the graph creation in Kaldi is (asl - add self loops, rds - remove disambiguation symbols):

$$HCLG = asl(min(rds(det(H_a \circ min(det(C \circ min(det(L \circ G)))))))) \tag{2.10}$$

2.5 Back-off language models as finite state automata

The ARPA language model (LM) format is one of the most widely used standards for encoding N-gram back-off LMs in text form. The ARPA format has most probably been created by Douglas B. Paul [Paul and Baker(1992)] from MIT Lincoln Labs for the DARPA Spoken Language System (SLS) community - hence its name. A wide class of LMs can be encoded in the ARPA format - including e.g. interpolated LMs.

Statistical language models estimate the probability of a word sequence W (usually sentence or utterance):

$$\begin{aligned} P(W) &= P(w_1, w_2, \ldots, w_N) \\ P(W) &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_N|w_1, \ldots, w_{N-1}), \end{aligned} \tag{2.11}$$

where the terms $P(w_i|\ldots)$ are the conditional probabilities of words given their history. N-gram LMs approximate the conditional probability of a word by shortening the history to the previous $N-1$ words:

$$P(w_1 \ldots w_m) \approx \prod_{i=1}^{m} P(w_i|w_{i-N+1}\ldots w_{i-1}) \tag{2.12}$$

Every history corresponds to a possible state of the search space. Even if more powerful LMs are available today (based on e.g. on the maximum-entropy principle or on recursive neural networks), often they are still approximated with N-gram models, since long histories lead to intractable search spaces.

Statistical smoothing techniques are applied to the distribution of counts, since it is not possible to observe all possible word sequences (N-gram, including history) sufficiently often in the training texts. On top of that, typically models of different N-gram order are combined. Either different orders of history are interpolated, or the higher order model performs backing-off by leaving out the first word in the history and looking-up the shorter history in the lower order model. This process is repeated recursively until the word in context is found.

N-gram LMs can be expressed as weighted finite state acceptors (WFSA) - each LM history corresponds to one state of the automaton ($h_i = w_{i-N+1}\ldots w_{i-1}$). N-gram LMs can be conveniently integrated into the speech decoding process - the search space is defined by the WFSA corresponding to the N-gram LM. However, the number of possible states of a model of order N with a vocabulary size V is V^{N-1} and the number of possible arcs (and N-grams) is V^N, which becomes clearly intractable for higher orders of N (typical vocabulary sizes go into the hundreds of thousands). As a consequence, ARPA language models only store the probabilities of those N-grams that occur sufficiently often. The probability of other N-grams is estimated by recursively „backing-off“ to models of lower order $(N-1)$

that use shortened histories (leaving out the first word of the history). Back-off LMs were introduced by S. Katz [Katz(1987)]:

$$P_{Katz}(w_i|w_{i-N+1}\dots w_{i-1}) = \\ = \begin{cases} P'(w_i|h_i) = d_{w_{i-N+1}\dots w_i} \cdot \dfrac{C(w_{i-N+1}\dots w_{i-1}w_i)}{C(w_{i-N+1}\dots w_{i-1})} & \text{if } C(w_{i-N+1}\dots w_i) > k \\ \alpha_{w_{i-N+1}\dots w_{i-1}} \cdot P_{lower}(w_i|w_{i-N+2}\dots w_{i-1}) & \text{otherwise} \end{cases} \tag{2.13}$$

Here, d is the amount of discounting applied ([Katz(1987)] used Good-Turing smoothing), C is the occurrence count of the given N-gram in the training corpus and k is the cut-off frequency (minimum number of occurrences). $\alpha_{w_{i-n+1}\dots w_{i-1}}$ is the so called back-off weight, which is dependent on the current history. It usually corresponds to the sum of probability mass that was discounted from all N-grams sharing the same history and is now available to be re-distributed by the lower order distribution P_{lower}, that can be recursively defined in exactly the same way as P_{Katz}. Alternatively, a different type of back-off distribution can be used, as e.g. in Kneser-Ney smoothing [Kneser and Ney(1995)].

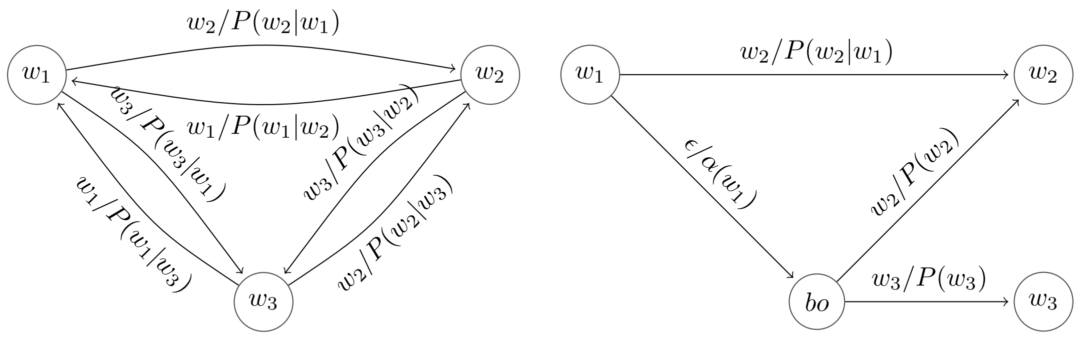


Figure 2.11: *Weighted finite state acceptor (WFSA) implementation of a bi-gram LM. Left: fully connected model ($V \times V$ arcs) Right: WFSA approximation of a bi-gram back-off model [Mohri et al.(2008)], just showing the representation of transitions leaving state w_1. The bi-gram $w_1 w_2$ was seen sufficiently often during training and is thus represented by a direct link between the history states w_1 and w_2. The bi-gram $w_1 w_3$ was not seen sufficiently often, thus the model backs-off to the history-less state "bo" with the cost of the back-off weight $\alpha(w_1)$. No symbol is consumed in this transition - indicated by the ϵ-symbol. Leaving the back-off state, the lower order (uni-gram) probabilities are applied ($P(w_3)$). The approximation with the back-off state can greatly reduce the number of arcs, but it also introduces non-determinism. If ϵ would be a regular label, the WFSA would be deterministic (only a single outgoing arc per label in each state). However, since ϵ doesn't consume any symbol, the bi-gram $w_1 w_2$ can be either formed by taking the arc $w_1 \rightarrow w_2$ or by going over the back-off arc: $w_1 \rightarrow bo \rightarrow w_2$.*

As seen in figure 2.11, back-off LMs can be represented as WFSA, but since not all possible history states and arcs can be specified for higher N and V, usually an approximate structure with back-off arcs is used [Allauzen et al.(2003)], [Mohri et al.(2008)]. The probabilities $P'(w_i|h_i)$ and the back-off weights α (equation 2.13) are pre-computed, and only those are stored in the ARPA format. For each N-gram (e.g. tri-gram abc) with $C > k$, an ARPA file contains an entry in the form '$P'(c|ab)$ abc $\alpha(abc)$', where $P'(c|ab)$ is the discounted probability $P(c|ab)$, and $\alpha(abc)$ is the back-off weight of backing-off from the higher order N-gram abc to the shortened history bc. Thus, if $C(abcd) < k$ then $P(d|abc) = \alpha(abc)P(d|bc)$. For the highest order, there are no explicit history states. For example in a tri-gram LM, we don't create states for every tri-gram, but we use the tri-grams as connections between bi-gram states (e.g. the tri-gram abc connects the states ab and bc).

We can interpret the highest order (e.g. tri-gram) connections in an alternative way: we could create the highest order (tri-gram) history state as target of the transition, but there would only be the possibility of immediately backing-off to the corresponding lower order (bi-gram) state. In this interpretation, the back-off weights for the highest order N are assumed to be always one (zero in log-domain), and therefore there is no need to specify them. This interpretation has the advantage, that there are only two types of arcs: going towards a higher order by extending the history, and backing-off to lower orders by shortening the history. This simplifies some derivations in chapter 4. Figure 2.12 shows an example ARPA text file and the corresponding WFSA.

```
\data\
ngram 1=4
ngram 2=2
ngram 3=2

\1-grams:
-5.234679 a -3.3
-3.456783 b
0.0000000 <s> -2.5
-4.333333 </s>

\2-grams:
-1.45678 a b -3.23
-1.30490 <s> a -4.2

\3-grams:
-0.34958 <s> a b
-0.23940 a b </s>
\end\
```

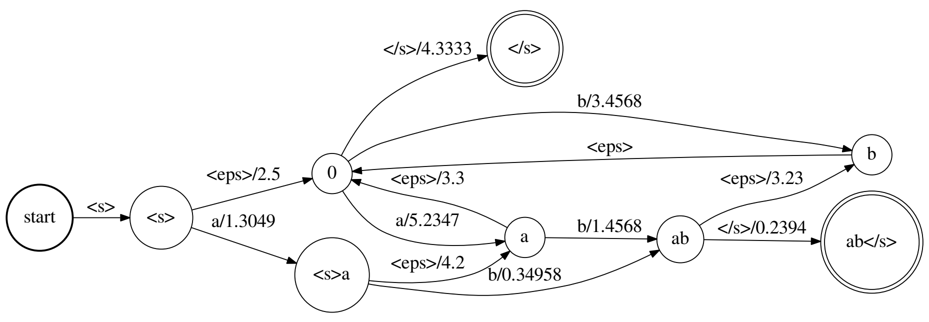


Figure 2.12: **Left:** *Definition of a tri-gram ARPA back-off language model. For each N-gram 'abc', there is an entry in the form '$P'(c|a,b)$ abc $\alpha(a,b,c)$', where $P'(c|a,b)$ is the discounted probability $P(c|a,b)$, and $\alpha(a,b,c)$ is the back-off weight of backing-off from a higher order N-gram to the shortened history abc. The probabilities are by convention given as logarithms to the basis of two.* **Right:** *The WFSA resulting from the tri-gram back-off ARPA LM defined on the left. The highest-order N-grams (tri-grams) behave slightly differently than lower-order N-grams: The transition for tri-gram <s>ab is going from state <s>a to state ab. In an alternative interpretation, this is equivalent to going to an imaginary state <s>ab and immediately backing-off to state ab. If for some reason the bi-gram ab would be missing in the ARPA file (removed line '$-1.45678\,a\,b\,-3.23$'), the state ab would be created as target state for the tri-gram abc, however, the arc from a to ab would not exist, and the back-off arc from ab to b would be with zero cost.*

2.5.1 Difficulties with the representation of back-off arcs

As seen in figure 2.11, the approximate structure of back-off arcs with the symbol ϵ introduces non-determinism. Therefore, it is important to pay attention to the computation of the arc weights of the LM WFST G. The same structure (figure 2.11) can be used to represent both back-off LMs and interpolated LMs. We compare the formulas for both models:

$$
\begin{aligned}
P_{backoff}(w_i|h_i) &= \begin{cases} P'(w_i|h_i) & \text{if } C(w_i,h_i) > k \\ \alpha_{bo}(h_i) \cdot P_{lower}(w_i|\bar{h_i}) & \text{otherwise} \end{cases} && (2.14)\\
P_{int}(w_i|h_i) &= P'(w_i|h_i) + \alpha_{int}(h_i) \cdot P_{lower}(w_i|\bar{h_i}) && (2.15)\\
\bar{h_i} &= w_{i-N+2} \dots w_{i-1}. && (2.16)
\end{aligned}
$$

We observe, that the only principal difference is the incorporation of the lower order probabilities – either we add them in the interpolated LM, or we decide to use it based on the condition $C(w_i, h_i) > k$. That means, in the back-off LM, we should not use the lower order distribution, if the higher order N-gram was observed sufficiently often. This has the important consequence, that the back-off weights α are computed differently in both cases:

$$
\begin{aligned}
\alpha_{int}(h_i) &= 1 - \sum P'(w_i|h_i) && (2.17)\\
\alpha_{bo}(h_i) &= \frac{\alpha_{int}(h_i)}{\sum_{w_i|C(w_i,h_i)<=k} P_{lower}(w_i|\bar{h_i})}. && (2.18)
\end{aligned}
$$

$\alpha_{int}(h_i)$ is computed so that $P_{int}(w_i|h_i)$ forms a valid distribution by assuming, that we always add the lower order distribution P_{lower}. This means, if we compute $P'(w_i|h_i)$, $\alpha_{int}(h_i)$ and $P_{lower}(w_i|\bar{h_i})$ for an interpolated LM, and use it as arc weights in the LM transducer G, we should always take the sum of all possible paths which accept the same symbol sequence. For example, in figure 2.11, we should sum the arc $w_1 \rightarrow w_2$ and the path going over the back-off arc: $w_1 \rightarrow bo \rightarrow w_2$ to correctly obtain the weight for the bi-gram $w_1 w_2$. In other words, we should use the probability semi-ring (or log-semi-ring). If we use the tropical semi-ring in decoding, which only picks the best of the possible paths, we do not obtain the correct probability.

Partly to avoid this problem and partly since the ARPA format is the most commonly used file format (which was designed for back-off LMs), many popular LM tool-kits[8] convert the probabilities of interpolated LMs to back-off LMs before saving them to the ARPA file:

$$
\begin{aligned}
P''(w_i|h_i) = P_{int}(w_i|h_i) &= P'(w_i|h_i) + \alpha_{int}(h_i) \cdot P_{lower}(w_i|\bar{h_i}) && (2.19)\\
\alpha''(h_i) &= \frac{1 - \sum P''(w_i|h_i)}{\sum_{w_i|C(w_i,h_i)<=k} P_{lower}(w_i|\bar{h_i})}. && (2.20)
\end{aligned}
$$

That means, we use the interpolated probability $P_{int}(w_i|h_i)$ instead of $P'(w_i|h_i)$. In case P_{lower} is itself an interpolated probability, we have to recursively add all lower orders. The resulting model can be used as a back-off LM. Its origin as an interpolated LM is no longer visible when stored in the ARPA format.

If we would just use $P_{int}(w_i|h_i)$, without changing the back-off weights $\alpha''(h_i)$, the decoding in the tropical semi-ring would produce the correct result, since it is guaranteed for interpolated LMs, that $P_{int}(w_i|h_i) \geq \alpha_{int}(h_i) \cdot P_{lower}(w_i|\bar{h_i})$. However, to interpret the model as a back-off LM, the back-off weights $\alpha''(h_i)$ are re-computed. Therefore, if the original counts are no longer available (only ARPA given), the original interpolation weights α_{int} are lost[9], and the decoding in the tropical semi-ring is not guaranteed to produce the correct result, since it is now possible that $P_{int}(w_i|h_i) < \alpha''(h_i) \cdot P_{lower}(w_i|\bar{h_i})$[10].

[8]For example SRILM http://www.speech.sri.com/projects/srilm/.

[9]We could reverse-engineer the equations, but due to complexities like missing N-grams and pruning, this is non-trivial.

[10]In rare cases, where $C(w_i, h_i) > k$ for almost all w_i, it is even possible, that $\alpha''(h_i) > 1.0$.

Luckily, this is not happening very frequently, and will not have a big impact e.g. on word error rates. For back-off LMs, $\alpha_{bo}(h_i)$ (equation 2.18) is computed under the assumption, that we only back-off, if $C(w_i, h_i) <= k$. Therefore, when we compute $P'(w_i|h_i)$, $\alpha_{bo}(h_i)$ and $P_{lower}(w_i|\bar{h_i})$ for a back-off LM (or convert an interpolated LM to a back-off representation as in equation 2.20) and use these as arc weights in G, we should only take the back-off arcs if there is no corresponding arc with the higher-order N-gram. For example, in figure 2.11, we are not allowed to use the back-off arc $w_1 \rightarrow bo \rightarrow w_2$, because there is a bi-gram arc $w_1 \rightarrow w_2$. In other words, for these models, when we represent back-off arcs with ϵ, we do not obtain the correct probability, neither with the probability semi-ring, which is summing all possible paths, nor with the tropical semi-ring (taking only the best path). When decoding with the tropical semi-ring, we would incorrectly take the back-off path if $P'(w_i|h_i) < \alpha_{bo}(h_i) \cdot P_{lower}(w_i|\bar{h_i})$[11]. As already said, luckily, this does not happen very frequently, and most often, this inconsistency for the tropical semi-ring is neglected. [Allauzen et al.(2003)] introduce an algorithm to obtain a back-off WFST G', that produces correct results in the tropical semi-ring, even when using ϵ-arcs.

An exact and deterministic implementation of back-off LMs with WFSA would require a different type of arc. The so called *failure arcs* were introduced for efficient string matching [Aho and Corasick(1975)]. Usually, in the literature (e.g. [Allauzen et al.(2003)]), a special arc label φ (or ϕ) is used to mark failure arcs. A failure arc doesn't consume any symbol and it has the semantic interpretation, that it can only be taken, if no other symbol on any of the other out-going arcs of the same state can be accepted. This works similar to the 'default' case in a C-language 'switch' statement. [Allauzen et al.(2003)] shows how to evaluate paths through WFSTs with failure arcs. These failure-arc-type WFSA accept sequences of words with exactly the same probabilities as when correctly implemented as a back-off LM in any of the LM tool-kits. The algorithm that we are going to develop in chapter 4 will work for both, ϵ- and failure-arc-type WFSA. Figure 2.13 shows an example of a tri-gram back-off LM implemented with failure arcs, and explains, why failure arcs contradict the Markov assumption.

As explained, there are correct solutions for WFSTs generated from back-off LMs, when working in the tropical semi-ring. This is true when taking the probabilities as estimated for a back-off LM or when converting an interpolated LM to a back-off LM. However, during the construction of the recognition network (section 2.4.1), we usually work in the (log-) probability semi-ring – most important are the determinization and weight pushing operations. When we compute the arc weights for back-off LMs, but implement the back-offs arcs with ϵ, the summation of the redundant back-off paths in the probability semi-ring actually has the consequence, that the probabilities of outgoing arcs do not sum to one, but to a slightly higher value, i.e. the resulting WFSA is no longer stochastic. Since the LM WFSA has cycles, the weight of a cycle can be greater than one, which causes the WFSA to have an infinite total weight (equation 2.5). As we will see in section 3.1, this can cause the conventional weight pushing algorithm to fail.

The correct solution would be to implement weight pushing (i.e. the shortest path algorithm) and the determinization for WFSA with failure arcs, i.e. respecting the semantics of failure arcs. As seen in figure 2.13, this is a non-trivial task, since failure arcs violate the Markov assumption, i.e. when following a failure arc, we have to remember the history of arcs to be able to choose the successor arcs correctly[12].

[11] This might be the case, if $P(w_1, w_2) \ll P(w_1)P(w_2)$.

[12] In the composite recognition network $HCLG$, the number of past arcs to be remembered can be quite high, since the arcs with word label can be farther apart.

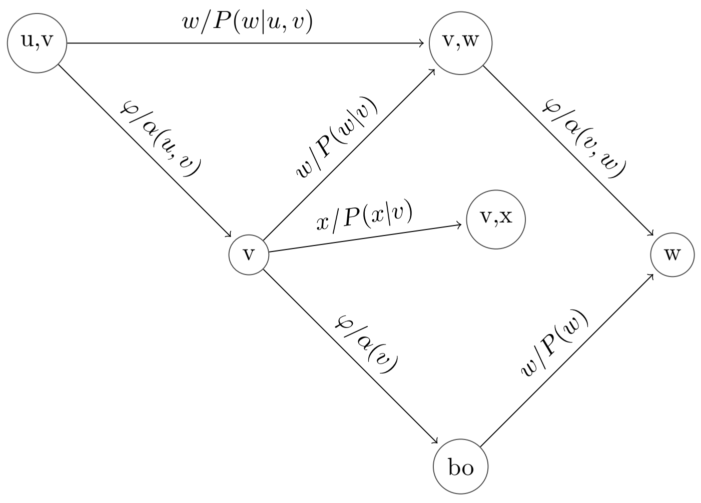


Figure 2.13: *Weighted finite state acceptor (WFSA) implementation of a tri-gram back-off LM using failure arcs. Only selected arcs are shown. A failure-arc, indicated by φ, doesn't consume any symbol and it can only be taken, if no other symbol on any of the other out-going arcs of the same state can be accepted. I.e. we can not take the arc $(v) \rightarrow (v, w)$, since there is a direct arc $(u, v) \rightarrow (v, w)$. Failure arcs have the peculiarity that the decision, which arc to take, is made based on the current symbol, but the symbol is only consumed later in the next non-failure arc. If there are several failure-arcs in a row (e.g. backing-off $(u, v) \rightarrow (v) \rightarrow (bo)$), we compare the same input symbol (w) several times against the outgoing arcs of different states (v and bo). Therefore, the decision to not take the arc $(bo) \rightarrow (w)$ is based on the fact, that there exists an arc $(u, v) \rightarrow (v, w)$. This contradicts the Markov property of the model, because after backing-off $(u, v) \rightarrow (v)$, due to the Markov assumption, it is not possible to decide in state bo, whether we came originally from (u, v) or from some other state (x, v), or just from (v). Therefore, to correctly implement the semantics of failure arcs, completely different algorithms are necessary.*

Since, to the best of our knowledge, determinization and shortest path algorithms for failure arcs in the probability semi-ring are not yet available, we conclude, that the correct way to construct recognition networks is to use interpolated LMs. However, we should not convert those probabilities to a back-off LM, as it is usually done. Then, we would compose the recognition network in the log-semi-ring, i.e. apply determinization and weight pushing in the log-semi-ring. Once the final recognition network is obtained, we apply the algorithm in [Allauzen et al.(2003)] (section "Exact offline representation") to obtain a WFST, that can be correctly used for decoding in the tropical semi-ring. For back-off LMs, the situation is worse. We would have to convert the back-offs LMs to an interpolated LM for the purpose of building and optimizing the recognition network in the log-semi-ring. Without going into details, this is not always possible. Therefore, we have to be aware of the fact, that when interpreted in the log-semi-ring (using ϵ arcs), these models are not stochastic and can have an infinite total weight. This has important consequences for the weight pushing algorithm (section 3.1). Therefore, we present an alternative weight pushing algorithm (section 3.3), that can handle this problem.

2.6 Parallel Speech Decoding

During the last decade, Moore's law, the trend of increasing clock rates by reducing transistor gate lengths, has slowed down and the power consumption of chips has become a mayor issue [Horowitz et al.(2005)]. There seems to be a Pareto relation [Horowitz et al.(2005)] between increasing performance and increased power consumption. Therefore, it is more efficient to run more units or cores in parallel at a lower clock speed instead of a single core running at higher clock speed. As a result, there is a need for parallel algorithms, which can efficiently use multiple cores that are usually present in recent systems.

The challenge in parallelization is to divide the task into sub-problems, that are as independent of each other as possible - to minimize the communication between the tasks and to avoid waiting times. At the same time, each of the sub-problems should be of approximately the same computational complexity to achieve a good load balancing and thus work efficiency. The maximum possible speed-up in parallelization is determined by the proportion of code that needs to be run serially (Amdahl's law). Depending on the size of the sub-problems that can be identified as being independent, we can distinguish coarse level and fine-grained parallelization. We think of fine-grained parallelizations as working on the level of individual states/arcs/densities or even on instruction level, while coarse refers to parallelism among decoding passes or chunks/segments of the utterance.

2.6.1 Coarse and fine-grained parallelization

Coarse level parallelization for LVCSR can be achieved, if different stages of processing or different knowledge sources are distributed to different cores. Speech recognition has several stages of processing (feature extraction, acoustic model evaluation, graph search), which can be distributed among cores. Another opportunity for coarse parallelization is the presence of multiple acoustic feature sets (acoustic models), where different feature streams can be computed on multiple cores. Coarse level parallelization has the advantage, that the serial algorithms do not need to be changed, i.e. no overhead due to communication and extra data structures is introduced. However, the scalability is limited by the number of available stages or feature streams and model components, which is usually small. The optimal distribution of computation tasks to multiple cores depends on the task (e.g. size of the recognition network, complexity of the acoustic model) and on the processor and memory configuration [You et al.(2009)]. A typical system can have a combination of several CPU cores with shared memory and a graphic processing unit (GPU), or multiple CPUs can be connected over a network. Two recent examples of systems using parallel CPUs and GPUs are given in figure 2.14 and in [Cardinal et al.(2013)].

In a single-threaded system, typically the majority of time is spent in the acoustic model evaluation (e.g. 80% in [You et al.(2009)]). At the same time, the acoustic model evaluation (either GMM or neural networks) is easily parallelizable (e.g. [Dixon et al.(2009)]). While the acoustic model evaluation can be easily parallelized in a fine-grained way [Dixon et al.(2009)], the fine-grained parallelization of the graph search is less trivial.

In dynamic programming algorithms (and SSSP), the sub-problems that do not depend on each other, and thus can be computed in parallel, form stages or wave-fronts [Maleki et al.(2014)] (see figure 2.5). There are efficient and scalable parallel algorithms to solve the general SSSP, many of them are based on the delta-stepping algorithm [Meyer and Sanders(2003)]. The idea is, that nodes are assigned to buckets and all nodes within a bucket are updated at the same time in parallel. Also the queue (and sorting) operations

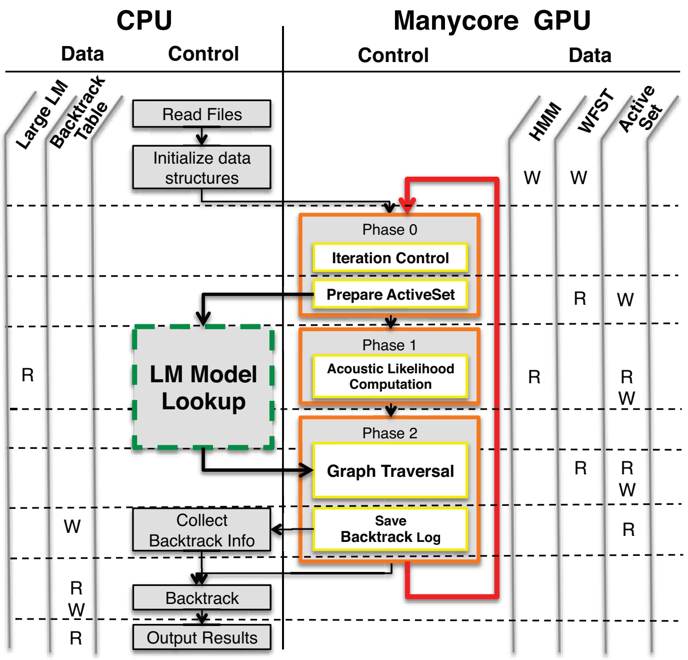


Figure 2.14: *LVCSR implementation on CPU and GPU [Kim et al.(2012)] – acoustic model evaluation and graph search is implemented on GPU, language model re-scoring due to memory requirements on CPU.*

can be parallelized. There are also recent implementations on GPUs, which are able to process huge graphs and achieve significant speed-ups [Davidson et al.(2014)].

While the general SSSP is well parallelizable, this not true for search in LVCSR. As already pointed out, the arc weights are computed dynamically (acoustic scores are added). Also, due to pruning, only a fraction of the states of the recognition network are kept in memory. Thus, the application of pruning changes the properties of the basic SSSP fundamentally, which makes it difficult to design a parallel algorithm for Viterbi search in LVCSR, that would be scalable to a high degree of parallelism and at the same time stays efficient.

Several parallelization attempts have been made with word-based HMMs and recognizers using linear lexica, however, for ASR systems with large vocabularies, a lexical-tree or more efficient WFST based decoders are desirable. [Phillips and Rogers(1999)] describe an WFST-based approach, that introduces fine-grained parallelism by organizing the data structures and grouping the computations according to a state of the recognition network. The computations belonging to each state are assigned to a core according to the modulo operation, which should achieve uniform load balancing. They achieved approximately 4x speed-up.

Another idea was pursued by [Parihar and Hansen(2008)]. They use a lexical-tree based decoder, i.e. using the lexicon transducer (uni-gram) and storing the word history with each token. Then, it is possible to split the lexical tree at the root into several sub-trees for each thread. To achieve a good load balancing, the split was done between similar sounding phones. However, the approach did not scale to more than 2-4 threads.

[You et al.(2009)] present an analysis on several algorithm designs to implement fine-grained parallelism for the Viterbi graph search. They showed, that the optimal algorithm design varies with the architecture (multi-core CPU vs. GPU). [Chong et al.(2009)] implemented a WFST based recognizer for a medium sized LVCSR completely on the GPU, which already achieved a 11x speed-up. However, in case huge language models of higher order (tri-gram and more) need to be used, the limited memory on the GPU would not be sufficient. To solve this, [Kim et al.(2012)] showed how to utilize the CPU in parallel for the language model on-the-fly re-scoring.

An alternative architecture is presented in [Cardinal et al.(2013)] - here, an A-star algorithm is used for search, and the computation of the heuristic is performed on the GPU. For that purpose a backward decoding with a uni-gram recognition network (LM look-ahead) is performed, which is parallelized by distributing transitions among cores according to the destination state (aggregation approach).

2.6.2 Stage parallelism through rank convergence

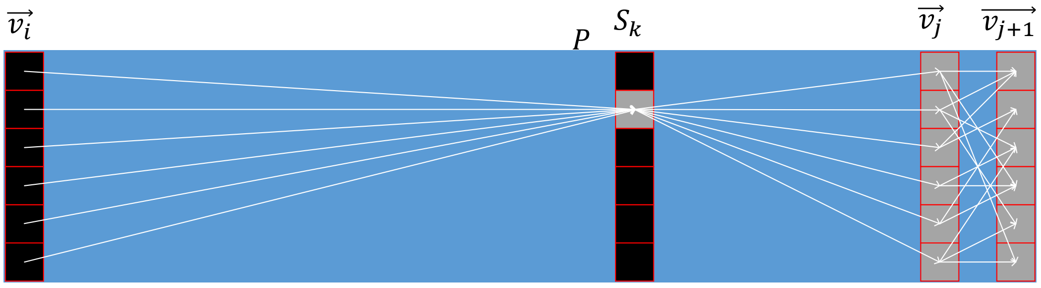


Figure 2.15: *Rank convergence in Viterbi algorithm [Maleki et al.(2014)]. $\vec{v_i}$ contains the scores of the states at time i. If at time k, the search converges to only one active state, then all future frames depend only on the score of this single state. Therefore, independent of the initialization at time i, the resulting vector $\vec{v_j}$ will be equal, except for an additive offset, which is constant for all its components. That means, the rank of state scores is independent of the initialization.*

An interesting observation concerning Viterbi decoding (and many FST based algorithms in general) was made by [Maleki et al.(2014)]: If we would start decoding in the middle of a sentence by assigning a random score to all states, usually, after a quite limited amount of time frames (20-50), the algorithm converges to a small set of active states, which is independent of the initialization at the start frame. We can interpret this as an all-pair-shortest-path problem [Cormen et al.(2009)], i.e. finding the shortest path between any pair of states in the graph. If we represent the possible transitions between states in one time step as a transition matrix, each time step of the Viterbi algorithm can be seen a matrix-matrix multiplication of the transition matrix in the tropical semi-ring. The observation is, that resulting all-pairs-shortest-path matrix (containing the score of paths between two states) will converge to a matrix of small orthogonal rank after several frames. We obtain the Viterbi forward score for the final time frame by multiplying the all-pairs matrix with an initial vector from the left. As seen in figure 2.15, if the rank of the all-pairs matrix is one, this leads to the situation that the Viterbi forward scores for all frames after the point of convergence are independent of the initialization vector (off by a constant).

This fact can be exploited to parallelize the Viterbi algorithm across stages - in other words to split it into time chunks which can be processed in parallel. If each randomly initialized chunk is long enough for the algorithm to converge to a single state at some point, then the state sequence after that is independent of the initialization and only the beginning frames of each chunk need to be fixed in a consecutive parallel fix-up phase.

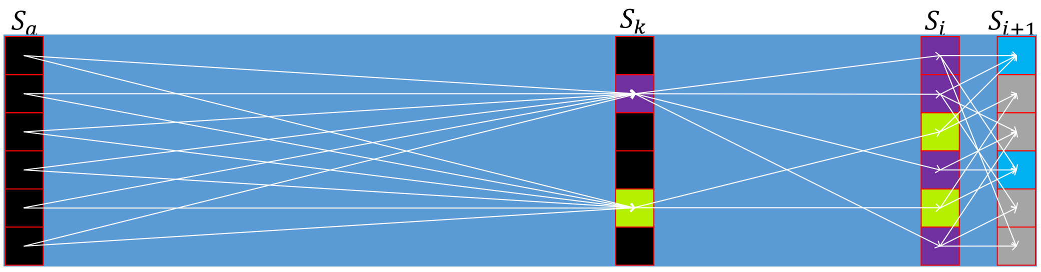


Figure 2.16: *Rank convergence in Viterbi algorithm with rank bigger than one. [Maleki et al.(2014)], unpublished. During the fix-up phase S_{i+1}, only the values of those input nodes S_i that originate from different active states (different color) need to be fixed.*

For small decoding tasks, this algorithm showed very promising results [Maleki et al.(2014)], but for LVCSR, the rank (number of active states) does usually not converge to one, but to a small number. In this case, the state scores will be linear combinations of vectors resulting from the few active states - see figure 2.16. A similar parallelization can be applied as in the singular case, but the fix-up phase gets slightly more complex. Using huge networks also makes it necessary to introduce state pruning. It is not clear, which states to activate during the random initialization. Therefore, the set of states in the fix-up phase might only be partly overlapping with the random-initialization phase, which complicates the fix-up phase of the algorithm.

An open research question is whether it is possible to automatically detect frames in advance, where the rank will converge, and what is the optimal segmentation into chunks for a given task. While we have no direct answer to that question, we suspect, that at the points with few remaining active states, the decoding results of a forward and backward search will agree (see chapter 5). Therefore, it should be possible to split the segments at points, where forward and backward search agree. This leads to an approach to parallelization, which is described in section 5.3.

Chapter 3

An alternative weight pushing algorithm

We explain the connection between Markov chains, non-negative matrices and weighted finite state acceptors (WFSA). Based on that, we introduce an alternative weight pushing algorithm, that is able to deal with possibly infinite total weight, and is much more efficient for acceptors with cycles. This alternative weight pushing algorithm is suitable for an important class of models - i.e. language model transducers or more generally (cyclic) WFSAs with failure transitions.

3.1 Weight pushing algorithm

As a prerequisite to this chapter, we assume, that we are able to construct a model (WFSA) that has some desired properties (i.e. being deterministic and minimal). The application we had in mind (chapter 4) was to construct a (back-off) language model (LM) acceptor, that has the desired size and structure and is deterministic (except for the ϵ-arcs). If we want to use the resulting acceptor in a pruned search (i.e. for LVCSR), it is desirable, that the acceptor has yet another property - to be (locally) *stochastic*.

Two WFSAs are equal, if they accept the same set of input label sequences with the same path weights. In other words, two equivalent WFSAs (or WFSTs) may differ by the way the weights (and output labels) are distributed along the path, even if they have the same topology with the same input labels [Mohri and Riley(2001)]. It was pointed out by [Mohri and Riley(2001)], that the distribution of weights along the path plays a crucial role in pruned search. Pruning is typically based on the weight accumulated along a path explored so far - often it is a combined weight (e.g. acoustic, pronunciation and language model for LVCSR). Typically, we prune by limiting the breadth of the search around the current best path (called beam pruning).

[Mohri and Riley(2001)] conjectured, that the optimal distribution of weights for pruned search should be such, that the weights (coming from different knowledge sources such as acoustic and language model) are locally synchronized for the sequential decisions, which arc to take next. Another common wisdom is, that the knowledge should be applied as early as possible in search - to be able to rule out unlikely paths as early as possible. This manifests itself in techniques like LM look-ahead [Ortmanns et al.(1996)], which are used in LVCSR decoding with dynamic networks. For statically compiled networks (or monolithic models like the LM WFSAs dealt with here), this corresponds to “pushing” the weights as much as possible towards the initial state. As shown in [Mohri et al.(2008)], pushing weights towards the initial state is actually equivalent to making the weights of the outgoing arcs of every state sum to one in the given semi-ring, i.e. making the WFSA stochastic. When the weights are distributed in such a way, the pruned search will be more effective - i.e. a smaller beam can be used. However, the overall best path (and accuracy) is still the same - in the asymptotic case of a very wide beam. [Mohri and Riley(2001)] show substantial speed-ups for several tasks in LVCSR, when modifying the transition probabilities of a WFSA in such a way, that the weights of paths through the WFSA form a stochastic distribution. Therefore, for optimal pruning in LVCSR with the probability semi-ring, we want to obtain a WFSA, where weights of outgoing arcs sum to one for each state.

We give a general definition of weight pushing for WFSAs, where we refer to the definition of a WFSA given in section 2.3. The generalization to WFSTs is given by interpreting weight-output label pairs as new weights combined by the appropriate semi-ring [Mohri(1997)][1].

Re-weighting [Mohri et al.(2008)] is an operation on WFSAs that alters the weights $w[t_i]$ of individual transitions and the final-probabilities $\rho(n[t_n])$, while leaving unaffected the weights $w[\pi]$ of successful paths (i.e. from initial to final states). The possible ways to change the transition weights of a WFSA can be expressed with the help of a *potential function* $V : Q \to \mathbb{K} - \bar{0}$, which can be an arbitrary function on states, assigning a value of $\mathbb{K}$ (except $\bar{0}$) to every state q. Given such a function, we can update the initial weight λ, the transition weights $w[e]$ and the final weights $\rho(f)$ according to the following [Mohri and Riley(2001)]:

$$\begin{aligned} \lambda &\leftarrow \lambda \otimes V(i) && (3.1)\\ \forall e \in E, w[e] &\leftarrow [V(p[e])]^{-1} \otimes (w[e] \otimes V(n[e])) && (3.2)\\ \forall f \in F, \rho(f) &\leftarrow [V(f)]^{-1} \otimes \rho[f] && (3.3) \end{aligned}$$

If the re-weighting is carried out this way, it is easy to see, that the overall weight of a successful path is not changed, since the potentials along any successful path cancel each other. Thus, the resulting WFSA is equivalent to the original one. The simplest possible re-weighting operation is to multiply $\otimes$ a fixed value k to the weights of all incoming arcs into a particular state q' and to divide $(^{-1})$ the same value from the weights of all arcs leaving that state. This is achieved with the potential function:

$$V(q) = \begin{cases} k \in \mathbb{K} - \bar{0} & \text{if } q = q' \\ \bar{1} & \text{otherwise} \end{cases} \tag{3.4}$$

Weight pushing is a special case of re-weighting, that aims to make the WFSA stochastic, or in other words to push the weights as much as possible towards the initial state. This is achieved [Mohri and Riley(2001)] by setting the potential function $V(q)$ to the *shortest distance* $d[q]$ from q to any of the final states F:

$$\forall q \in Q,\, V(q) = d[q] = \bigoplus_{\pi \in P(q)} w[\pi] \tag{3.5}$$

Here, $P(q)$ is the set of all paths from q to any of the final states F. Figure 3.1 shows an example of weight pushing in the tropical and probability semi-ring.

[1]OpenFST uses the Gallic semi-ring, which uses composite weights (ProductWeight) of an output label string and the arc weight. For the strings, we use the longest common prefix as $\oplus$ and concatenation as $\otimes$.

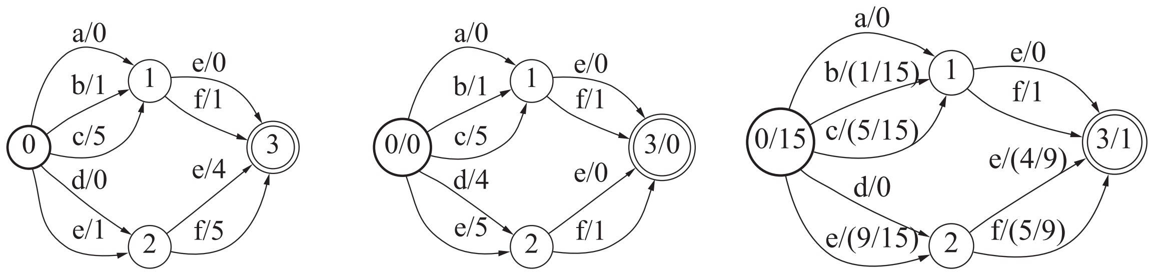


Figure 3.1: *Example of weight pushing [Mohri et al.(2008)]. Left: WFSA before applying the weight pushing. Center: Weight pushing in the tropical semi-ring ($\oplus$ is the minimum operation and $\otimes$ is addition). In this case, the potential function is the shortest distance to the final state, as computed by a Viterbi algorithm, that runs backwards from the final state. State 1 and 3 can be reached with zero cost, state 2 with cost 4. Thus, for each arc we $\otimes$ (add) the potential of the destination state and $^{-1}$ (subtract) the potential of the source state. Right: Weight pushing in the probability semi-ring. The potential function is the sum of all paths meeting in a state, as computed by the forward algorithm.*

There are several algorithms to compute the shortest distance, based on the dynamic programming principle, whose complexity depends on the semi-ring and the type of WFSA that is dealt with. For the tropical semi-ring, a Viterbi algorithm can be used. If the log-probability or probability semi-ring is used, however, all possible paths towards a state need to be summed, which is especially difficult, if the WFSA has cycles. A cycle can be followed an infinite amount of times, generating an infinite number of paths that need to be summed. So we need to guarantee that the weight of any cycle is $w(\pi) < \bar{1}$. In other words, we need to be able to compute the closure $\bigoplus_{i=1}^{\infty} w^i$, otherwise the cycle would result in an infinite total weight. If a semi-ring fulfills this condition for $\forall w \in \mathbb{K}$, it is called *closed semi-ring* - see [Mohri(2002)] for an exact general definition. If the structure of the WFSA is simple, i.e. the cycles are not nested and can be easily identified, the closure operation could be directly applied. For WFSAs resulting from LMs, this is not true, since the cycles are nested in a complex way.

In [Mohri(2002)], the set of *k-closed semi-rings* is introduced, which guarantees that the maximum number of times a cycle needs to be followed is k:

$$\forall a \in \mathbb{K}, \exists k : \bigoplus_{i=0}^{k+1} a^i = \bigoplus_{i=0}^{k} a^i = \bigoplus_{i=0}^{l} a^i, \forall l \geq k \tag{3.6}$$

For that class of semi-rings, figure 3.2 presents a generic shortest distance algorithm as given by [Mohri(2002)]. The algorithm manages a queue of states S that need to be updated. After extracting the state q from the queue, the so called *relaxation* operation (starting from line 10) consists in propagating the accumulated weight update $r[q]$ to all arcs e leaving the state. For all destination states $n[e]$ which meet the relaxation condition (line 11), i.e. the update is bigger than zero, we update the distance $d[n[e]]$ and the tentative update $r[n[e]]$ and add the state to the queue. The algorithm continues until the queue is empty. For k-closed semi-rings, the relaxation condition (line 11) makes an update unnecessary, if the weight of a loop has already been added k times. The algorithm is thus iterative and operates by locally forwarding weight mass through the WFSA according to the queue policy. The algorithm is efficient, if the number k is small, and if the state transition matrix is sparse, i.e. $|E| \ll |Q|^2$.

```
GENERIC-SINGLE-SOURCE-SHORTEST-DISTANCE $(G, s)$
 1  for $i \leftarrow 1$ to $|Q|$
 2      do $d[i] \leftarrow r[i] \leftarrow \overline{0}$
 3  $d[s] \leftarrow r[s] \leftarrow \overline{1}$
 4  $S \leftarrow \{s\}$
 5  while $S \neq \emptyset$
 6          do $q \leftarrow head(S)$
 7             DEQUEUE$(S)$
 8             $r' \leftarrow r[q]$
 9             $r[q] \leftarrow \overline{0}$
10             for each $e \in E[q]$
11                 do if $d[n[e]] \neq d[n[e]] \oplus (r' \otimes w[e])$
12                       then $d[n[e]] \leftarrow d[n[e]] \oplus (r' \otimes w[e])$
13                            $r[n[e]] \leftarrow r[n[e]] \oplus (r' \otimes w[e])$
14                            if $n[e] \notin S$
15                               then ENQUEUE$(S, n[e])$
16  $d[s] \leftarrow \overline{1}$
```

Figure 3.2: *Pseudocode of single-source shortest path algorithm used in the generic weight pushing algorithm [Mohri(2002)]. The algorithm computes the shortest distance $d[q]$ from the initial state s for each state q of the WFSA G. To compute the shortest distance to the final state, we have to start with the final state and follow the arcs in the opposite direction.*

For the (log-) probability semi-ring, there is no $k < \infty$, for which equation 3.6 would hold, but it is still a closed semi-ring ($k \to \infty$). The generic shortest-distance algorithm can't be used in this case, however, closed semi-rings are covered by the generic Floyd-Warshall and Gauss-Jordan algorithms [Lehmann(1977)]. These algorithms solve the so called *algebraic path problem* by computing the all-pair shortest distance with a time complexity of $O(n^3)$ (n proportional to the number of states), but they are not as efficient for our purpose, since they don't take advantage of the sparsity of the transition matrix and since we are actually just interested in the distance from one single (start) state. As soon as the WFSA has thousands of states, an algorithm with complexity $O(n^3)$ is clearly not feasible.

One strategy is to decompose the WFSA into several *strongly connected components*, where any state of a component is reachable by any other state of the same component by a path of limited length. In this case, the all-pair shortest distance only needs to be computed for each component separately. However, it is relatively easy to see, that a model of language such as the N-gram consists basically of just a single huge strongly connected component. Since the history is limited to a few previous words, and even completely erased on sentence boundaries, it is obvious that in principle any sequence of words is repeatable after a limited amount of time steps. Thus, the complexity of the Floyd-Warshall algorithm cannot be reduced this way.

The original relaxation condition for the generic algorithm (figure 3.2, line 11) is given by:

$$d[n[e]] \neq d[n[e]] \oplus (r' \otimes w[e]) \tag{3.7}$$

where d is the distance and r' is the tentative update to be propagated, i.e. the weight accumulated since the last relaxation of $q = p[e]$. Thus, $(r' \otimes w[e])$ is the weight to be added in the relaxation. To handle also semi-rings that are not k-closed, [Mohri(2002)] replaces the relaxation condition by an approximate test with a metric Δ:

$$\Delta(d[n[e]], d[n[e]] \oplus (r' \otimes w[e])) \geq \delta \tag{3.8}$$

where $\delta \geq \bar{0}$ is a positive number used for approximation. For the probability semi-ring, the condition simplyfies to $(r' \times w[e]) \geq \delta$. Due to limited machine precision, there is actually always some δ for which this condition will not be met. Thus, with a k-closed semi-ring, a cycle will not be followed more than k times, and in our case the algorithm stops updating as soon as:

$$l : \Delta\left(\bigoplus_{i=0}^{l} w[\pi]^i, \bigoplus_{i=0}^{l-1} w[\pi]^i\right) = w[\pi]^l \leq \delta \tag{3.9}$$

where $w[\pi]$ is the weight of the cycle. If l is large ($w[\pi] \to \bar{1}$ or $\delta \to \bar{0}$) the algorithm will iterate for a long time until it converges. For $w[\pi] \geq \bar{1}$, the algorithm fails to converge at all. In this case, the total weight of the WFSA becomes infinity.

When we apply the weight pushing algorithm to WFSA that are constructed from back-off language models, we have to distinguish two cases: As explained in section 2.5.1 and figure 2.11, the ϵ-style back-off arcs lead to duplicate paths. In case the weights were taken from a back-off LM estimated for failure arcs, but the back-off arcs are represented with ϵ, the outgoing arcs will not exactly sum one, but to a slightly higher value. When occurring in a loop, the condition $w[\pi] < \bar{1}$ does no longer hold. That means, the generic weight pushing algorithm [Mohri(2002)] as implemented in OpenFST will fail to converge, because the total weight of the entire WFSA will not be finite. If the weights are correctly estimated as interpolated LM, the representation with ϵ arcs results in a stochastic WFSA. However, the weight in cycles can still be very close to one, so that the generic algorithm is inefficient (equation 3.9).

3.2 Ergodic Markov chains and non-negative matrices

In the previous section, we explained why we want a model that has a (locally) stochastic weight distribution, and why for WFSA, that are cyclic, the standard weight pushing algorithm [Mohri and Riley(2001)] is either inefficient or completely fails to converge[2], if the total weight of the WFSA is not finite. For this purpose, we need a weight pushing algorithm that will always succeed. We show here, that this problem can be solved, if we represent the WFSA as an ergodic Markov chain.

Here, we deal with Markov chains, which are, by definition, already stochastic, and thus don't need weight pushing. However, they will serve us for the purpose of introducing important concepts and the basic idea of our algorithm. Later, we generalize to non-negative matrices. A *Markov chain* [Grinstead and Snell(1997)] can be defined similar to the WFSA (equation 2.3), but discarding the labels Σ. To have a more flexible definition, every state can be a potential initial state:

$$M = (Q, F, E, \lambda) \text{ over a semi-ring } (\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1}). \tag{3.10}$$

It is given by:

[2]The algorithm for weight pushing in the log semi-ring provided with OpenFST `www.openfst.org` might still work for some smaller models (e.g. LMs with small vocabulary), if the delta parameter is chosen to be sufficiently small. We observed that typically as soon as the WFSA contains states with a huge fan-out ($\gg 1000$), the algorithm fails to converge.

- a set of states Q
- a set of final or absorbing states $F \subseteq Q$
- a set of transitions $E \subseteq Q \times \mathbb{K} \times Q$
- an *initial weight* λ

A transition $t = (p[t], w[t], n[t]) \in E$ is seen as a move (step) from the source state $p[t]$ to the destination state $n[t]$ with weight $w[t]$. Markov chains were introduced with probabilities as weights, thus, in this section, we only consider the probability semi-ring $(\mathbb{R} \in [0,1], +, \times, 0, 1)$. Usually, the transition weights are represented in form of the transition matrix $\mathbf{P}_{i,j} \in |Q| \times |Q|$, where each entry $p_{ij} = w[t]$ contains the weight of the transition t from state $i = p[t]$ to state $j = n[t]$. If the corresponding transition doesn't exist, the entry is $\bar{0}$. In place of the initial weight λ, one can add a super-initial state i with outgoing weights defined by λ – as done in the WFSA definition (equation 2.3). Instead of having a final weight ρ as in equation 2.3, a Markov chain can be seen as having a super-final state f' and transitions from $f \in F$ to f' with $p_{f,f'} = \rho[f]$. The literature on Markov chains doesn't use the term final state, but instead uses the term *absorbing state*, which is a state that cannot be left, i.e. it has a self-loop probability of one. All other states all called *transient*. A Markov chain is absorbing if it has at least one absorbing state and from every state it is possible to reach an absorbing state. If we represent an WFSA as Markov chain by using the sum of all arcs from state i to state j as entry p_{ij} in the transition matrix $\mathbf{P}$, we see, that every trim (connected) WFSA corresponds by definition to an absorbing Markov chain.

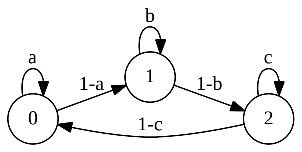
Representing the transition weights in a matrix has the advantage, that we can elegantly compute the outcome of a process after several steps: The ijth entry $p_{ij}^{(n)}$ of the matrix $\mathbf{P}^n$ (n-th power of the matrix) gives the probability that the Markov chain, starting in state q_i, will be in state q_j after n steps [Grinstead and Snell(1997)]. The function λ with the probability of starting in a particular state can be represented as a state probability vector $\boldsymbol{\lambda}$. Similar, if $\mathbf{v}$ is the row vector with elements v_i representing the probability of being in state q_i at a certain time n, then:

$$\mathbf{v}^{(n)} = \boldsymbol{\lambda}\,\mathbf{P}^n \tag{3.11}$$

An important class of Markov chains are *ergodic Markov chains* [Grinstead and Snell(1997)], also called *irreducible*. A Markov chain is ergodic, if it is possible to go from any state to any state (not necessarily in one move). In this case, the corresponding WFSA consists only of one strongly connected component, and there are no absorbing final states. An important sub-class of ergodic Markov chains are *regular chains* (also called primitive). A Markov chain is called a regular chain if any positive power of the transition matrix $\mathbf{P}^n$ has only positive elements[3]. In other words, for some n, it is possible to go from any state to all other states (including self-loop) in exactly n steps. Every regular chain is ergodic, but not all ergodic chains are regular - see the example in figure 3.3.

An absorbing Markov chain is not ergodic. This holds for WFSAs, which are absorbing Markov chains with only the final states being absorbing states. However, if the WFSA is trim (every state can be reached on a successful path), we can make the WFSA ergodic by connecting the final states $f \in F$ to the initial state i. Now, every state can be reached from any other state, by going over any of the final states.

[3]When talking about probabilities, this means not zero.



$$\mathbf{P} = \begin{bmatrix} a & 1-a & 0 \\ 0 & b & 1-b \\ 1-c & 0 & c \end{bmatrix}$$

Figure 3.3: *Example of an ergodic Markov chain and its corresponding transition matrix. If $a > 0 \wedge b > 0 \wedge c > 0$, then it can be easily shown that already for $\mathbf{P}^2$ ($n = 2$) all $p_{ij}^{(2)} > 0$, so that the Markov chain is also regular. That means any state can be reached from any state with a maximum of two steps. For that reason, with $n \to \infty$, the state distribution approaches an equilibrium, according to the proportion of a, b and c. If $a = b = c = 0$ (removing the self-loops), the chain is no longer regular. It is still obvious that every state can be reached from any other state, but the matrix $\mathbf{P}$ becomes a permutation matrix, which means that the state distribution oscillates between three different configurations, but never converges. From this example, it is easy to see, that adding self-loops, i.e. increasing the values on the diagonal makes an ergodic chain a regular one.*

In a next step, if $\mathbf{P}$ is the transition matrix of an ergodic Markov chain, then we can obtain the transition matrix of a regular chain by:

$$\mathbf{P}' = k\,\mathbf{I} + (1-k)\,\mathbf{P}\,,\; 0 < k < 1,\; k \in \mathbb{R} \tag{3.12}$$

Since the ergodicity guarantees, that every state can be reached, interpolating with the identity matrix $\mathbf{I}$ guarantees, that the diagonal elements $p'_{ii} > 0$ are positive, which means, that it is possible to take self-loops to stay in every state. Thus, after n steps, when all states of the ergodic chain have been reached, $\mathbf{P'}^n$ will have all elements positive[4]. It is easy to see, that $\mathbf{P}'$ and $\mathbf{P}$ have the same eigenvectors $\mathbf{v}$[5] ($\mathbf{P}'\mathbf{v} = k'\mathbf{v}$):

$$0 = \left(\mathbf{P}' - k'\,\mathbf{I}\right)\,\mathbf{v} = \left(k\,\mathbf{I} + (1-k)\,\mathbf{P} - k'\,\mathbf{I}\right)\,\mathbf{v} = (1-k)\,\left(\mathbf{P} - \frac{k'-k}{1-k}\,\mathbf{I}\right)\,\mathbf{v} \tag{3.13}$$

The fundamental limit theorem for regular chains [Grinstead and Snell(1997)] says that if $\mathbf{P}$ is the transition matrix of a regular Markov chain, then with $n \to \infty$, the powers $\mathbf{P}^n$ approach a limiting matrix $\mathbf{W}$ with all rows containing the same vector $\mathbf{w}$ where all components of $\mathbf{w}$ are positive and sum to one:

$$\mathbf{W} = \lim_{n\to\infty} \mathbf{P}^n. \tag{3.14}$$

This states that the probability of being in state q_j (the jth entry of $\mathbf{v}$) in the long run is independent of the starting state q_i ($v_j \to w_j$). From this, it follows that $\mathbf{w}\,\mathbf{P} = \mathbf{w}$, and any row vector $\mathbf{v}$ with $\mathbf{v}\,\mathbf{P} = \mathbf{v}$ is a constant multiple of $\mathbf{w}$.

The unique normalized vector $\mathbf{w}$ is called *fixed row vector* and represents the *stationary distribution* of the process. In other words, there is just one stationary distribution, i.e. only one *left eigenvector* corresponding to the eigenvalue one[6], that solves the equation $\mathbf{v}\,\mathbf{P} = \mathbf{v}$. From equation 3.14, it follows that for any initial probability vector $\boldsymbol{\lambda}$, the process approaches the fixed row vector $\mathbf{w}$ for $n \to \infty$:

[4]Once a state is entered with some probability, the non-zero self-loop guarantees that is possible to stay in all successive time steps.

[5]If $k \ll 1$ or $k' \to 1$, then also the eigenvalue will be very similar.

[6]There can be other eigenvectors, whose eigenvalue (absolute value) are smaller than one, which will vanish with $\lim_{n\to\infty} \mathbf{P}^n$.

$$\lim_{n\to\infty} \boldsymbol{\lambda}\,\mathbf{P}^n = \boldsymbol{\lambda}\,\mathbf{W} = \mathbf{w}. \tag{3.15}$$

Given equation 3.12, we can convert every ergodic chain into a regular chain with the same eigenvector. Thus it is also clear, that there is only one strictly positive fixed vector for ergodic Markov chains. However, this fixed vector has a slightly different interpretation [Grinstead and Snell(1997)]:

$$\mathbf{A}_n = \frac{\mathbf{I} + \mathbf{P} + \mathbf{P}^2 + \ldots + \mathbf{P}^n}{n+1}, \lim_{n\to\infty} \mathbf{A}_n = \mathbf{W}, \tag{3.16}$$

where $\mathbf{W}$ is a matrix, all of whose rows are equal to the unique fixed probability vector $\mathbf{w}$ for $\mathbf{P}$. Therefore, the ijth entry of the matrix $\mathbf{A}_n$ gives the expected value of the proportion of times that the process is in state q_j in the first n steps, when starting from state q_i. As already seen in figure 3.3, for ergodic Markov chains that are not regular, the state distribution doesn't converge. However, the state distribution averaged over time does converge. The *law of large numbers* for ergodic Markov chains [Grinstead and Snell(1997)] states, that the proportion of times that an ergodic chain is in state q_j in n steps - $H_j(n)$ - is independent of the starting state q_i:

$$P\left(|H_j(n) - w_j| > \epsilon\right) \to 0, \forall \epsilon > 0. \tag{3.17}$$

So far, we were dealing with regular Markov chains, i.e. we assumed that $\mathbf{P}$ is a *row-stochastic matrix*, where every row sums to one. Now, we want to generalize equation 3.14 to the case, where $\mathbf{P}$ is not normalized. A generalization of the results on Markov chains is given within the theory of non-negative matrices [Berman and Shaked-Monderer(2012)]. A *non-negative matrix* is a matrix with all entries $p_{ij} \geq 0$. In the same way as the transition matrix $\mathbf{P}_{ij}$ of Markov chains, every non-negative matrix can be associated to a directed graph, with the only difference, that the transition weights are no longer limited to be in the interval $[0, 1]$ and that the matrix is not required to be row-stochastic. The Perron theorem [Berman and Shaked-Monderer(2012)] states, that for every non-negative primitive (i.e. regular) matrix $\mathbf{P}$, the maximum eigenvalue $\rho(\mathbf{P})$ (also called spectral radius) is positive, simple[7], singular (only one eigenvalue of this modulus) and has a positive eigenvector (called left and right Perron vector, whose normalized entries sum to one). For $n \to \infty$, the matrix converges:

$$\lim_{n\to\infty} \left(\frac{\mathbf{P}}{\rho(\mathbf{P})}\right)^n = \mathbf{L},\ \mathbf{L} = \mathbf{x}^T\,\mathbf{y},\ \mathbf{x}\,\mathbf{y}^T = 1, \tag{3.18}$$

where $\mathbf{x}$ and $\mathbf{y}$ are positive right and left eigenvectors: $\mathbf{P}\,\mathbf{x}^T = \rho(\mathbf{P})\,\mathbf{x}^T,\ \mathbf{x} > 0,\ \mathbf{y}\,\mathbf{P} = \rho(\mathbf{P})\,\mathbf{y},\ \mathbf{y} > 0$. From this, it follows that for any initial probability vector $\boldsymbol{\lambda}$, the ratio of state weights in $\boldsymbol{\lambda}$ will converge[8] to a vector proportional to the left eigenvector $\mathbf{y}$:

$$\lim_{n\to\infty} \boldsymbol{\lambda} \left(\frac{\mathbf{P}}{\rho(\mathbf{P})}\right)^n = \boldsymbol{\lambda}\,\mathbf{L} = (\boldsymbol{\lambda}\,\mathbf{x}^T)\,\mathbf{y} = c\,\mathbf{y}. \tag{3.19}$$

[7]The algebraic multiplicity is one. Algebraic multiplicity is the number of times an eigenvalue appears in the characteristic polynomial of a matrix.

[8]We don't know $\rho(\mathbf{P})$, but independent of the normalizing constant, the ratio of the components of vector $\boldsymbol{\lambda}$ will converge to the ratio in the left eigenvector. To achieve numerical stability, we need to normalize, and we could normalize $\boldsymbol{\lambda}$ to unit length or to the first component being one.

Similarly, for any initial probability vector $\boldsymbol{\lambda}$, if we multiply from the right, the ratio of state weights in $\boldsymbol{\lambda}$ will converge to a vector proportional to the right eigenvector $\mathbf{x}^T$:

$$\lim_{n\to\infty}\left(\frac{\mathbf{P}}{\rho(\mathbf{P})}\right)^n \boldsymbol{\lambda}^T = \mathbf{L}\,\boldsymbol{\lambda}^T = \mathbf{x}^T\,(\mathbf{y}\,\boldsymbol{\lambda}^T) = d\,\mathbf{x}^T. \tag{3.20}$$

If the matrix $\mathbf{P}$ is not regular, but non-negative and irreducible (ergodic), the Perron-Frobenius theorem [Berman and Shaked-Monderer(2012)] states, that the maximum (absolute value) eigenvalue $\rho(\mathbf{P})$ is still positive, simple (algebraic multiplicity one) and has a positive eigenvector (called Perron vector). There are no non-negative eigenvectors for $\mathbf{P}$ except for multiples of the Perron vector. All of them have eigenvalues with modulus $\rho(\mathbf{P})$, however, there can be several complex eigenvalues with this maximum modulus. The eigenvalues with modulus $\rho(\mathbf{P})$ are $\rho(\mathbf{P})\,e^{2\pi il/k}$ with $l = 0, 1, \ldots, k-1$ and k is called the *index of cyclicity*.

With the help of a permutation matrix $\mathbf{R}$, every non-negative matrix $\mathbf{P}$ can be converted into the Frobenius form:

$$\mathbf{R}^T\,\mathbf{P}\,\mathbf{R} = \begin{pmatrix} \mathbf{0} & \mathbf{P}_{12} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{P}_{23} & \ldots & \mathbf{0} \\ \vdots & & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{P}_{k-1k} \\ \mathbf{P}_{k1} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \end{pmatrix}, \tag{3.21}$$

where the $\mathbf{0}$-matrices on the diagonal are square. This means, that every non-negative matrix with index of cyclicity $k > 1$ (i.e. ergodic but not regular) corresponds to a directed graph, whose states can be clustered into k stages, where the states of stage $l+1$ can only be reached by the states of stage l. This can be seen in the example in figure 3.3. When $a = b = c = 0$, there are three alternating stages with one state each.

3.3 Alternative weight pushing algorithm

In section 3.1, we motivated the need to make the WFSA stochastic through the use of a weight pushing algorithm. We showed, that for the (log-) probability semi-ring, the generic (exact) algorithm for k-closed semi-rings is not applicable, and the generic algorithm for closed semi-rings (all-pair-shortest-path) is not feasible for WFSAs with big strongly connected components (which is the case for WFSAs based on N-gram LMs). We showed, that the convergence of the approximate iterative algorithm depends on the weight in a loop (equation 3.9). If the weight of a loop $\omega[\pi] \geq 1$ (or the sum of several loops meeting in the same state), the algorithm fails to converge. As already explained, this can be the case for ϵ-style back-off LMs, when the weights were taken from a back-off LM (section 2.5.1). Thus, we need a weight pushing algorithm, that will also succeed for those kinds of WFSAs.

We represent the WFSA in the probability semi-ring[9] by using the transition matrix $\mathbf{P}_{ij}$, where p_{ij} is the sum of the weight of all transitions between state i and state j. The transition matrix is usually sparse (contains $\bar{0}$ for all non-existing transitions). Our solution is based on the theory of non-negative matrices and ergodic Markov chains as introduced in section 3.2. The fundamental limit theorem for regular chains in equation 3.15 suggests an

[9]Even for the implementation, we found it more convenient to use actual probabilities instead of negative logs, as used in the log-semi-ring.

iterative algorithm to find the stationary distribution. This is similar to the power method for finding the dominant eigenvector $\mathbf{w}$ of the matrix $\mathbf{P}$, by starting from a random or uniform positive vector $\mathbf{v}$ and iterating by letting $\mathbf{v} \leftarrow \mathbf{P}\,\mathbf{v}$ each time.

If the WFSA is not normalized, the generalization is given by the Perron theorem in equation 3.18. Since we do not know the normalizing spectral radius $\rho(\mathbf{P})$ in advance, we re-normalize the resulting vector $\mathbf{v}$ at each step so that v_I is 1, where I is the initial state of the WFSA[10]. By equation 3.19, we know that if we iterate $\mathbf{u}^T \leftarrow \mathbf{u}^T\,\mathbf{P}$, we converge to a multiple of the dominant *left* eigenvector $\mathbf{y}$. This corresponds to a multiple of the stationary distribution $\mathbf{w}$ of the normalized chain and also corresponds to the minimum distance from the initial state in the probability semi-ring. The law of large numbers for Markov chains (equation 3.17) tells us, that the elements of this vector correspond to the average proportion of times that the chain is in each of the states in the long run. If we instead iterate $\mathbf{v} \leftarrow \mathbf{P}\,\mathbf{v}$ (equation 3.20), it results in the dominant *right* eigenvector of $\mathbf{P}$, which, in the probability semi-ring, is the minimum distance towards the final states (or the super-final state).

The Perron theorem is only true for regular chains, but as explained in section 3.2, we can make every trim WFSA ergodic by connecting the final states $f \in F$ to the initial state I[11]. That means we modify one column in the transition matrix: if j is the initial state, then p_{ij} is set to the final-probability $\rho[i]$ of state i. As a second step, we need to guarantee, that the resulting ergodic WFSA is also regular[12]. We can make the WFSA regular by interpolating $\mathbf{P}$ with the identity matrix (equation 3.12). Alternatively, we can modify the iteration to $\mathbf{v} \leftarrow \mathbf{P}\,\mathbf{v} + k\,\mathbf{v}$[13]. The parameter k is set to a small value (0.1) to not slow down the convergence too much. This algorithm is very efficient in practice, it generally converges within several tens of iterations.

At the end, we have a vector $\mathbf{v}$ with $v_I = 1$, and a scalar $c > 0$, such that

$$c\,\mathbf{v} = \mathbf{P}\,\mathbf{v}. \tag{3.22}$$

The vector $\mathbf{v}$ contains the distribution of average state occupancies and is used as the potential function $V(q) : Q \to \mathbb{K} - \bar{0}$ for the re-weighting operation (equation 3.2). This means we compute a modified transition matrix $\mathbf{P}^*$, by letting

$$p^*_{ij} = p_{ij}\,v_i/v_j, \tag{3.23}$$

and transforming the final probabilities by $\rho^*_i = \rho_i\,v_I/v_i$, where v_I is the potential of the initial state. Using the re-weighting with the potential function V guarantees, that the resulting WFSA is equivalent to the original one.

If we apply the *left* Perron eigenvector as potential function in the re-weighting operation (equation 3.2), it results in pushing the weight towards the final state, or more precisely in making the WFSA input stochastic. That means, either all incoming arcs sum to one, if the total weight is one, or more generally they sum to the same quantity for all states. If we use the *right* Perron eigenvector as potential function in the re-weighting, it results in

[10] Any normalization will lead to the same eigenvector. Actually, in the implementation, we normalize to unit length, as in the matrix power algorithm.

[11] This acts like an arc from the super-final state to the initial state with probability one. It will not change an already stochastic WFSA, since all components of the resulting eigenvector will be equal.

[12] That is, we want index of cyclicity one. This is, for example, necessary for linear WFSAs (in figure 3.3), which have several multiple eigenvalues with the same magnitude but different complex phases.

[13] More exact would be $(1-k)\,\mathbf{P}\,\mathbf{v} + k\,\mathbf{v}$, but it doesn't affect the result.

pushing the weights towards the initial state and in making the WFSA output stochastic - i.e. all outgoing arcs sum to one (or more generally to the same quantity).

We show this by writing one element of equation 3.22 as

$$c\,v_i = \sum_j p_{ij}\,v_j, \tag{3.24}$$

by dividing by v_i, it easily follows that $c = \sum_j p^*_{ij}$. This means each row of the modified matrix $\mathbf{P}^*$ sums to c (modulus of the eigenvalue of the Perron vector). In the classical weight pushing algorithm [Mohri and Riley(2001)], we assume a stochastic WFSA (equation 2.6), so that after weight pushing, all outgoing transitions of a state "sum to" $\bar{1}$ in the given semi-ring. Our solution is to use a modified pushing operation, which results in a WFSA, for which the transitions out of all states (and the final probability ρ) "sum to" the same quantity c:

$$\forall q \in Q, \left(\bigoplus_{e\in E[q]} w[e]\right) \oplus \rho[q] = c. \tag{3.25}$$

This means that the left-over weight, which is usually added to the initial or final states and which can cause the standard algorithm to fail, is now uniformly "smeared" all over the WFSA.

Algorithm 1 gives the pseudo-code of the alternative weight pushing algorithm[14]. The main program consists of the flat initialization of the vector $\mathbf{v}$, then we iterate until convergence (*Iterate*) and finally apply the reweighting operation (*ModifyFst*). The heart of the algorithm is the function *Propagate*, which is both used in the test for stochasticity (*TestAccuracy*) and in the main iterative algorithm (*Iterate*). The only difference is, whether the outgoing arcs are reweighted with the potential function ($prob \cdot \mathbf{v}[d]/\mathbf{v}[s]$) or the propagated probability mass is summed from the destination states of the outgoing arcs ($prob \cdot \mathbf{v}[d]$). Notice also the symmetry of pushing towards the final or towards the initial state – we only have to switch the role of source and destination state[15]. The stopping criterion in *TestAccuracy* is the ratio between the maximum and minimum arc sum – which should converge to one.

Our algorithm is in practice an order of magnitude faster than the more generic algorithm for conventional weight-pushing [Mohri and Riley(2001)], when applied to cyclic WFSAs. The speed of the algorithm is determined by the convergence of the matrix power method, iterating by repeatedly multiplying the state distribution vector with the (sparse) transition matrix, i.e. going through all states in a pre-defined order every time. The convergence of the matrix power method depends on the ratio between the biggest $\rho(\mathbf{P})$ and the second biggest eigenvalue.

Mohri's algorithm (3.2) is similar to a backward (Viterbi-like) algorithm on the given semi-ring, using the new relaxation condition (equation 3.8) to propagate the probability mass and to update the queue. A state is put to the queue, if the accumulated probability mass has changed more than the delta parameter since the last visit. The queue is processed according to the queue policy until it is empty. For the queue method, the proofs for convergence of the matrix power algorithm can no longer be applied.

[14]The algorithm assumes that the weights were stored in the log-semi-ring.

[15]We check if the log-ratio is below the threshold. We do not test at each iteration, to save the time of re-weighting the arcs.

Algorithm 1 *Pseudo-code of the alternative weight pushing algorithm.*

```
void Main() {
  vector<double, num_states> v = 1 / sqrt(num_states) // flat init
  for (all states s):
    for (all arcs arc leaving s):
      prob = exp(-arc.weight) // convert to probability semi-ring
      predecessors[t].add( tuple(s, prob) )
    final = exp(-finalweight(s))
    predecessors[initial_state].add( tuple(s, final) ) // force ergodicity
  Iterate(backwards, delta) // until stationary distribution found
  ModifyFst(backwards)      // weight pushing operation
}

double TestAccuracy(bool backwards) { // test stochasticity
  vector<double, num_states> state_sums =
                    Propagate(backwards, do_reweighting)
  return log( max (state_sums) / min (state_sums) )
}

void Iterate(bool backwards, delta) {
  for (maximal 2000 iterations):
    vector<double, num_states> new_v =
                     Propagate(backwards, not_reweighting)
    new_v += 0.1 * v // regular, using power method (M + 0.1*I)
    v = new_v / sqrt(new_v dot new_v) // renormalize with L2-norm
    if (test_iteration and TestAccuracy() <= delta):
      return // has converged
  output warning: Did not converge!
}

void ModifyFst(bool backwards) {   // weight pushing operation
  v = -log(v) // convert to log-probability
  for (all states s):
    for (all arcs arc leaving s):
      if (backwards)
        arc.weight = arc.weight + (v[t] - v[s]) // outgoing normalization
      else // forwards
        arc.weight = arc.weight + (v[s] - v[t]) // incoming normalization
    if (backwards)
      finalweight(s) = finalweight(s) + (v[initial_state] - v[s])
    else
      finalweight(s) = finalweight(s) + (v[s] - v[initial_state])
}

vector<double> Propagate(bool backwards, bool reweighting) {
  for (all states d):
    for (all (state s, prob) in predecessors[d]):
      if (backwards) {
        if (reweighting)
          state_sums[s] += prob * v[d] / v[s]
        else
          state_sums[s] += prob * v[d] // v_j = sum_i v_i * p_ji.
          // summing in the source state -> distance to the final_state
          // pushing the weights towards the initial state
      } else { // forwards
        if (reweighting)
          state_sums[d] += prob * v[s] / v[d]
        else
          state_sums[d] += prob * v[s] // v_i = sum_j v_j * p_ji
          // summing in destination state -> distance to initial state
          // pushing the weights towards the final_state
      }
  return state_sums
}
```

Apart from that, both algorithms use different initial distributions. Mohri's algorithm starts from the initial state, the new algorithm uses a uniform or a random vector as initial state distribution. Also, a different test for convergence is used. Mohri's algorithm uses the delta relaxation and the new algorithm checks the minimum and maximum over all states of the sum of the outgoing weights of a state. However, both differences can be considered minor.

One issue with the alternative weight pushing algorithm is that it was derived under the assumption, that all arcs in the WFST are of the same type. The transition matrix P_{ij} treats all arcs in the same way. The LM transducer, which accepts/emits sequences of words, actually consists of arcs with a word label, and arcs representing the back-off arcs, that don't accept any symbol. Therefore, there are multiple paths through the model with different number of arcs to accept the same word sequence. We can see this in analogy to composite HMMs, which have emitting and non-emitting arcs, which are computed in two separate steps in the forward/Viterbi algorithm.

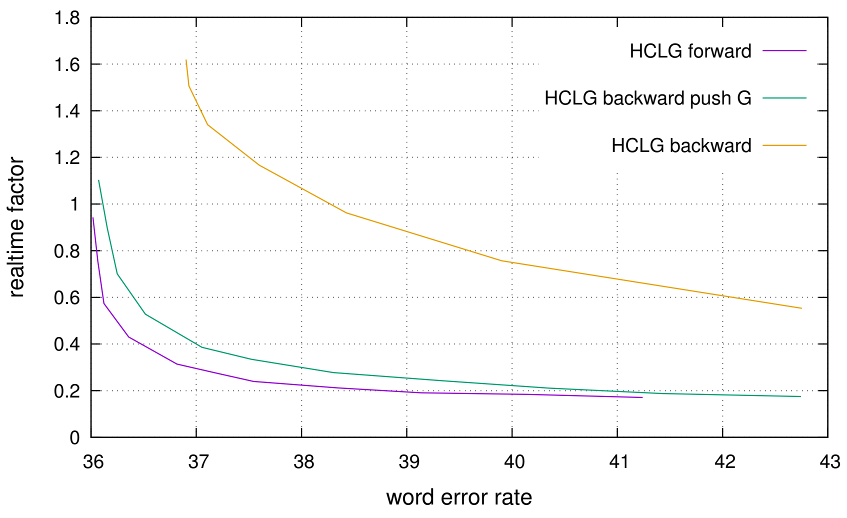
Another related open problem to derive a weight pushing algorithm, i.e. a shortest distance algorithm in the log-semi-ring, which respects the special semantics of failure arcs. Under this interpretation, the back-off LM would be correctly normalized and the total weight of the transducer would be one. Due to the incorrect interpretation of back-off LM (section 2.5.1), we are pushing weights that are greater than one. In case a state has a single outgoing arc, this results in a negative arc weight (when represented as negatied log-probability), which can cause to problems for several graph algorithms and their common implementations.

3.4 Experimental validation

We measure the effect of the new weight pushing algorithm by constructing a full recognition network (*HCLG*, section 2.4.1). In addition to the forward network, we construct a backward network (section 5.2) as the composition of the reversed components. The language model is reversed according to chapter 4, but the resulting transducer is not yet stochastic. We want to emphasize, that this is not just a reversed LM WFST. As will be explained in chapter 4 WFST reversal is not feasible to LM WFST, but we have to use the algorithm explained in the same chapter. The outcome of the LM reversal is not yet normalized, thus we want to apply weight pushing. The generic weight pushing algorithm (figure 3.2) is not applicable in this case, therefore, we apply the new weight pushing algorithm described in this chapter. We measure the decoding performance of the backward network with and without applying the new weight pushing algorithm to the reversed language model before the composition of *HCLG*. The experiment was done using Kaldi's Switchboard recipe (`egs/swbd/s5c/tri3/`). We report the performance on the Eval2000 data set[16], using a speaker-independent tri-phone GMM model on LDA transformed MFCC features and a language model trained on Switchboard. We test on two different recognition networks of different sizes (using a bi-gram and tri-gram LM). The real-time-factor was measured on a single core of a Intel(R) Core(TM) i5-2500 CPU at 3.30GHz with 8 GB of memory.

As seen from figures 3.4, the application of the weight pushing is crucial for the performance - the un-pushed backward language model performs much worse. We also compare to the performance of the forward graph. Since all components, including the language

[16]The decoding parameters are set to: rescore-acoustic-score 13.0, word insertion penalty 0.0, acoustic scale 1/12, max-active 7000, lattice-beam 6.0.



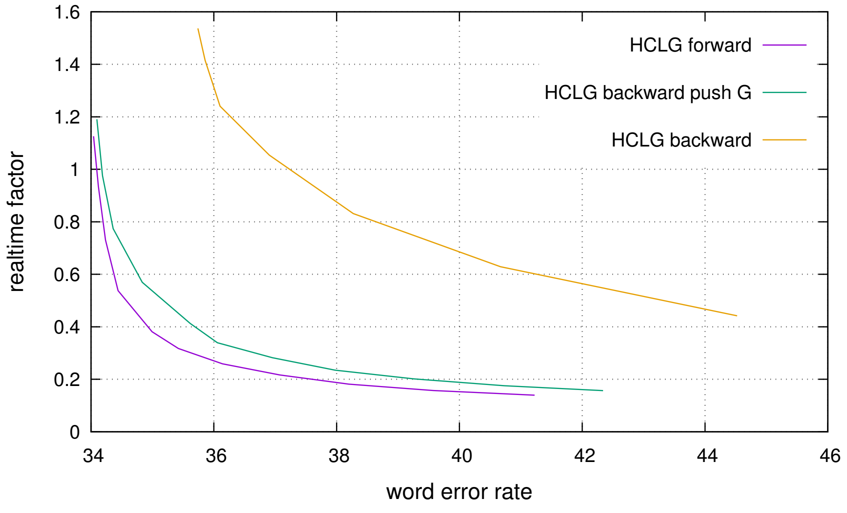


Figure 3.4: *Decoding performance of backward decoding network reported on the Eval2000 data set with a GMM model. We tested it using a bi-gram LM (top) and a tri-gram LM (bottom). Shown is the relation between word error rate and real-time-factor. Better performance is indicated by curves closer to the lower left corner. We compare the performance of the backward decoding network (HCLG backward) with and without the application of the new weight pushing algorithm. For comparison, we also show the performance of the forward decoding network (HCLG forward). In this case, no weight pushing is necessary.*

model, are stochastic (except for the issues with back-off arcs mentioned above) and thus should have optimal performance, it is not necessary to apply weight pushing. As explained in chapter 5, the performance of the forward and backward graphs is not necessarily the same, depending e.g. on properties of language and domain (here conversational English). However, the comparison indicates, that the weight pushing results in a network, that is not far from the optimal performance.

3.5 Conclusions

To achieve an optimal pruning behavior, it is desirable that the given WFST is stochastic. If the WFST is stochastic or has a similar optimal weight distribution, we showed that it is necessary to perform weight pushing. Given the theoretical background for weight pushing, we showed why the standard algorithm is either inefficient for the type of models used or fails to converge at all. By explaining the connection between non-negative matrices and ergodic Markov chains, we motivated an alternative weight pushing algorithm, that always converges and is much more efficient for the given task of optimizing language model transducers.

Chapter 4

Exact reversal of ARPA back-off language models

We present details about how to exactly reverse ARPA back-off language models (LM). For the purpose of searching for the best path through composed probabilistic models forwards and backwards in time and to combine these searches, it is desirable to have a backwards language model, that assigns exactly the same scores as the forward language model, while at the same time it has the same properties of being deterministic, stochastic and of minimal size to guarantee an optimal search. We show an approach to construct such a backwards LM, which is valid when using failure arcs and also when using epsilon arcs to represent the back-off structure as weighted finite state acceptor. This means that the weight of a path in the backward LM WFSA is equal to the corresponding forward LM WFSA, independent of the origin of the weights, i.e. whether estimated as interpolated, converted interpolated or back-off LM. We test the reversal algorithm on language models of different sizes and on different corpora and we compare it to training a language model on the time reversed training texts.

4.1 Motivation: forwards and backwards search

The application that we had in mind while investigating into this kind of models was the search for the best path through a composed probabilistic model. This can be for example the decoding of the most probable sequence of words in large vocabulary automatic speech recognition (LVCSR). However, the reversed language models presented here can also be used in many other tasks as e.g. in finding the most probable sentence in statistical machine translation. Given the complexity of most of the tasks, it is necessary to use heuristic pruning techniques, which introduce search errors. As will be explained in more detail in section 5.1, the search errors of searching forwards and backwards are mutually independent. Therefore, backward search has the potential to find the best path, even if it was pruned by the forward search.

Both models, forward and backward, should be equally powerful, i.e. have roughly the same accuracy and run-time requirements, and have similar structure, size and level of determinism to guarantee an optimal search. If both models, forward and backward, assign exactly the same probabilities to a hypothesis, it has the advantage, that the results of forward and backward decoding can be compared or combined (section 5.1). To be able to compare and combine the scores of partial results (paths), also the model structure (the distribution of weights along paths) should be similar in the forward and backward model.

Our assumption is that we are given a composed forward graph $HCLG_{fwd}$ (section 2.4.1), where one of the components is an LM acceptor G (section 2.5). The task is to obtain a backward graph $HCLG_{bwd}$ that fulfills all of the above mentioned requirements. As will be explained in more detail in section 5.2, the solution is to reverse each component separately and then construct the backward graph $HCLG_{bwd}$ in analogy to the forward graph. The trivial solution to apply WFST reversal is not sufficient, since the resulting graph would not have a similar level of determinism and not have a similar distribution of weights as the forward graph, i.e. it would show sub-optimal behavior when used in a pruned search. As we will see in this chapter, especially the (reversed) LM component would introduce a great degree of local ambiguity - the word context is delayed in the reversed model (section 4.2) and might result in falsely pruned paths.

To reverse the LM acceptor G turns out to be a complex task – this is the topic dealt with in this chapter. The task is to construct an LM acceptor G_{bwd}, that assigns exactly the same scores as G_{fwd} (to the reversed utterances). Again, a trivial solution is to apply WFSA reversal to G_{fwd} (followed by epsilon removal, determinization, and weight pushing in the log semi-ring - as explained in section 3.1, this is needed to achieve optimal pruning behavior and makes the WFSA stochastic). However, for LMs of higher order than bi-gram, this approach fails. As explained in section 4.2, this is because of the delayed word context and different backing-off states. Additionally, as already explained in section 3.1, the conventional weight pushing cannot be applied to WFSAs resulting from LMs, but we should use the alternative weight pushing (section 3.3). Thus, to achieve optimal search behavior, we need to construct an LM acceptor G_{bwd} with similar structure as the forward acceptor G_{fwd} that assigns exactly the same scores to the reversed utterances, and that also makes it possible to compare partial word sequences of forward and backward decoding.

Another trivial solution would be to train a new model on the reversed training texts (e.g. [Tang and Cristo(2008)]) - given that those are still available. This does however not result in exactly same scores for the same utterances, since there is usually no such constraint applied in the LM estimation[1]. Since we wanted exactly the same scores, we did not follow this approach further. Also, it would make our approach inconvenient to use in cases where the original LM text is no longer available. To determine the impact of those score differences, we compare an exactly reversed tri-gram model to a tri-gram model trained on the reversed training texts. We test the LM reversal algorithm on language models of different sizes.

4.2 Construction of an exactly reversed language model

In section 2.5, we explained how N-gram back-off language models are represented as weighted finite state automata and stored in the ARPA format. In this section, we show how to construct a backward LM, that has the same properties as the forward LM – i.e. it is deterministic (except for the ϵ-arcs), has the same size and a similar structure. The algorithm presented here is valid for failure arcs and for epsilon arcs.

[1]Based on the idea that Kneser-Ney models apply a constraint for the marginal distributions to estimate the probabilities in the LM, we assume that it is possible to formulate another type of constraint, that the LM probabilities should be estimated in such a way, that when estimated on reversed training texts and applying the exact reversal algorithm presented here, it should give the same probabilities as when estimating the normal forward LM.

The WFSA corresponding to the forward LM accepts a sequence of words and accumulates the weights along the path - see figure 4.1. If the probability semi-ring is used, the path weight is the product of the individual probabilities. If logarithmic probabilities are used, the path weight is the sum of the individual scores. Two WFSA are equal, if they accept the same set of sequences with the same path weights. Thus, it is possible to distribute the weights differently along the path, as long as the total product (or sum for logarithmic weights) stays the same for all paths. When we directly apply FSA reversal, which basically corresponds to swapping the source and destination states of the arcs, the resulting structure would be highly non-deterministic. In the example (figure 4.1), we would start backwards from the final state. All incoming arcs into the final state (only one example is shown) have the label </s>. Thus, we would have to apply the tri-gram probability $P(</s>|c,d)$ after only having seen only one symbol of the tri-gram (</s>). However, only after two more symbols d, c have been seen, the destination state can be determined unambiguously. For that reason, it would be logical to delay the application of the weight (probability), until a sufficient number of symbols have been consumed to unambiguously determine the destination state. For a tri-gram LM, this means delaying the weights by two steps. Figure 4.2 shows the corresponding path in the backward LM.

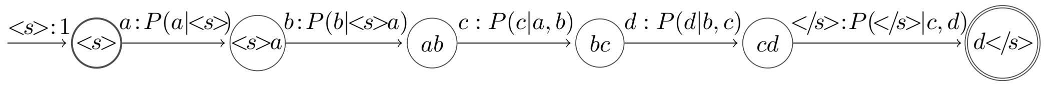


Figure 4.1: *Example of a forward path through a tri-gram language model - every state corresponds to a history of the two last symbols consumed. The model accepts the sequence a, b, c, d (input symbols) and the path weight is the product of the individual probabilities. For simplicity, sentence-start and sentence-end are treated here as ordinary symbols. Only one path is shown, but the reader has to keep in mind, that there e.g. multiply arcs entering the final state, all with the same label.*

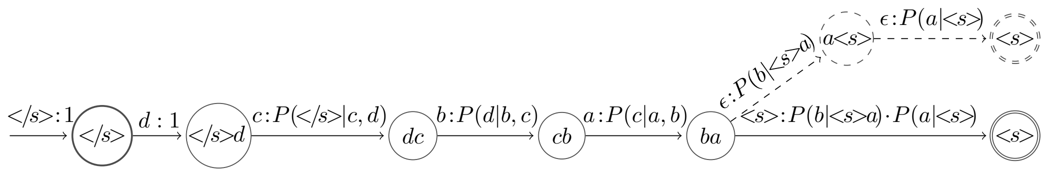


Figure 4.2: *The backward path corresponding to the path in figure 4.1. Additionally to reversing the path, the weights/probabilities have been delayed by two steps. Therefore, two arcs with probability one have been inserted at the beginning. To compensate, we could add two ϵ-arcs at the end (as depicted in dashed), where the last arc corresponds to backing-off to a history-less state at the sentence beginning (now end). Instead of introducing back-off arcs at the end of the sentence, we can collapse the probabilities of all the lower-order back-offs onto one arc, i.e. we use $P(b|<s>a) \cdot P(a|<s>) \cdot P(<s>)$. This is well in line with the common WFST LM implementation, which assumes, that the state reached by an N-gram containing the sentence-end symbol is a final state.*

When certain N-grams do not have sufficient coverage in the training corpus and are approximated by backing-off to lower order N-grams (see figure 4.3), the sequence of the weights in the backward LM is again exactly reversed as in the forward LM, and the same

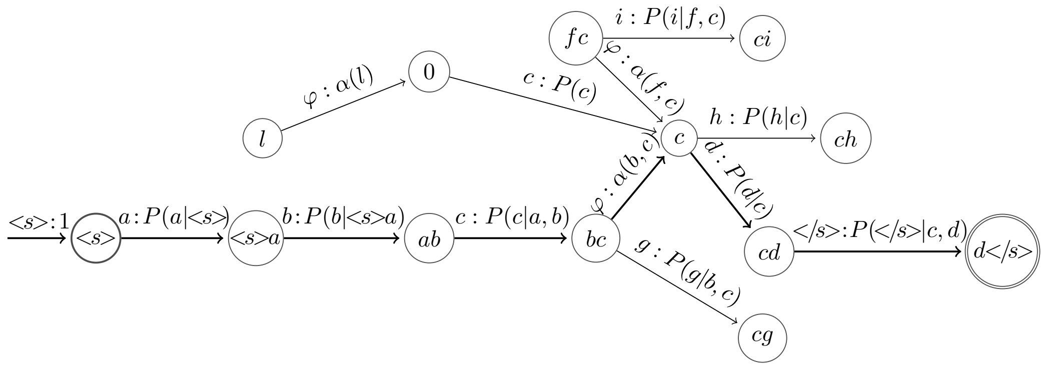


Figure 4.3: *The same example of a forward path as in figure 4.1, but with backing-off. The thick arcs correspond to the path in fig. 4.1 and further arcs have been added to illustrate the structure of the back-off LM. Since the N-gram bcd was not seen sufficiently often, it is approximated by backing-off to state c with back-off weight $\alpha(b, c)$ and then using the bi-gram cd with probability $P(d|c)$. The failure-arc (symbolized by φ) doesn't consume any symbol, but this arc is chosen for all symbols, that have no outgoing arc out of the same state ('default' clause). For the non-deterministic WFSA approximation, we would use the symbol ϵ instead and not consume a symbol either. The state 0 corresponds to the history-less back-off state when backing-off to uni-grams.*

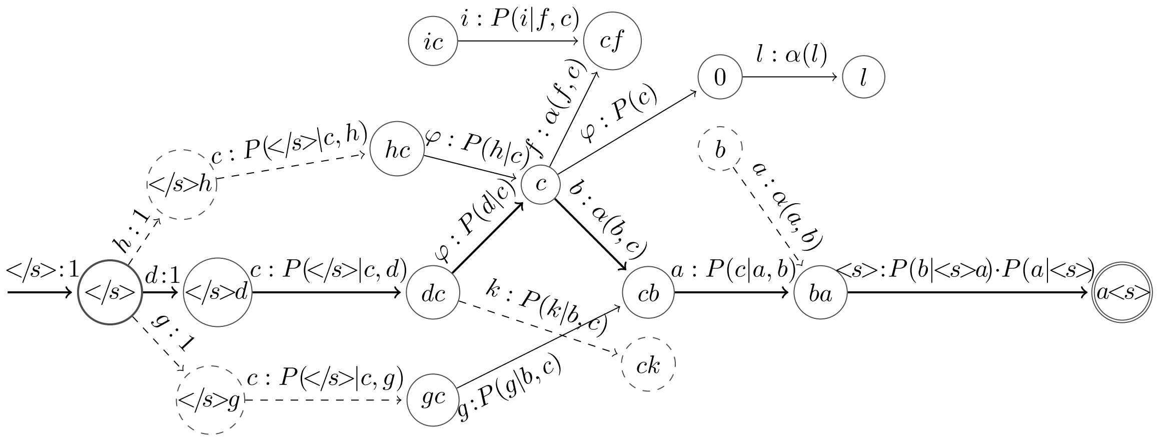


Figure 4.4: *The backward structure corresponding to figure 4.3. The thick arcs correspond to the path in fig. 4.2, and all solid arcs are reversed arcs from fig. 4.3. Dashed arcs have been added to illustrate further structure of the backward model. Similar as in figure 4.2, the weights have been delayed by two steps. Compared to the forward structure in figure 4.3, the sequence of weights is exactly reversed. The probability on an arc between two particular states is the same in the forward and backward model. I.e. compare the forward arcs bc − c − cd in fig. 4.3 to the backwards arcs dc − c − cb in this figure (cb corresponds to bc). However, since all labels are off by two states in the backward model, the back-off probability $\alpha(b, c)$ is now actually applied on a bi-gram arc with a label (b) and the bi-gram probability $P(d|c)$ is applied on a back-off arc with φ. Since all backward-tri-grams ending in cb (like dcb, hcb) share b as last label, it is logical to first back-off from the history (dc, hc) to the common history c and then apply the common label b. Since the reverse order of the weights has been preserved, the bi-gram probabilities serve now as back-off weights, and the former back-off weights serve as bi-gram probabilities. The same holds for the history-less state 0 - the uni-gram back-off weight $\alpha(l)$ and the uni-gram probability $P(c)$ have switched their role.*

delay of the weights is applied to make the model deterministic[2]. However, the sequence of the labels for back-off arcs is changed - back-off weights and lower-order N-grams change their role. The reason for that is we always have to back-off to a common history before consuming the next label - so the failure-arc (symbolized by φ) in the backward model takes the lower-order N-gram probability (from the forward model) and the label-arc takes the former back-off weight. Figure 4.4 shows this in the construction of a backward back-off LM from figure 4.3.

Figure 4.4 shows that it is possible to construct a backward LM, that has the same size and structure as the forward LM and is deterministic. From the construction, we observe, that a forward LM can be transformed into a backward LM by a series of relatively simple steps: Since the sequence of labels is processed in reversed order, the names of all states and N-grams are reversed (*abc* becomes *cba*). The N-grams of the highest order do not have back-off weights, and thus they stay unchanged (arcs appear similar in the forward and backward models). However, for all lower-order N-grams, the role of the back-off weight and the N-gram probability changes. When represented in the ARPA format (figure 2.12), the transformation becomes even simpler: For all lower-order N-grams, the whole line is reversed, and for the highest-order N-gram, only the N-gram is reversed. E.g. for a tri-gram LM, a bi-gram entry $P(b|a)\ a\,b\ \alpha(a,b)$ becomes $\alpha(a,b)\ b\,a\ P(b|a)$ and a tri-gram entry $P(c|a,b)\ a\,b\,c$ becomes $P(c|a,b)\ c\,b\,a$. The symbols for sentence begin and sentence end have to be exchanged, and special care has to be taken for N-grams starting and ending a sentence. For all N-grams ending a sentence, we multiply all lower-order probabilities (e.g. for N-gram *ba</s>* we use $P(b|a,\text{<s>}) \cdot P(a|\text{<s>}) \cdot P(\text{<s>})$).

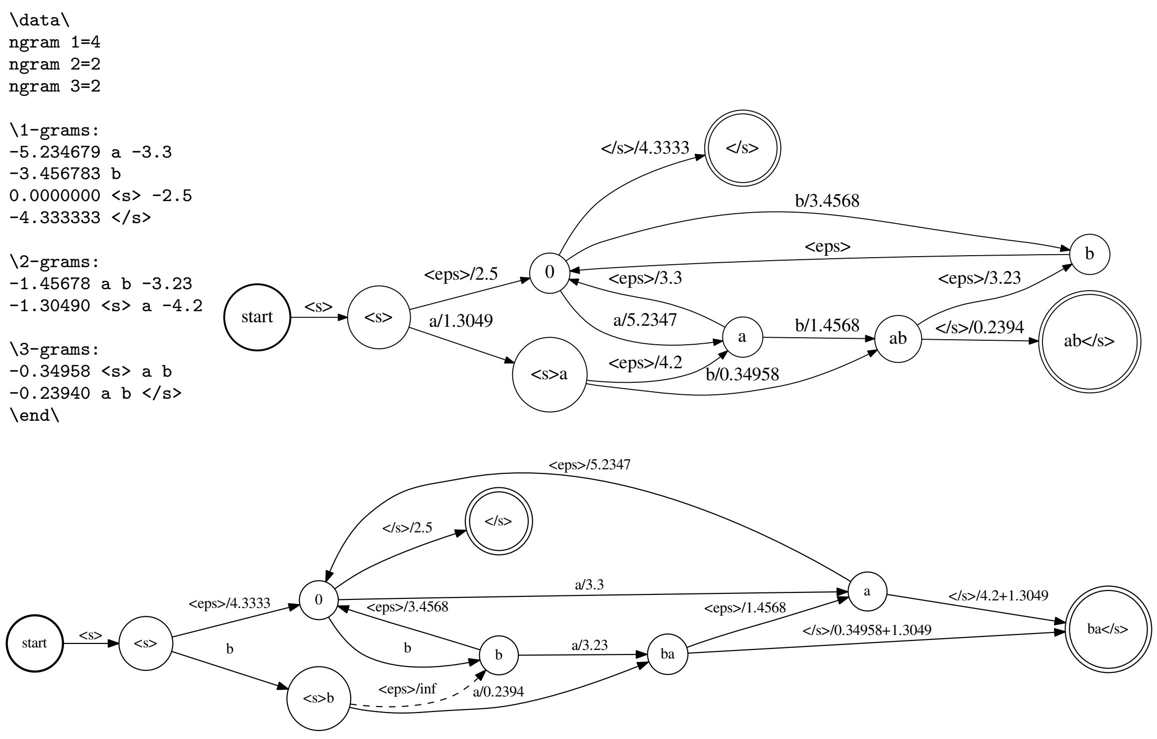
By introducing the short-hand notation $P(ABCD) = P(D|A,B,C)$, we can write the rules for a four-gram LM in the form of equations:

$$\begin{aligned}
P(A) &= \alpha(A) \qquad (4.1)\\
\alpha(A) &= P(A)\\
P(BA) &= \alpha(AB)\\
\alpha(BA) &= P(AB)\\
P(CBA) &= \alpha(ABC)\\
\alpha(CBA) &= P(ABC)\\
P(DCBA) &= P(DCBA).
\end{aligned}$$

4.3 The treatment of missing N-grams

Figure 4.5 shows the LM reversal rules (equation 4.2) applied to a tri-gram back-off ARPA LM. While the rules are rather simple, an additional complexity arises, when representing ARPA models (back-off N-gram LMs in general) as WFSAs. If there is an N-gram entry for *abcd* in the ARPA, the resulting WFSA needs the back-off states *bcd*, *cd* and *d*. Due to LM pruning, and due to other reasons that we are going to explain in this section, for some of the N-grams *abcd* defined in the ARPA file, there is no corresponding tri-gram entry *bcd* or bi-gram entry *cd*, i.e. we are not given the probalty $\alpha(bcd)$ of backing-off *abcd* $\rightarrow$ *bcd*,

[2]The model is only truly deterministic, if we use failure arcs, but the construction presented here is also valid for ϵ-arcs.



```
\data\
ngram 1=4
ngram 2=2
ngram 3=2

\1-grams:
-5.234679 a -3.3
-3.456783 b
0.0000000 <s> -2.5
-4.333333 </s>

\2-grams:
-1.45678 a b -3.23
-1.30490 <s> a -4.2

\3-grams:
-0.34958 <s> a b
-0.23940 a b </s>
\end\
```

Figure 4.5: *Upper part: Forward WFSA for the tri-gram back-off ARPA LM (repeated from figure 2.12). We apply the rules from equation 4.2 to obtain the backward WFSA (lower part). We see that exactly the same probabilities are used between the states (e.g. $a \rightarrow ab$ in upper model and $a \rightarrow ba$ in lower model) and that back-off weights in the upper model are now on word arcs in the lower model. As already said, an alternative interpretation of a tri-gram transition $ab \rightarrow bc$ is to go to an imaginary state abc and immediately backing off to state bc. Therefore, the final state in the upper part is the tri-gram state $ab</s>$. However, since this is a final state, there is no way to back-off from it to the state $b</s>$. Moreover, in the forward ARPA definition (upper left), the N-gram corresponding to the state $b</s>$ is missing.*

neither $P(d|b,c)$. N-grams that are needed for the construction of the WFSA, but not defined in the ARPA, we call missing N-grams.

During the construction of the recognition graph from the forward LM, missing back-off states are usually added automatically. For example, in the tri-gram LM in the upper part of figure 4.5, the tri-gram $<s>ab$ leads into the state ab. Let's imagine the corresponding back-off bi-gram ab is not given in the ARPA file (e.g. due to pruning): In this case, during the construction of the recognition graph, the state ab needs to be automatically created, as it is the target of the tri-gram. Since there is no bi-gram probability for ab ($P(b|a) = 0.0$, and no successor bi-grams), we should immediately back-off to state b. Thus, the bi-gram ab is added with back-off weight $\alpha(a,b) = 1.0$ (zero in log-domain). However, it should not be possible to reach the newly created state ab from a, since the N-gram ab is missing ($P(b|a) = 0.0$ or minus infinity in log-domain).

In terms of the WFSA representation of the LM (right part of figure 4.5), this would mean, that there would be no link between a and ab, and the link between ab and b would be added with zero cost. In the reversed LM, where forward probability and back-off weight change their role, this does lead to the situation, that we are able to reach ba from b with $\alpha(a,b) = 1.0$, but we are not able to back-off from ba to a, since this corresponds to a path that was not present in the forward model ($P(b|a) = 0.0$). If we would make the missing N-grams explicit in the ARPA file, in the forward ARPA file, the missing N-gram would result in an entry '$-$`inf` $a\,b$ 0.0', and in the backward ARPA file, this results in entries of the type '0.0 $b\,a$ $-$`inf`'. This might seem awkward, because we never have infinte log-back-off weights in the forward LM, but it is necessary to make the forward and backward LMs match exactly.

An example of a missing N-gram is in figure 4.5: For the entry ab</s>, the back-off state b</s> is missing in the ARPA. It would be automatically added when constructing the WFSA, but here it is not necessary, since after observing the sentence-end symbol, no other N-gram can follow[3]. Even if we don't need it in the forward model, it still has an effect on the backward model. To make the missing N-gram explicit in the forward model (upper part), we would add it in such a way, that we can back-off to the missing state b</s> and from it to the final state </s>. However, b</s> should not be reachable through the missing N-gram $b \to b$</s>. According to the rule, we create '$-$`inf` b </s> 0.0' in the forward ARPA. In the reverse model (lower part), this corresponds to being able to reach the missing state by the N-gram <s>$\to b$<s>, but not being able to back-off from b<s> to the lower order state b, since this corresponds to a path that was not present in the forward model. This results in adding '0.0 </s> b $-$`inf`' in the backward ARPA and exactly corresponds to the state <s>b in figure 4.5, which can be reached with probability one, but there can be no back-off arc leaving this state (indicated as dashed arc with infinite cost). To summarize, we need to add the state <s>b in the reverse model, but the back-off link <s>$b \to b$ is not allowed to have an equivalent backward model.

Missing N-grams result from a complex interplay of the type of back-off distribution, cut-off frequencies and LM pruning (e.g. based on entropy [Stolcke(1998)]). As we have already explained, the first type of N-grams that are missing in the ARPA file are back-off N-grams that end a sentence (e.g. b</s>). Otherwise, if we don't apply pruning, we would expect that the presence of a higher-order N-gram implies the presence of the lower-order N-gram (e.g. with a shortened history), since the absolute observation count of the lower-order N-gram should be equal or higher than the count of the higher-order N-gram. We encounter missing N-grams, when we use different cut-off frequencies (parameter k in equation 2.13) for different N-gram orders. For example, when we use SRILM's default setting $k_4 = 1, k_3 = k_2 = k_1 = 2$ for four-gram LMs, we get missing tri-grams for all four-grams, whose back-off tri-gram was only observed once.

As we will see now, we also get missing N-grams, if we use lower-order distributions, which are not based on counts. The distribution for the highest-order N-grams $P'(w_i|h_i)$ (equations 2.13, 2.16) is usually based on the counts $C(h_i, w_i)$. When back-off models were introduced [Katz(1987)], also the lower-order back-off distributions P_{lower} were based on counts. However, when we have to back-off, we should make use of the fact that this particular word is unseen in the given context. We would expect a different distribution of words, than when we were not given that information, i.e. not just expect any frequent word. In other words, we should use a different type of distribution for $P_{lower}(w_i|\bar{h_i})$ than for $P'(w_i|h_i)$. Following that intuition, Kneser-Ney-type LMs [Kneser and Ney(1995)] use a back-off distribution, where the probability of a word, unseen in a certain context, is proportional to the number of possible predecessor words types that can occur before that context:

[3]By convention, a state reached by an N-gram containing the sentence-end symbol is a final state.

$$P_{backoff}(w_i|w_{i-n+1}\dots w_{i-1}) = \frac{|w_{i-n} : C(w_{i-n}\dots w_i) > k|}{\sum_{w_i} |w_{i-n} : C(w_{i-n}\dots w_i) > k|}. \tag{4.2}$$

As a consequence, we can expect words or phrases, that appear frequently, but only in very few different contexts, to have a low probability in the back-off model. For example, we would not expect the word "Francisco" to appear in many other contexts than together with "San Francisco", despite the fact that it is a frequent word. For that reason, we often find higher-order N-grams in the LM, such as "San Francisco area", for which the back-off N-gram "Francisco area" is missing. When constructing the backward LM, we will add the missing N-gram "area Francisco" with probability one and infinite back-off weight. This means, that after observing "area Francisco", we are only able to continue with "San" and there can be no back-off to "Francisco", which would allow to continue with another word.

In fact, when experimenting with LMs trained on sentences from the Wall Street Journal corpus [Paul and Baker(1992)], we observed that any common multi-word phrase can result in missing lower-order N-grams. An N-gram starting within a multi-word phrase has very few different left contexts, which causes it to have low back-off probability. If the right context of that N-gram is either almost completely undetermined or completely determined (e.g. sentence end), all N-grams that would continue the phrase fall below the cut-off frequency and are thus not present in the LM. Typically, a multi-word phrase like "on behalf of" or "New York City" is followed by a word that introduces lot's of ambiguity - e.g. "on behalf of **the**". If no N-gram "behalf of the X" is above the cut-off frequency, then also the back-off N-gram "behalf of the" is missing in the LM, since the probability of seeing it in a new context other than "on" is extremely low. As already mentioned, also for all N-grams ending a sentence, there is no succeeding N-gram, which is a similar situation. It is quite obvious, that LM pruning (e.g. based on entropy [Stolcke(1998)]) will increase the number of missing N-grams. According to the same principle, N-grams with a low probability in the back-off distribution, and no successor N-grams (due to pruning) are missing as well.

4.4 Proof: Exact reversal of the language model

We have verified that our "reversed" ARPA LM, and also the corresponding WFST assigns the same score to a reversed sentence that our original ARPA LM assigned to the original sentence. In this section, we sketch a proof for the correctness of the algorithm for reversing the LM, as presented in section 4.2. The steps are valid for back-off and interpolated LMs. We do this by introducing a series of simple transformations, that each guarantee the equivalence of the LM WFSA (the same sequence of symbols gets the same score):

1. Modify the ARPA model to make the back-off costs zero while maintaining the sentence-level scores the same,
2. Convert to "max-ent" form, reverse in the "max-ent" form, which is easy,
3. Convert back to ARPA form [still not normalized per word],
4. Convert to a WFSA, and apply the new weight pushing.

Input to the algorithm is a language model (LM) in ARPA format[4], which contains entries in the form '$p(ABC)\ ABC\ \alpha(ABC)$', and its representation as a WFSA (top of

[4]ARPA stores log-probabilities, but for simplicity, we show probabilities here.

figure 4.5). Here, ABC stands for the three words A,B,C, p is the N-gram weight and α is the back-off weight. We use the notation $p(ABC)$ meaning $P(C|A,B)$. The result of these steps is a reverse LM that assigns exactly the same scores as the forward LM.

First step: Pushing the back-off costs

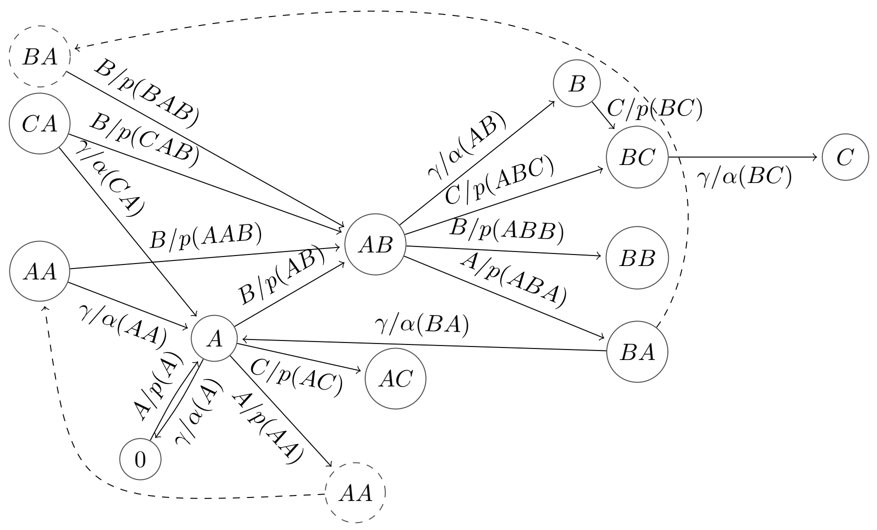


Figure 4.6: *WFSA of a toy tri-gram LM with just three words A, B, C, focusing on the N-grams that contain AB. On the left, there are states corresponding to histories ending in A, in the center are N-grams that start with A and on the right are N-grams starting in B. For a vocabulary $V = 3$, every state (except for the zero-gram 0) has four incoming and four outgoing arcs. Conceptually, we see a tri-gram arc $AB \to BC$ as two arcs $AB \to ABC$ with $p(ABC)$ and backing-off $ABC \to BC$ with $\alpha(ABC) = 1.0$. In this interpretation, all arcs with word labels go up one level in the hierarchy, and all back-off arcs go down one level in the hierarchy.*

In the first step, we push the weights $\alpha()$ of back-off transitions. We do this using the simple potential function defined in equation 3.4. This means, we multiply a fixed value k to the weights of all incoming arcs into a particular state q' and we divide the same value from the weights of all arcs leaving that state. We do so starting from the uni-gram-back-offs, and then going upwards to the bi-gram-back-offs, and so on (the highest order back-offs are 1.0 anyway). In figure 4.6, we show the relevant arcs for this operation: To push the back-off weight $\alpha(A)$ from the arc $A \to 0$, we divide all outgoing arcs of the state A by $\alpha(A)$ (e.g. the arc $A \to AB$, thus we change $p(AB)$), and we multiply all incoming arcs of A by $\alpha(A)$ (e.g. the arc $BA \to A$, thus we change $\alpha(BA)$). Pushing the weights from all uni-gram back-offs results in a WFSA $\mathbf{P}_{z1}$:

$$\begin{aligned} p_{z1}(A) &= p(A)\,\alpha(A) \\ \alpha_{z1}(A) &= 1.0 \\ p_{z1}(AB) &= p(AB)/\alpha(A) \\ \alpha_{z1}(AB) &= \alpha(AB)\,\alpha(B) \\ p_{z1}(ABC) &= p(ABC) \end{aligned} \tag{4.3}$$

We only write one equation for each type of arcs, i.e. from $p_{z1}(AB) = p(AB)/\alpha(A)$, we also know that $p_{z1}(CB) = p(CB)/\alpha(C)$, and so on. With our notation, we want to show the whole WFSA for all possible N-gram orders. Therefore, we write also the tri-gram probabilities $p_{z1}(ABC)$, which are not affected by this step. If we would have a four-gram LM, it is clear, that also $p_{z1}(ABCD)$ is not affected. If we would have a bi-gram LM, we join the tri-gram arc and the tri-gram back-off arc into one arc, using $\alpha(AB) = 1.0$:

$$p_{z1,bigram}(AB) = p_{z1}(AB) \cdot \alpha_{z1}(AB) = (p(AB)/\alpha(A)) \cdot (\alpha(AB)\,\alpha(B)) = p(AB) \cdot \alpha(B)/\alpha(A),$$

and we would truncate our derivation, since we would already have an WFSA with back-off weights one. For tri-gram LMs, we apply now a second pushing step, where we push the weight $\alpha_{z1}(AB) = \alpha(AB)\,\alpha(B)$ from the back-off arc $AB \to B$. From figure 4.6, we see, that we divide all outgoing arcs of the state AB by $\alpha_{z1}(AB)$, and we multiply all incoming arcs of AB by the same quantity. From figure 4.6, we see, that the arc $AB \to ABC$ is influenced by $AB \to B$ and $BC \to C$, thus both $\alpha_{z1}(AB)$ and $\alpha_{z1}(BC)$ influence $p(ABC)$. This step results in a WFSA $\mathbf{P}_{z2}$:

$$\begin{aligned}
p_{z2}(A) &= p(A)\,\alpha(A) \\
\alpha_{z2}(A) &= 1.0 \\
p_{z2}(AB) &= \frac{p(AB)\alpha(AB)\,\alpha(B)}{\alpha(A)} \\
\alpha_{z2}(AB) &= 1.0 \\
p_{z2}(ABC) &= \frac{p(ABC)}{\alpha(AB)\,\alpha(B)} \\
\alpha_{z2}(ABC) &= \alpha(ABC)\,\alpha(BC)\,\alpha(C)
\end{aligned}$$

For tri-grams, we use $\alpha(ABC) = 1.0$ and truncate or derivation with:

$$p_{z2,trigram}(ABC) = p_{z2}(ABC)\,\alpha_{z2}(ABC) = \frac{p(ABC)\,\alpha(BC)\,\alpha(C)}{\alpha(AB)\,\alpha(B)}.$$

A clear pattern is observed. To summarize, for an N-gram of any order, by a series of $N-1$ weight pushing steps, we obtain a WFSA $\mathbf{P}_z$ that has all back-off weights one (log zero):

$$\begin{aligned}
p_z(A) &= p(A)\,\alpha(A) \\
p_z(AB) &= \frac{p(AB)\alpha(AB)\,\alpha(B)}{\alpha(A)} \\
p_z(ABC) &= \frac{p(ABC)\,\alpha(ABC)\,\alpha(BC)\,\alpha(C)}{\alpha(AB)\,\alpha(B)} \\
&\dots
\end{aligned} \tag{4.4}$$

Second step: Converting to "max-ent" form

In a second step, we view the WFSA in a multiplicative space, which is inspired by the N-gram features in a maximum entropy LM [Berger et al.(1996)]. It models the probability of an N-gram with the help of a set of feature functions $f_i(h_i, w_i)$:

$$p_\lambda(w_i|h_i) = \frac{1}{Z_\lambda(h_i)} \exp\left(\sum_i \lambda_i\, f_i(h_i, w_i)\right) \tag{4.5}$$

Here, $Z_\lambda(h_i)$ is the normalizer to form a valid distribution and λ_i is the weight of the feature function $f_i(h_i, w_i)$, which in the simplest case is a binary indicator function, which can select particular words and histories (N-grams), but also other types of features can be used, asking about part-of-speech etc. Here, we assume an indicator function f_i for each uni-gram, bi-gram, tri-gram and so on. When evaluating the words ABC, a back-off LM would only consider $p(BC)$, if there is no $p(ABC)$ in the LM, in which case it has to back-off to history B. On the other hand, in interpolated LMs, $p(ABC)$ is estimated by interpolating the N-gram features of all orders. This is similar in a 'maxent-type' model, however, we don't use additive interpolation, but multiply (exponential of sum) the feature contributions of lower orders. Inspired by this, we transform our LM weights into a multiplicative space, where all N-gram orders contribute:

$$p_{maxent}(ABC) = p_f(ABC) \cdot p_f(BC) \cdot p_f(C). \tag{4.6}$$

With this step we actually leave the original semi-ring, but we use it just as an intermediate step for explanation. When we use a back-off LM, we want to construct the model so that:

$$p_{maxent}(ABC) = \begin{cases} p_z(ABC) & \text{if } p_z(ABC) \\ p_z(BC) & \text{else if } p_z(BC) \\ p_z(C) & \text{elsewhere} \end{cases}. \tag{4.7}$$

This can be achieved by setting:

$$\begin{aligned} p_f(C) &= p_z(C) \\ p_f(BC) &= p_z(BC)/p_z(C) \\ p_f(ABC) &= p_z(ABC)/p_z(BC) \\ &\dots \end{aligned} \tag{4.8}$$

The combined weight pushing from back-off arcs and conversion to "max-ent" type results in:

$$\begin{aligned} p_f(A) &= p(A)\,\alpha(A) \\ p_f(AB) &= \frac{p(AB)\,\alpha(AB)}{\alpha(A)\,p(B)} \\ p_f(ABC) &= \frac{p(ABC)\,\alpha(ABC)}{\alpha(AB)\,p(BC)} \\ p_f(ABCD) &= \frac{p(ABCD)\,\alpha(ABCD)}{\alpha(ABC)\,p(BCD)} \\ &\dots \end{aligned} \tag{4.9}$$

The advantage is that this 'maxent-type' model can be easily reversed by:

$$p_{fr}(ABC) = p_f(CBA) \tag{4.10}$$

As already mentioned in section 4.2, the sentence begin and end symbols have to be switched in the ARPA file, and we assign the uni-gram probability of the former sentence begin (usually it is ignored, but in our case it won't be one) to the new sentence begin (former sentence end). Also, as explained in section 4.3, we have to explicitly add the missing back-off states before the reveral (e.g. if back-off state AB is not present for an entry XAB - see lower part of figure 4.5). In the following equations, we treat $p_b(AB) = 1.0$ if the state AB was added as a missing back-off state.

The resulting reverse maxent-type model can be transformed back to the ARPA-type:

$$\begin{aligned} p_b(C) &= p_{fr}(C) \\ p_b(BC) &= p_{fr}(BC) \cdot p_{fr}(C) \\ p_b(ABC) &= p_{fr}(ABC) \cdot p_{fr}(BC) \cdot p_{fr}(C) \\ &\dots \end{aligned} \tag{4.11}$$

The resulting backward LM is then:

$$\begin{aligned} p_b(A) &= p(A)\,\alpha(A) \\ p_b(BA) &= \frac{p(AB)\,\alpha(AB)\,p(A)}{p(B)} \\ p_b(CBA) &= \frac{p(ABC)\,\alpha(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \\ p_b(DCBA) &= \frac{p(ABCD)\,\alpha(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \\ &\dots \end{aligned} \tag{4.12}$$

So far, for the conversion to the "max-ent" form, we assumed a back-off LM (equations 4.7, 4.8). If the probabilities are represented as an interpolated LM, we want to construct a model, so that:

$$p_{maxent,int}(ABC) = \begin{cases} p_z(ABC) + p_z(BC) + p_z(C) & \text{if } p_z(ABC) \\ p_z(BC) + p_z(C) & \text{else if } p_z(BC) \\ p_z(C) & \text{elsewhere} \end{cases}. \tag{4.13}$$

This can be achieved by setting:

$$\begin{aligned} p_m(C) &= p_z(C) \\ p_m(BC) &= \frac{p_z(BC) + p_z(C)}{p_z(C)} \\ p_m(ABC) &= \frac{p_z(ABC) + p_z(BC) + p_z(C)}{p_z(BC) + p_z(C)} \\ &\dots \end{aligned} \tag{4.14}$$

Also here, the model can be easily reversed by:

$$p_{mr}(ABC) = p_m(CBA) \tag{4.15}$$

We can convert the max-ent model for interpolated LMs back with:

$$\begin{aligned} p_b(C) &= p_{mr}(C) \\ p_b(BC) &= p_{mr}(BC) \cdot p_{mr}(C) - p_{mr}(C) \\ p_b(ABC) &= p_{mr}(ABC) \cdot p_{mr}(BC) - p_{mr}(BC) \cdot p_{mr}(C) \\ p_b(XABC) &= p_{mr}(XABC) \cdot p_{mr}(ABC) - p_{mr}(ABC) \cdot p_{mr}(BC) \cdot p_{mr}(C) \\ &\dots \end{aligned} \tag{4.16}$$

Here, we show also the equation resulting for four-grams, to indicate continuation of the series. In the further steps, we only continue with the equations for back-off LMs, to save space.

Last step: Pushing the forward probabilities to back-off arcs

Now, in a final step, we take the result for back-off LMs (equation 4.12) and we apply a similar but inverse operation to what we applied to push the weights from the back-off arcs (equation 4.4). We can show, that this results exactly in the algorithm presented in section 4.2 (equation 4.2). First, we transform the uni-gram probabilities into the desired form (equation 4.2) by pushing the $p(A)$. This is the equivalent step to equation 4.3, but this time, we don't push $\alpha(A)$ from the arc $A \to 0$, but $1/p(A)$. From figure 4.6 (now we have to read AB as BA), we see that this affects the arcs to and from the state A (and B), among them are bi-gram probabilities and bi-gram back-offs:

$$\begin{aligned}
p_{b1}(A) &= \alpha(A) \\
\alpha_{b1}(A) &= p(A) \\
p_{b1}(BA) &= \frac{p(AB)\,\alpha(AB)\,p(A)\,\alpha(B)}{p(B)} \\
\alpha_{b1}(BA) &= \frac{1}{p(A)} \\
p_{b1}(CBA) &= \frac{(ABC)\,\alpha(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \\
\alpha_{b1}(CBA) &= 1.0 \\
p_{b1}(DCBA) &= \frac{p(ABCD)\,\alpha(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \\
\alpha_{b1}(DCBA) &= 1.0.
\end{aligned} \tag{4.17}$$

Of course, $p_{b1}(CBA)$ and $p_{b1}(DCBA)$ are not affected by this step, but we copy them from equation 4.12. Now, we transform the bi-gram probabilities by pushing $1/\left(p(AB)\,p(A)\right)$. This also affects the tri-gram CBA and its back-off arc:

$$\begin{aligned}
p_{b2}(A) &= \alpha(A) \\
\alpha_{b2}(A) &= p(A) \\
p_{b2}(BA) &= \alpha(AB) \\
\alpha_{b2}(BA) &= p(AB) \\
p_{b2}(CBA) &= \frac{p(ABC)\,\alpha(ABC)\,p(AB)\,p(A)}{} \\
\alpha_{b2}(CBA) &= \frac{1}{p(AB)\,p(A)} \\
p_{b2}(DCBA) &= \frac{p(ABCD)\,\alpha(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \\
\alpha_{b2}(DCBA) &= 1.0.
\end{aligned} \tag{4.18}$$

The third step is analogous for the tri-grams, by pushing $1/\left(p(ABC)\,p(AB)\,p(A)\right)$:

$$\begin{aligned}
p_{b3}(A) &= \alpha(A) \\
\alpha_{b3}(A) &= p(A) \\
p_{b3}(BA) &= \alpha(AB) \\
\alpha_{b3}(BA) &= p(AB) \\
p_{b3}(CBA) &= \alpha(ABC) \\
\alpha_{b3}(CBA) &= p(ABC) \\
p_{b3}(DCBA) &= p(ABCD)\,\alpha(ABCD)\,p(ABC)\,p(AB)\,p(A) \\
\alpha_{b3}(DCBA) &= 1/\left(p(ABC)\,p(AB)\,p(A)\right).
\end{aligned} \tag{4.19}$$

For a four-gram LM $\alpha(ABCD) = 1.0$ and there is only one arc with weight:

$$p_{b4}(DCBA) = p_{b3}(DCBA)\,\alpha_{b3}(DCBA) = p(ABCD) \tag{4.20}$$

We see, that the N-gram probability for the highest order stays the same $p_b(DCBA) = p(ABCD)$, and for all lower orders, the N-gram and back-off probabilities change their role. Thus, we have shown, that these steps result exactly in the same solution as in the equations 4.2.

4.5 Motivation by Bayes' formula

[Lee and Kawahara(2009)] point out, that the reverse LM can be constructed with the help of Bayes' rule. However, no details were given, especially it is unclear how to treat back-offs and back-off states. Here, we try to derive the LM reversal using Bayes' rule. The basic assumption is that the joint probability of word sequences should be the same in the forward and backward models:

$$p_b(w_N, \ldots, w_1) = p_f(w_1, \ldots, w_N). \tag{4.21}$$

This sounds reasonable, if the probabilities are based on counts. However, this might not be the case, if the lower-order probabilities are following another distribution, as e.g. the left-continuation probabilities used in Kneser-Ney language models (equation 4.2). We start our derivation with the uni-grams ($p_b(A) = p(A)$) and bi-grams:

$$\begin{aligned}
p_b(A) &= p(A) \\
p_b(B) &= p(B) \\
p_b(B, A) &= p(A, B) \\
p_b(B)\,p_b(A|B) &= p(A)\,p(B|A) \\
p_b(A|B) &= \frac{p(A)\,p(B|A)}{p(B)}.
\end{aligned} \tag{4.22}$$

We continue with the tri-grams:

$$\begin{aligned}
p_b(C,B,A) &= p(A,B,C) \\
p_b(C)\,p_b(B|C)\,p_b(A|C,B) &= p(A)\,p(B|A)\,p(C|A,B) \\
p(C)\,\frac{p(B)\,p(C|B)}{p(C)}\,p_b(A|C,B) &= p(A)\,p(B|A)\,p(C|A,B) \\
p_b(A|C,B) &= \frac{p(A)\,p(B|A)\,p(C|A,B)}{p(B)\,p(C|B)} = \frac{P(A,B,C)}{P(B,C)}. \qquad (4.23)
\end{aligned}$$

We can generalize this derivation for all N-gram orders. Given the forward word sequence w_1^N, we derive:

$$p_b(w_1|w_N, w_{N-1}, \ldots, w_2) = \frac{p(w_1, \ldots, w_N)}{p(w_2, \ldots, w_N)} = \frac{\prod_{i=1}^{N} p(w_i|w_1^{i-1})}{\prod_{i=2}^{N} p(w_i|w_2^{i-1})}. \qquad (4.24)$$

If we compare these formulas to equation 4.12, we see that we derived the same formulas – except for the additional back-off weights of the forward model. We repeat equation 4.12 here (moving the back-offs for clarity):

$$\begin{aligned}
p_{b0}(A) &= p(A)\,\alpha(A) \\
p_{b0}(BA) &= \frac{p(AB)\,p(A)}{p(B)} \cdot \alpha(AB) \\
p_{b0}(CBA) &= \frac{p(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \cdot \alpha(ABC) \\
p_{b0}(DCBA) &= \frac{p(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \cdot \alpha(ABCD) \\
&\ldots
\end{aligned}$$

With the help pf Bayes' rule, we determined the N-gram probabilities of the reversed model, but we didn't figure out the back-off weights. If we transform our solution to a form, that exactly retrieves the probabilities obtained from Bayes' rule, we can use the resulting back-offs weights as a Bayes'-like solution, i.e. we can assume, that the resulting model is correctly normalized to sum to one. In a similar series of pushing steps as before, we can push the former back-offs weights α of the forward model, so that the structure of the Bayes' formula is retrieved. Thus, we start again from equation 4.12, as we did in equation 4.17, but instead of pushing $1/p(A)$, we push the back-off weights $(1/\alpha(A))$ from the uni-grams:

$$\begin{array}{rcl}
p_{b1}(A) & = & p(A) \\
\alpha_{b1}(A) & = & \alpha(A) \\
p_{b1}(BA) & = & \dfrac{p(AB)\,p(A)}{p(B)} \cdot \alpha(AB)\,\alpha(B) \\
\alpha_{b1}(BA) & = & 1/\alpha(A) \\
p_{b1}(CBA) & = & \dfrac{p(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \cdot \alpha(ABC) \\
\alpha_{b1}(CBA) & = & 1.0 \\
p_{b1}(DCBA) & = & \dfrac{p(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \cdot \alpha(ABCD) \\
\alpha_{b1}(DCBA) & = & 1.0.
\end{array} \tag{4.25}$$

Then, we push $1/\left(\alpha(AB)\,\alpha(B)\right)$ from the bi-grams:

$$\begin{array}{rcl}
p_{b2}(A) & = & p(A) \\
\alpha_{b2}(A) & = & \alpha(A) \\
p_{b2}(BA) & = & \dfrac{p(AB)\,p(A)}{p(B)} \\
\alpha_{b2}(BA) & = & \dfrac{\alpha(AB)\,\alpha(B)}{\alpha(A)} \\
p_{b2}(CBA) & = & \dfrac{p(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \cdot \alpha(ABC)\,\alpha(BC)\,\alpha(C) \\
\alpha_{b2}(CBA) & = & 1/\left(\alpha(BC)\,\alpha(C)\right) \\
p_{b2}(DCBA) & = & \dfrac{p(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \cdot \alpha(ABCD) \\
\alpha_{b2}(DCBA) & = & 1.0.
\end{array} \tag{4.26}$$

Now, we push $1/\left(\alpha(ABC)+\alpha(BC)+\alpha(C)\right)$ from the tri-grams:

$$\begin{aligned}
p_{b3}(A) &= p(A) \\
\alpha_{b3}(A) &= \alpha(A) \\
p_{b3}(BA) &= \frac{p(AB)\,p(A)}{p(B)} \\
\alpha_{b3}(BA) &= \frac{\alpha(AB)\,\alpha(B)}{\alpha(A)} \\
p_{b3}(CBA) &= \frac{p(ABC)\,p(AB)\,p(A)}{p(BC)\,p(B)} \\
\alpha_{b3}(CBA) &= \frac{\alpha(ABC)\,\alpha(BC)\,\alpha(C)}{\alpha(BC)\,\alpha(C)} \\
p_{b3}(DCBA) &= \frac{p(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \cdot \alpha(ABCD)\,\alpha(BCD)\,\alpha(CD)\,\alpha(D) \\
\alpha_{b3}(DCBA) &= \frac{1}{\alpha(ABC)\,\alpha(BC)\,\alpha(C)}.
\end{aligned} \tag{4.27}$$

Now, we would continue[5]:

$$\begin{aligned}
p_{b4}(DCBA) &= \frac{p(ABCD)\,p(ABC)\,p(AB)\,p(A)}{p(BCD)\,p(BC)\,p(B)} \\
\alpha_{b4}(DCBA) &= \frac{\alpha(ABCD)\,\alpha(BCD)\,\alpha(CD)\,\alpha(D)}{\alpha(ABC)\,\alpha(BC)\,\alpha(C)}.
\end{aligned} \tag{4.28}$$

The general rule for the back-off arcs is then:

$$\alpha_b(w_N, w_{N-1}, \ldots, w_1) = \frac{\prod_{i=1}^{N} \alpha(w_i, \ldots, w_N)}{\prod_{i=1}^{N-1} \alpha(w_i, \ldots, w_{N-1})}. \tag{4.29}$$

Together, equations 4.24 and 4.29 give another formalism to construct the backward LM probabilities. The computation is slightly more complicated than the simple rule given in section 4.2, but the resulting probabilities are closer to a normalized (stochastic) distribution, so that it should be possible to skip the weight pushing step.

4.6 Conclusions

We motivated the idea of performing a forwards and backwards search through composed finite state machines and explained that, in order to construct the backward decoding graph, we need a reversed language model that has a similar structure and gives similar scores as the forward LM. We explained the approximation of the LM with weighted finite state acceptors and showed a constructive solution for an algorithm that results in a backward LM that assigns exactly the same scores as the forward LM. We paid special attention to the back-off structure and explained how to deal with missing N-grams. Finally, we showed

[5]For the highest order, there would be just one arc, multiplying $p \cdot \alpha$, e.g. for the four-gram model, there is an arc from state DCB to state CBA with weight $p_b(DCBA) \cdot \alpha_b(DCBA)$.

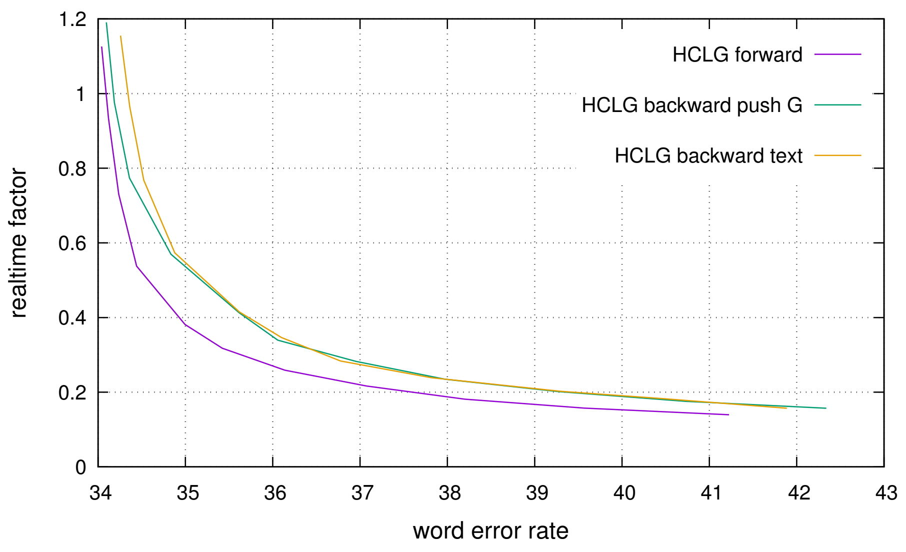


Figure 4.7: *Decoding performance of backward decoding network reported on the Eval2000 data set with a GMM model and a tri-gram language model. Shown is the relation between word error rate and real-time-factor. Better performance is indicated by curves closer to the lower left corner. We compare the performance of the backward decoding network (HCLG backward push G) with the application of the new weight pushing algorithm to the performance of a tri-gram LM trained on the reversed training texts (HCLG backward text). For comparison, we also show the performance of the forward decoding network (HCLG forward). In this case, no weight pushing is necessary.*

that the constructive algorithm for the language model reversal can be derived by a series of steps, where each step guarantees WFSA equivalence, as well as the motivation from Bayes' rule with constraints on joint word probabilities.

Up to this point, we are able to construct a backward LM that gives exactly the same scores for the reversed sentences as the forward LM, and at the same time, has the same size and a similar structure and is deterministic (except for the ϵ-arcs). For tasks, for which the backward LM is used in a pruned search (including LVCSR), it is desirable that it has yet another property - (locally) *stochasticity*. For that purpose, we can apply the alternative weight pushing algorithm as introduced in chapter 3.3.

We tested the algorithms on a bi-gram (see section 3.4) and tri-gram LM and compared it to the simpler but less exact method of reversing the training texts. Figure 4.7 shows the results of the decoding graphs from tri-gram LMs. The experimental setting is the same as in section 3.4. The performance of the backward model with exact reversal, and the one resulting from training with the reversed training texts is very similar, however in the area with low word error rates, the performance of the exact model matches the performance of the forward model more closely. As already pointed out, the performance of the forward and backward model are not necessarily the same, depending on the task (i.e. the properties of the language). However, the performances of forward and backward models are not far from each other. We can speculate, that language evolved in such a way, that humans can understand it with more ease. Therefore, we would expect that language is optimized to be easier understandable in the forward time direction, than when reversed in time. This might explain the advantage of the forward decoding.

Chapter 5

Combining forward and backward search in decoding

We introduce a speed-up technique for weighted finite state transducer (WFST) based decoders - applicable to both static and dynamic network decoders. The technique is based on the idea that one decoding pass using a wider beam can be replaced by two decoding passes with smaller beams, decoding forwards and backwards in time. The advantages of decoding backwards in time is explained in section 5.1. The approach that is followed in this thesis is to use forward and backward passes in a decoder that works with a variable beam width, controlled by the (dis)agreement of the two decoding passes. For the purpose of backwards decoding, we have to construct a backwards decoding network with certain properties, explained in section 5.2. The details of LM reversal have already been explained in chapter 4.

One possible realization of the variable beam width decoding is to run the forward and backward passes in parallel, and perform an iterative refinement with increased beam width in those places, where forward and backward decoding disagree. This is explored in section 5.3. Another realization of the basic idea is a technique we call *tracked decoding*, detailed in section 5.4. The main idea is that the second decoding pass (backwards) can use detailed information gathered from the first pass (forwards) to increase the decoding beam in places where the two passes disagree. The speed-up is achieved by using a narrow beam during the first pass, as well as in the second pass in places where no disagreement is detected. Otherwise the beam is increased to include all 'tracked' tokens. In section 5.4.4, we give an experimental validation of our method on a Wall Street Journal corpus (WSJ) decoding task. We find that our method gives a substantial speed-up of two to three times or even more, at the "more accurate" operating points of decoding where search errors are small.

5.1 Introduction: combining forward and backward search

The application that we had in mind while writing this thesis was the decoding of the most probable sequence of words in LVCSR. Given the complexity of the task - the search graph can contain up to millions of states - the resulting huge search spaces cannot be explored exhaustively. It is necessary to use heuristic pruning techniques. In this case, we have to distinguish *search errors*, which are due to the incomplete exploration of the search space, from *modeling errors*, which are due to insufficient training data or due to inaccurate models.

fwd: IS **SHERMAN ARE CONIFER AND THREE** MOST RECENT CASUALTY REPORT
bwd: IS **BADGER** A REMARK ON VANCOUVER+S MOST RECENT CASUALTY REPORT
ref: IS **THERE** A REMARK ON VANCOUVER+S MOST RECENT CASUALTY REPORT

Figure 5.1: *Forward and backward speech recognition: Example ASR result on Resource Management corpus. Bad signal quality at the start of the utterance confused almost the whole utterance in forward recognition, while it almost didn't harm the backwards decoding (only the immediate word 'there' is mis-recognized). An OOV could cause similar effects.*

The most widely used search technique in LVCSR is the Viterbi algorithm with beam search [Lowerre(1976)]. Beam search is a breadth-first style search, comparing partial paths of the same length (time-synchronously). At each time only those paths are kept and further expanded, whose partial path score is better than the current best score extended by a beam width. The beam width is a trade-off between speed and accuracy.

For many search tasks (e.g. in planning algorithms) for which the search space cannot be explored exhaustively, it is known, that if the average branching factor of the backward search graph is smaller than that of the forward graph, it is better to perform the search on the backward graph. For example [Tang and Cristo(2008)] showed, that the amount of errors in automatic speech recognition of street-city-state tuples (as used e.g. in the US) can be reduced when performing the search backwards in time, since this gives a lower 'dynamic task complexity' [Tang and Cristo(2008)]. Since forward search uses the 'history' and backward search uses the 'future', there is hope that the search errors of searching forwards and backwards are mutually independent. A path that is not promising (low scores) at the beginning is likely to be pruned by forward search, even if it has a high overall score towards the end. It has a chance not to be pruned by backwards search, because looking backwards this path has high scores at the beginning (which was the end in forward search). Figure 5.1 shows a recognition result obtained in an early stage of our experiments that demonstrates this situation. Figure 5.2 illustrates the potential of forward and backward search.

Additionally to beam search, another strategy to deal with the complexity of the task is to use multiple decoding passes, which has been a common practice already for a long time (e.g. [Murveit et al.(1993)]). Usually inexpensive and approximate models are used in a first pass to generate an intermediate representation, which is then 're-scored' using more complex models. As intermediate representation, among others, lists of N-best recognition results or lattices of possible hypothesis sequences are used.

Not only different types of models can be used in the successive decoding passes, but also different approaches to search. In [Austin et al.(1991)], the idea of performing the second pass backwards in time was introduced. As an intermediate representation, they use the active words for each time frame and the corresponding word end scores, obtained from a Viterbi beam search in the forward pass using approximate and faster models. The active words per frame are used to limit the word expansion in the backwards search, which is also a Viterbi search, and the word end scores serve as a good estimate of the path cost of the remaining speech. Thus the second pass usually takes only a fraction of the time of the first pass, so that more complex algorithms or models can be used. Alternatively, the forward pass can be sped up by using approximate models [Nguyen et al.(1993)]. A more recent re-discovery of the same idea is [Lee et al.(1998)] and [Lee and Kawahara(2009)], which use a word trellis as intermediate representation and stack decoding (A-star search) in the backward pass. Also [Cardinal et al.(2013)] use a uni-gram Viterbi backward pass,

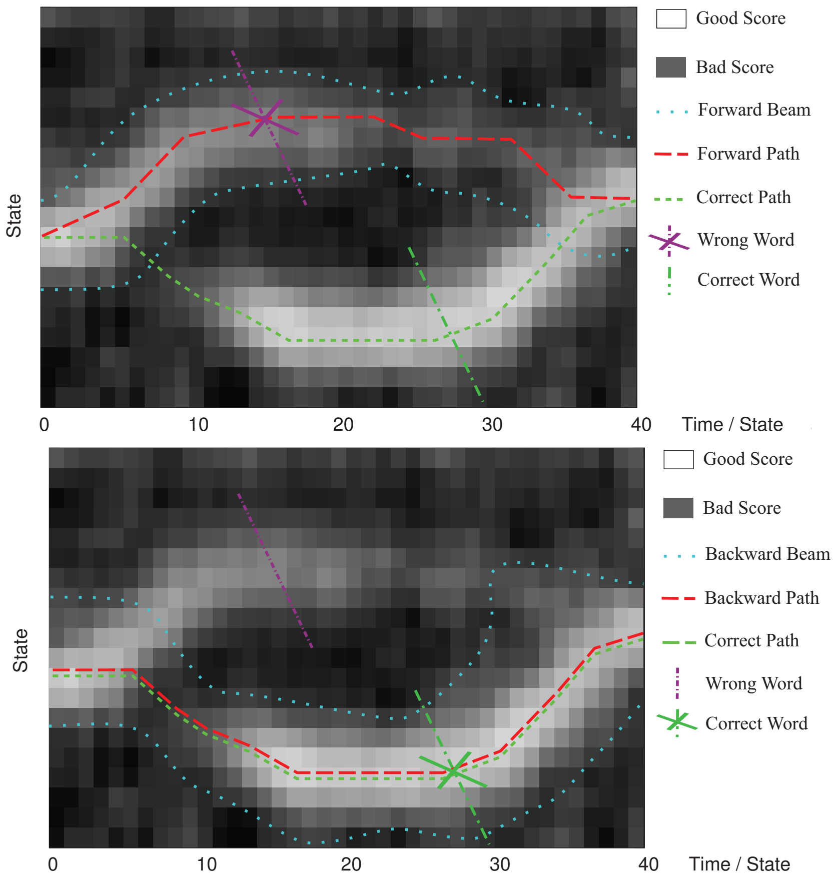


Figure 5.2: *Illustration of forward and backward search [Nolden et al.(2013)]. In the background, acoustic likelihoods for each state are shown as they evolve over time. Bright colors indicate higher probability. In the forward search (upper part), the low-score 'valley' around frame 7/8 causes the correct path (green) to fall out of the beam (dotted). The red path is chosen, but later (frames 20-30) it turns out to have poor scores. Even if it has better overall scores, the correct path can not be recovered, since it was already pruned. In the backward search (lower part), the situation is different - starting from the end, the lower path looks much more promising (frames 30-35) and the upper path falls out of the beam. The low likelihoods around frame 7/8 do not distract the recognizer this time, so the backward search does find the correct path. The illustration explains that, to a certain extent, search errors of forward and backward search are independent. Of course, with a wide-enough beam, also the forward search would find the overall best path.*

which is then used as a heuristic in A-star forward decoding with the full language model.

Opposed to these works, this work (first published in [Hannemann et al.(2013)]) focuses on using forward and backward passes that are balanced or symmetric, i.e. on using models that are similarly powerful in both passes. This has the advantage that the hypotheses of both passes can be used for comparison or combination. The idea of symmetric passes was already used by [Li et al.(2009)] and [Abo-Gannemhy et al.(2010)] (see also [Tang and Cristo(2008)]). They combine the outputs of the symmetric forward and backward passes based on LM scores or confidence measures (ROVER technique[1]). Also [Jouvet and

[1]The ROVER [Fiscus(1997)] procedure aligns the different hypotheses and relies on a voting procedure to determine the best candidate word sequence.

Fohr(2013a)] and [Jouvet and Fohr(2013b)] use the framework of [Lee and Kawahara(2009)] to ROVER two symmetric passes, and they show that the combination of forward and backward passes is especially effective in improving the performance. The follow-up work [Jouvet and Fohr(2014)] shows, that the comparison of hypotheses from the forward and backward passes is an effective confidence measure for selecting automatically transcribed data for semi-supervised LVCSR training.

The idea of our work [Hannemann et al.(2013)] is to speed up the decoding by using the (dis)agreement of the two symmetric decoding passes - decoding forwards and backwards in time. In beam search, a constant beam width is usually applied to the whole test set. We however use a decoder with a variable beam width, that is only increased in areas, where the two decoding passes disagree. There are two ways to implement this idea: Inspired by [Hannemann et al.(2013)], the authors of [Nolden et al.(2013)] showed that the comparison of the hypotheses of two symmetric forward and backward passes can be used in incremental decoding, where the search beam is extended in areas, where the two passes don't agree in the first run. As a consequence, the system uses a variable beam width and is dynamically focusing only on the parts that are difficult. Similar to all symmetric techniques mentioned so far, they use two independent forward and backward passes, which has the advantage that the two passes can run in parallel (section 5.3).

In analogy to the non-symmetric techniques, in this work we want to use the information gathered in the first pass (e.g. forwards) to guide the search of the second pass (e.g. backwards), as shown in [Hannemann et al.(2013)]. In this approach, the beam width can be adjusted for every frame, so that a more careful search (increased beam) is only carried out in areas where the two passes disagree. The speed-up is achieved by using a narrow beam during the forward pass, and in the backward pass in places where no disagreement is detected (section 5.4). The application of the presented methods assumes that a segmentation or an algorithm for end point detection is given.

5.2 Construction of a reversed decoding graph

The construction of decoding graphs with Kaldi was described in section 2.4.1. If we want to perform the search in two symmetric forward and backward decoding passes, we need two corresponding decoding graphs - $HCLG_{fwd}$ and $HCLG_{bwd}$. Both models should be equally powerful, i.e. have roughly the same accuracy and run-time requirements, and have similar structure, size and level of determinism to have optimal pruning behavior. We also want to compare the probabilities (or scores) of the outputs from the forward and backward passes (e.g. to estimate the optimal beam width). That means, we want two models $HCLG_{fwd}$ and $HCLG_{bwd}$, that ideally produce the same overall score for the same hypothesis in both the forward and the backward passes. Due to the pruned search, both passes can result in different search errors (due to the different dynamic task complexity forwards and backwards). However, both models should not make different modeling errors, hence they should assign the same scores to the same hypotheses. We also want to be able to compare the scores of partial results (paths). Therefore, also the model structure (the distribution of weights along paths) should be similar in the forward and backward passes.

Given a forward graph $HCLG_{fwd}$, the task is to obtain a backward graph $HCLG_{bwd}$ that will assign exactly the same overall score to the same utterance and will fulfill all the above stated requirements. Because our method treats disagreement between the best paths found by the two passes as a search error, we want the backward decoding graph to be equivalent to the reverse of the forward one.

The trivial solution to apply WFST reversal to $HCLG_{fwd}$ is not sufficient, since the resulting graph would not have a similar level of determinism and distribution of weights as the forward graph, i.e. it would show sub-optimal behavior when used in a pruned search. To make the resulting WFST determinizable, we would have to introduce disambiguation symbols [Mohri et al.(2008)] at different places than in the forward graph. As we already explained in section 4.2, especially the (reversed) LM component would introduce a great degree of local ambiguity.

Instead, the solution is to separately construct the time-reversed versions of H, C, L and G, and then to build a composed model $HCLG_{bwd}$ in an analogous way as the forward graph was constructed (section 2.4.1). Since the resulting $HCLG_{bwd}$ is a cyclic transducer, the conventional weight pushing algorithm cannot be used in case the total weight greater than one, as was explained in 3.1. We can resort to the alternative weight pushing introduced in section 3.3.

The time-reversed versions of H, C, L and G are again not simply the WFST reverses of the forward ones, but must be separately constructed. Depending on the task, the reversal of each component is of different complexity. The hardest input to reverse was the ARPA-format LM acceptor G. We have already given an algorithm for creating an equivalent but "time-reversed" LM in chapter 4. The reversal of H, C and L is rather trivial [Hannemann et al.(2013)] and is described in the next section. We made the code for all methods described here available as part of the Kaldi toolkit.

5.2.1 Reversing L, C and H

The construction of the reversed pronunciation lexicon transducer L_{bwd} (phones to words) is simple: the individual phone sequences (pronunciations) are reversed, and the disambiguation symbols [Mohri(1997)] (figure 2.10) are introduced after that. The disambiguation symbols now distinguish suffixes (ambiguous sequences at word endings), while in the forward case they distinguish prefixes. Figure 5.3 shows a reversed toy lexicon and the resulting transducer.

The context-dependency transducer C_{bwd} (figure 5.4) is constructed in the usual way, and looks identical to C_{fwd}. After the composition of $L_{\text{fwd}} \circ G_{\text{bwd}}$, the phonetic context window (which are the input symbols for C) is reversed in time (a-b-c to c-b-a). Therefore, to look-up the corresponding models (PDFs) in the phonetic decision tree, we have to reverse the phonetic context. Then, we look-up using the phoneme context window and the HMM state.

The HMM structure transducer H_{bwd}, is constructed in the same way as H_{fwd}, except for the reversed phonetic context. The individual (three-state) HMMs for each phone are constructed separately and the relevant PDFs are looked-up from the decision tree. Then, the phone HMMs must be reversed and weight-pushed in the log-semi-ring (including epsilon removal) to make the time-reversed transition probabilities of each state stochastic. As seen in figure 5.5, for the left-to-right HMMs, there is a simpler way to determine the transition probabilities of the reversed model: We can assign them in the reversed order. This observation is even true for more complicated symmetric structures. After reversing the phone HMMs individually, we construct the composite H_a transducer, which contains them in self-loops (example in figure 5.6). Due to the reversal of individual HMMs, the ordering of the self-loops and forward transitions changes, which doesn't matter for decoding, but needs to be considered when mapping resulting alignments at transition level.

A	ax #1
ABERDEEN	n iy d er b ae
ABOARD	dd r ao b ax
ABOVE	v ah b ax
ADD	dd ae #1
BOARD	dd r ao b #1

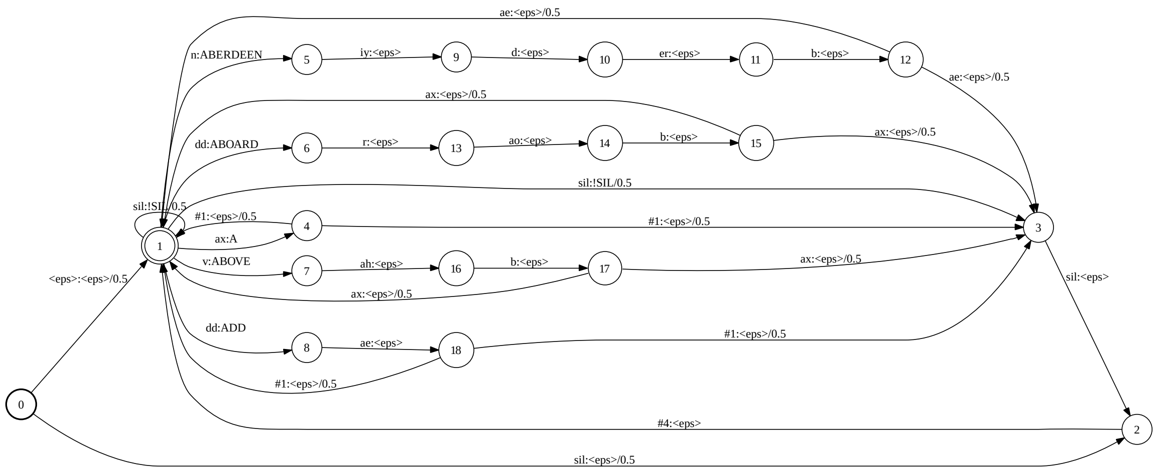


Figure 5.3: *Reversing lexicon transducer L. The phone sequences are reversed (upper part), and new disambiguation symbols (#1) are inserted afterwards. Then, the lexicon transducer is built in the same way as in the forward network (lower part).*

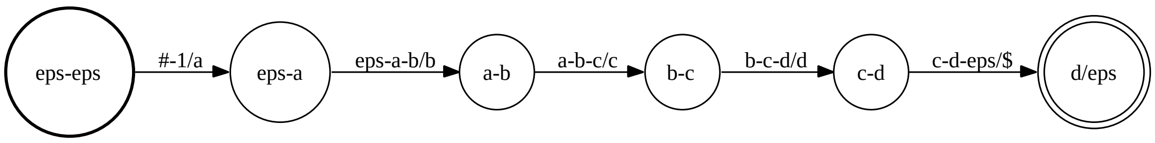


Figure 5.4: *One path of the context transducer C. The deterministic version [Mohri et al.(2008)] has a delay of two input symbols until the tri-phone-symbol is produced. The C transducer looks identically in forward and backward networks.*

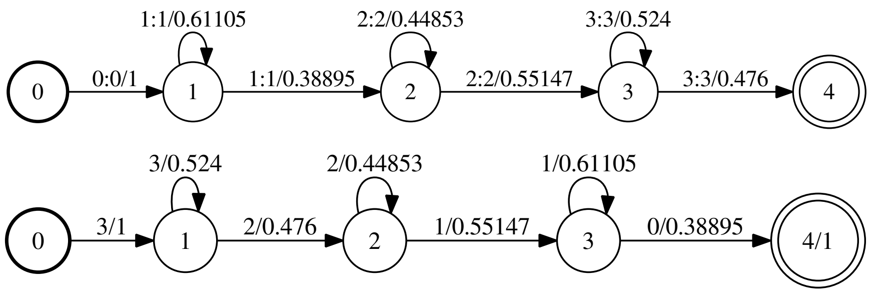


Figure 5.5: *Reversal of HMM structure for phoneme HMM: Top: forward HMM. We apply WFST reversal, weight pushing in the log-semi-ring and epsilon removal to obtain the backward HMM (bottom). We observe, that for left-to-right HMMs, the transition probabilities are exactly assigned in reverse order.*

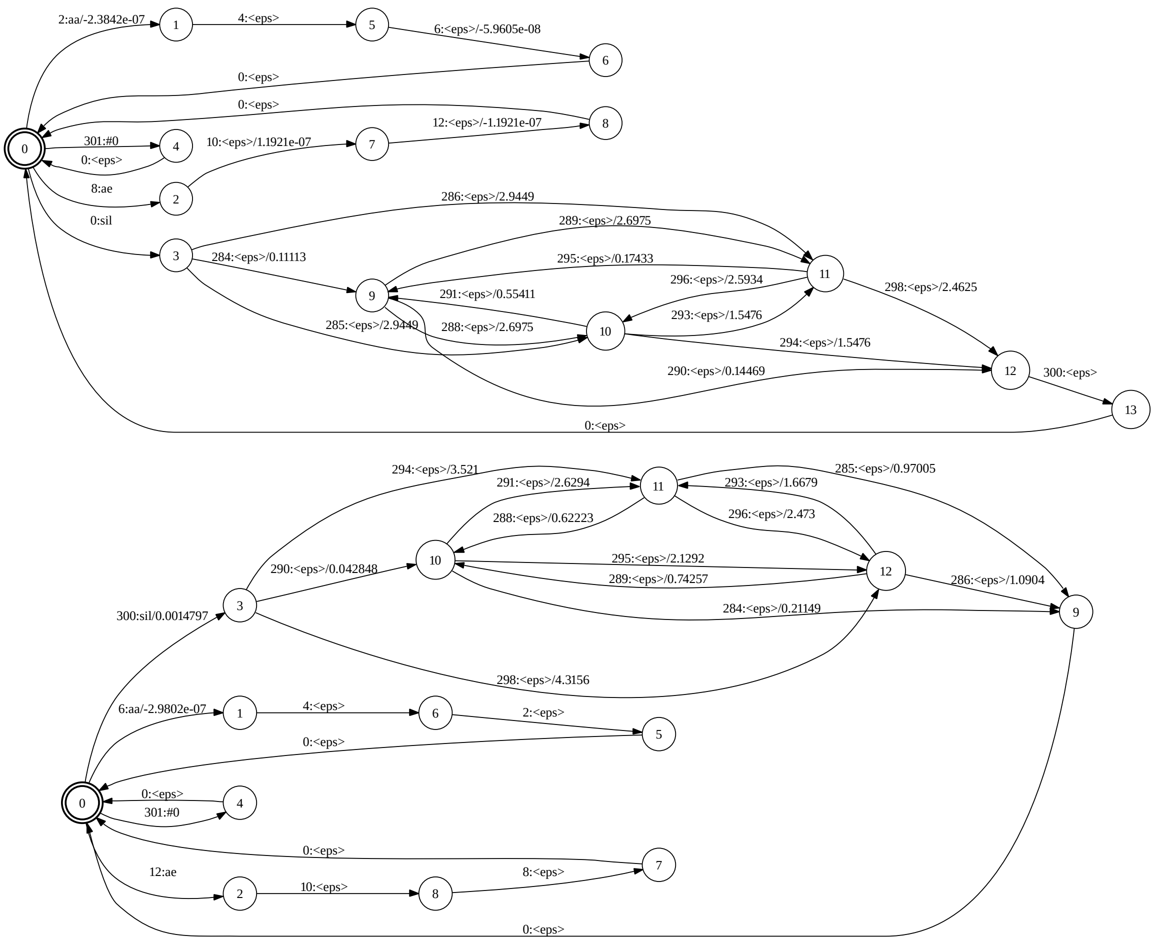


Figure 5.6: *Reversing the HMM transducer H_a - upper part: forwards transducer, lower part: reversed backwards transducer. Here, we show the mono-phone case, without self-loops. For each transducer, we show two mono-phone models (aa, ae) with three-state forward HMMs (Kaldi transition-ids 2,4,6 and 8,10,12; the odd numbers are for self-loops - not shown) and the silence model (transition ids 284-300, almost ergodic connections between states). The mono-phone models (aa,ae,silence) are reversed individually (including epsilon removal and weight pushing in the log-semi-ring) before composing H_a.*

5.3 Incremental forward and backward search

5.3.1 Finding the optimal operating point

So far, we have explained how to construct a static WFST based recognition network for backward decoding. However, the approach to the construction of the backward decoding network described in this chapter is not limited to static networks. Already [Nolden et al.(2013)] has applied the reversal of the components described here in a dynamic network decoder. Many recent dynamic network decoders are basically compiling a WFST based recognition network, but leaving out one component, which is then composed dynamically. For example, [Soltau and Saon(2009)] use a uni-gram LM (more precisely LM look-up scores) to compile a WFST based recognition network and then apply the higher-order N-gram LM dynamically. Since in our approach, all components are reversed individually, no change is necessary when dynamically composing the components.

We want to replace one decoding pass with a wide beam by a forward and backward pass with narrow beams. Thus, we must find the right operating point for the forward and backward passes. If badly chosen, the two passes will be two times slower than the single pass. The beam should be small enough to allow for substantial speed-ups, but on the other hand, the beam must be big enough to allow for a reasonable comparison of the forward and backward paths. For significant portions of the decoding, we would like to find a good path with one of the two (forward or backward) passes. If both, forward and backward decoding, are completely off, we have to increase the beam everywhere, and there is no advantage over the single pass approach.

Usually a decoder doesn't have only a single parameter (beam width) to tune [Lowerre(1976)], but a series of parameters, which are not independent of each other. The most important parameter is the global beam width, given as a log-constant, indicating how much the likelihood of partial paths can be worse than the current best partial path before the partial path gets pruned out. This is called acoustic pruning. Additionally, most decoders apply so called histogram pruning [Steinbiss et al.(1994)]. The idea is to limit the number of hypotheses being generated at a certain point in time. This is an upper limit, which is applied mainly in portions of the speech signal with high uncertainty. If the best partial hypothesis has a low score, too many other bad hypotheses are kept. In this case, too much computational effort is spent with little chance of actually finding the correct path. Thus, by limiting the maximum active tokens, the computation can be significantly reduced without much affecting the word error rate. We can selected a tightened beam limit based on a histogram over state hypothesis scores, therefore the name histogram pruning. However, for our purposes it is sufficient to think of the tokens as being ranked. To effectively limit the number of tokens to the given upper limit (called "max-tokens"), we pick the score of the token at rank "max-tokens", and use it as a tightened beam threshold, which we call the *max-tokens beam*. As soon as the number of tokens exceeds the limit, this max-tokens beam width is used for the decoding.

Depending on the architecture, other tuning parameters might be applied, too. [Nolden et al.(2012)] gives an overview of pruning techniques. Many decoders predict the max-tokens beam based on the max-tokens beam used in the last frame, to avoid generating tokens, which will be pruned anyway. [van Hamme and van Aelten(1996)] formulate this approach as an adaptive controller. Dynamic decoders usually apply tighter beams on tokens at word ends [Steinbiss et al.(1994)]. If the decoder is implemented with a re-entrant tree and token passing, where lists of tokens are attached to a state of the search network, we can impose a limit on the maximum number of tokens assigned to each state (called LM state pruning). When dynamically composing the recognition network with higher-order language models, special techniques to deal with LM scores might be effective. [Agarwal et al.(2014)] describes the use of a language model slack[2] on top of the beam to 'smear' the effect of the LM score over several frames. Also, parameters like the number of N-best paths being generated or the width of the generated lattice [Povey et al.(2012)] have a significant effect on the decoding speed. In our case, we use lattices as intermediate representation and we try to find a balance between a sufficient depth to contain the relevant hypotheses and a minimum impact on the real-time factor.

[2]The word is used in a similar way as for the slack variables used in support vector machines.

5.3.2 Tuning the beam parameters

We ran initial experiments to determine the effect of the two most important parameters: the beam width and the maximum number of active tokens applied in the histogram pruning. These two parameters are used in almost all types of decoders. We ran the experiment with the Microsoft Argon decoder (documented in [Agarwal et al.(2014)], Version 2016-02-17). It is a highly optimized dynamic network decoder, developed by Microsoft Research (mainly Geoffrey Zweig and Jasha Droppo). We report the results on the HUB5 2000 English Evaluation Speech database from LDC ("Eval 2000"). However, the word error rates reported here are not computed with the official scoring tools (NIST scoring toolkit SCTK), thus they are about 3% worse than when using this tool. The acoustic model is a deep neural network trained on a subset of Switchboard, using 1500 context-dependent tied states. For decoding, we use a tri-gram language model (7.2 million entries) that is dynamically composed.

Figure 5.7 summarizes the relation between performance (word error rate - WER) and speed (real-time-factor - RTF) on many different operating points (defined by a setting of beam width and maximum active tokens - called max-tokens). We observe that both parameters depend on each other in a non-trivial way. Therefore, we would have to test all possible combinations of parameters and then determine the optimal WER and the corresponding tuning parameters for each RTF. The resulting curve is sometimes called Pareto-optimal. For the forward and backward passes, we want to achieve the most accurate decoding using only a fraction of the decoding time of the single pass. Thus, starting from a point on the Pareto-optimal curve with low WER (and high RTF), we would move along the optimal curve towards lower RTF.

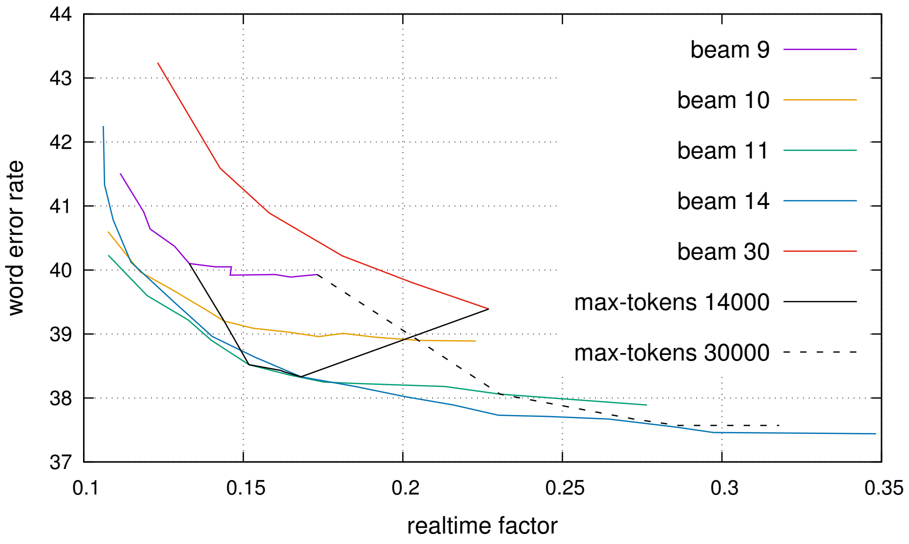


Figure 5.7: *Finding the optimal operating point on the real-time-factor and word error rate curve, while tuning the maximum number of active tokens (max-tokens) and beam width (beam). The settings of beam width and max-tokens are grouped by lines that leave one of the parameters fixed while varying the other. All curves 'beam' leave the beam width constant while running experiments with different values for max-tokens. For clarity, we don't show the curves for beam width 12,13,15 which follow a similar trend. The curves 'max-tokens' (black) measure different beam widths for a fixed number of max-tokens.*

One simplified strategy that is used most often is to treat the beam width as the main parameter that is varied, and to use a high number of max-tokens which is only effective in areas of high confusion. From figure 5.7, it is evident (black lines) that this strategy is not optimal in our case. For the lower RTFs, this setting is sub-optimal, since too many tokens are created. The decoder is in the optimal operating point, when only those tokens are generated, that actually have a chance to become the best path. In other words, in areas of high confusion (e.g. due to noisy speech), many tokens are generated, but most probably, these portions of speech will result in errors anyway. Therefore, the max-tokens beam was introduced. We should set it as low as possible, i.e. to the value, from where the WER starts increasing. From this, it is clear, that when decreasing the acoustic beam to tune to a lower RTF (and unfortunately higher WER), we should also decrease the max-tokens beam. For this reason, the optimal setting of max-tokens is to some extent proportional to the the average number of active tokens that we get, if we decode with a certain acoustic beam.

For the highly optimized token-passing decoder used here, we observed that the optimal operating point is when we set max-tokens in such a way, that for more than half of the frames, the resulting max-tokens beam is smaller than the acoustic beam. That means, max-tokens (the number of active tokens) is the dominant parameter determining the amount of computation that needs to be done for each frame. In the lower part of figure 5.8, we observe that the optimal operating point (beam 13, max-tokens 14000) is actually most of the time dominated by the max-tokens (i.e. the max-tokens beam is smaller than the beam 13 - gray line).

If we assume that max-tokens is the main factor determining the computation time, another simple strategy would be to keep the beam width fixed and to change only the max-tokens. From figure 5.7, we see that as soon as the beam is wide enough (around 13-14) this is (almost) the optimal solution for a wide range of RTFs. Only for the higher WER (above 38-39%), this strategy is slightly sub-optimal. We would have to decrease the beam width as well. What we observe is that along the Pareto-optimal curve, we have to proportionally increase both the beam width and the max-tokens. As a first approximation, it may be sufficient to choose a reasonable operating point and keep one parameter fixed while varying the other.

The Argon decoder uses the idea of [van Hamme and van Aelten(1996)], who propose an adaptive controller for steering the beam width for each frame in such a way, that the resulting number of tokens is approximately equal to the max-tokens parameter. Thus, for each frame a different (adaptive) beam is used, which is increased if less than max-tokens have been generated, and decreased, if too many tokens have been generated. The upper limit is the acoustic beam width.

Since in the optimal operating point, the max-tokens beam is the limiting parameter, it is clear that we will not improve the WER by increasing the acoustic beam width while keeping max-tokens fixed. However, what we observed when increasing the beam much further is that the WER actually increased (figure 5.7, red line, beam 30). This (perhaps) surprising non-linear effect is a particularity of the decoder, that most probably results from the adaptive beam [van Hamme and van Aelten(1996)]. In the lower part of figure 5.8, we see that the resulting max-tokens beam at acoustic beam 30 is most of the time lower than the max-tokens beam at acoustic beam 13. The dynamically adapted beam (steering to follow the max-tokens beam) is sometimes narrower than necessary (i.e. under-generates).

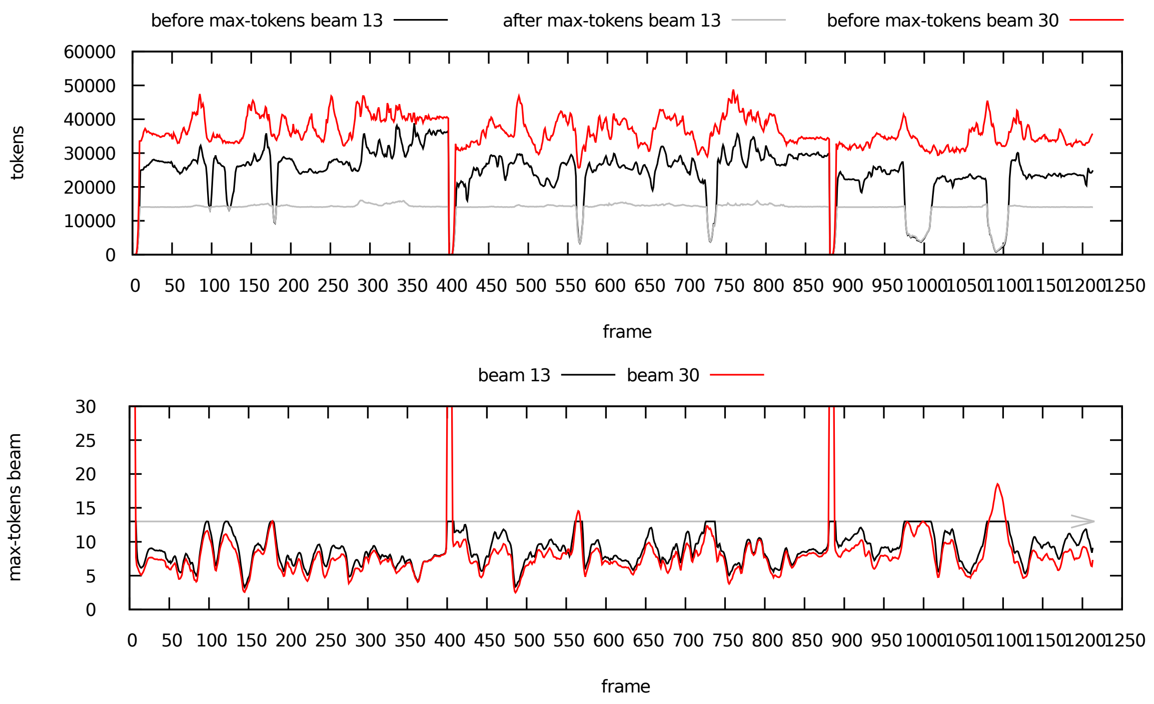


Figure 5.8: *Frame-wise scores for three files from the Eval2000 test set. Utterance boundaries are around frame 400 and 880. Compared are two different acoustic beam widths - 13 (black), which is about optimal for the RTF/WER and 30 (red), which is over-shooting. Upper part: Numbers of active tokens before and after the application of the histogram pruning ('after max-tokens 30' is not shown, as it looks very similar too 'after max-tokens 13'). Max-tokens is set to 14000 in both cases, which is the optimal setting for beam 13. Lower part: max-tokens beam (after applying the max-tokens limit).*

5.3.3 Parallel incremental forward and backward search

As introduced in section 2.6 the parallelization of the decoding of an utterance into chunks seems to be an interesting idea. According to [Maleki et al.(2014)], it is possible to split an utterance at places, where the rank of all-pairs-shortest-path matrix will converge a to singular matrix. In other words, this happens at frames, where just one token will survive. An open question is whether it is possible to automatically detect such frames in advance, in order to find the optimal segmentation of a given utterance into chunks. At the points with low rank, i.e. with few remaining active states, a small beam should be sufficient to decode them. In other words, at those points, we would expect the decoding results of the forward and backward search to agree, even if both run with a small beam. Therefore, a good segmentation for the parallelization of the decoding is to split the utterance at points, where forward and backward search agree. These thoughts lead to an approach to parallelization, which is described here.

The idea of performing a symmetric forward and backward search as introduced in section 5.1, first published in [Hannemann et al.(2013)], was used by [Nolden et al.(2013)] to implement an incremental high-level decoding algorithm, that can tune the pruning beam for individual words in an unsupervised way. As opposed to [Hannemann et al.(2013)], where the results of the first pass are integrated into the second decoding pass, both passes, forward and backward search, are run independently and symmetrically.

The incremental decoding, as described in [Nolden et al.(2013)] first runs a forward and backward decoding on the whole utterance, but with a small beam. Then the decoded words are aligned to each other. Words are considered matching, if they have the same word identity as well as a matching time boundary. All non-matching words are grouped into continuous segments which are extended by one matching word to the left and to the right. The assumption is that the acoustic alignment of words further apart than one matching word will have no effect on the alignment and acoustic score of the current word to be decoded. The identified segments of non-matching words are then decoded with an increased beam and the results are integrated into the results of the first pass. This process is iterated until the whole utterance matches. This way, the beam for each word is tuned to the minimum necessary beam. As pointed out by [Nolden et al.(2013)], for the incremental decoding of partial utterances, the left and right LM contexts of the segment need to be correctly initialized in the decoding. We also need to remember the left and right acoustic cross-word contexts, which can be achieved by remembering the states of the recognition network at the segment boundaries in the first pass – these can then serve as initial and final states for the second pass decoding.

We re-implemented the incremental forward-backward decoding in the Microsoft Argon decoder (documented in [Agarwal et al.(2014)], Version 2016-02-17), and show an additional analysis focussing on the parallelization of the approach. Since the forward and backward searches are run independently, this approach has the advantage, that forward and backward search can be run in parallel. Figure 5.9 illustrates a parallel implementation of the incremental decoding. Due to the high-level nature of the incremental forward-backward decoding, even each mis-matching segment on its own can be decoded in parallel using the approaches to parallelization described in section 2.6.

In figure 5.10, we show an analysis of the incremental forward-backward decoding on the Eval2000 database. We ran the experiment with the Microsoft Argon decoder (documented in [Agarwal et al.(2014)], Version 2016-02-17) and used the setup described in the last section. We observe, that the overall speed-up of the technique will be determined by the setup of the first pass (forward and backward) decoding. We choose an operating point from the Pareto-optimal curve in figure 5.7, that is several times faster than a well-tuned baseline (tuned for a trade-off RTF/WER, around RTF 0.3-0.5), but still in the area where the results of forward and backward decoding are partially matching. Using such a setting, we observe that after the first parallel forward/backward pass, in average approximately 50% of the complete utterances agree and thus the decoding can be finished.

For the utterances that are partially mis-matching, we find in average around 1.5 mis-matching segments ('islands') per file. That means we can achieve a speed-up of 1.5 on these utterances, and only a part of the utterance actually needs to be decoded again. Therefore, the total amount of time spent in the second pass will be much smaller than in the first pass, even if it runs at a higher RTF (increased beam). Similarly, the amount of time spent in further iterations will quickly decrease. One could reduce the scheme to a two-pass decoding and directly set the beam to the single-pass beam in the second pass (parallel forward/backward), which would still result in a significant overall speed-up. The acoustic model scores in the parallel forward/backward decoding can be shared, as well as they can be shared across the iterations. The amount of time spent in calculating the acoustic scores can be significant, however, for the particular acoustic model used in the experiments (a DNN implemented on a GPU) the computation of scores consumes only ca. 20% of the time, and this could even be further reduced by further parallelization.

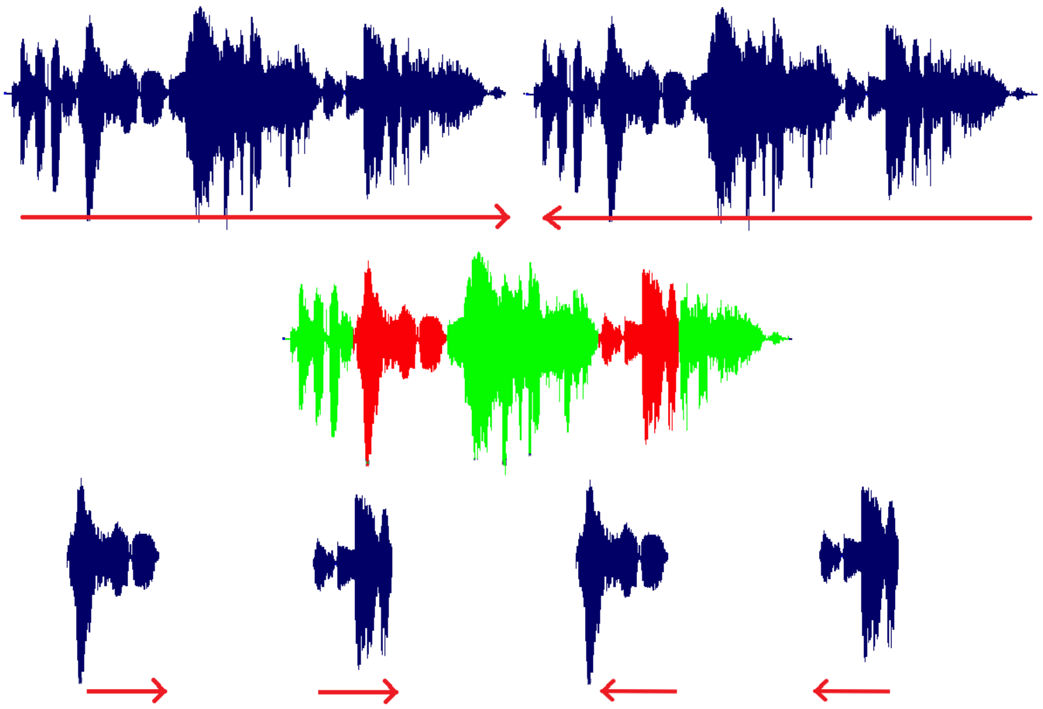


Figure 5.9: *Parallel implementation of incremental forward backward decoding [Nolden et al.(2013)]. First (upper part), two cores run a quick initial forward and backward decoding of the whole utterance with a narrow beam in parallel, then (center) the results are aligned and mis-matching regions ('islands') are identified (indicated in red). If there are no mis-matching regions, the decoding is done. Else (lower part), in a second pass, the identified mis-matching segments are decoded in parallel with a wider beam. In this example, there are two 'islands', both of them are decoded forwards and backwards, which means four cores can be used in parallel. The results of the decoded segments are integrated into the results of decoding the whole utterance, and this process is iterated until the results for the whole utterance match.*

5.4 Tracked decoding

After using independent and parallel forward/backward decoding passes in the last section, in this section, we want to use the information gathered in the first pass (e.g. forwards) to guide the search of the second pass (e.g. backwards). In this approach, the beam width can be adjusted for every frame, so that a more careful search is only carried out in the areas where the two passes disagree.

While analyzing the pruning behavior of the Kaldi decoder on the Wall Street Journal (WSJ) test set, we found that, except for a few points in time, for most of the speech frames a narrow beam is sufficient. We analyze the pruning behavior by comparing the score of the current best active token at each frame[3] and the score of the token that will ultimately result in the best overall path[4]. Figure 5.11 explains the effect of pruning with the help of one example utterance, and figure 5.12 quantitatively analyzes the score differences between the current best and the final best path. Most of the time this difference is much smaller

[3]We think of a token as a record of a particular state in HCLG that is active on a particular frame and has the accumulated score of the partial path explored so far.

[4]To determine this, we run a full decoding, back-track the best path and compute its score at each frame.

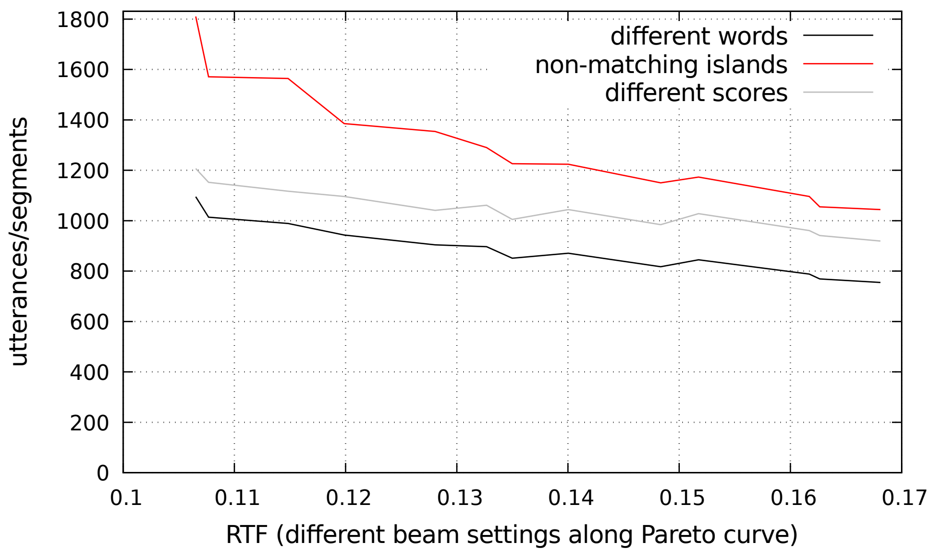


Figure 5.10: *The first iteration of the incremental forward-backward decoding on the Eval2000 test set. The test set has 1831 files, each is decoded with the forward and the backward decoder independently. Shown are the number of utterances that have either mis-matching total utterance scores ('different scores') or differ in the decoded words ('different words'). We see that requiring the exact same score is a stricter criterion than requiring that the same sequence of words are decoded. Not shown, but very closely above the line 'different scores' is also the line for 'differing state sequences', which is an even stricter criterion. To evaluate these criteria, we selected a set of operating points approximating the Pareto-optimal curve from figure 5.7 (resulting in the RTF along the x-axis). As a baseline, we assume that the single pass decoding will run at approximately 0.3-0.5 RTF, where we start approaching the lowest WER (37.3%). Therefore, we show operating points, which are two to five times faster than that (corresponds to WER 38.3%-40.3%) - this constitutes the speed-up we can expect from the technique (forward and backward decoding run in parallel). As seen from figure 5.7, going for even lower RTF would result in much worse WER. The number of non-matching stretches of words ('islands', shown as red line) is related to the number of utterances with different words (black line). The ratio is between one and two and slightly increasing towards the lower RTF.*

than the typical beam width between 10 and 15. This suggests that it would be beneficial to be able to identify those problematic areas (frames) and to only use the wide beam in these areas, while otherwise using a small beam. We aim to use the decoding results of an initial forward pass to identify the problematic frames on the backward pass.

Based on this motivation, our approach towards decoding is to do a first pass (which happens to be a forward pass) with a narrow beam, and then to do a second pass in the opposite direction, also with a narrow beam, but using knowledge obtained during the first pass. The first pass outputs a lattice with state-level alignments [Povey et al.(2012)]. Note that this lattice does not contain all partial paths explored in the first pass, but only those word-sequences that are within a specified beam of the best word-sequence (posterior pruning with lattice beam). We want to treat the paths in this lattice in a special way in the second decoding. That is,

1. We want to avoid pruning out paths that appeared in the first-pass lattice.
2. On frames where we would otherwise have pruned out those paths, we want to increase the pruning beam.

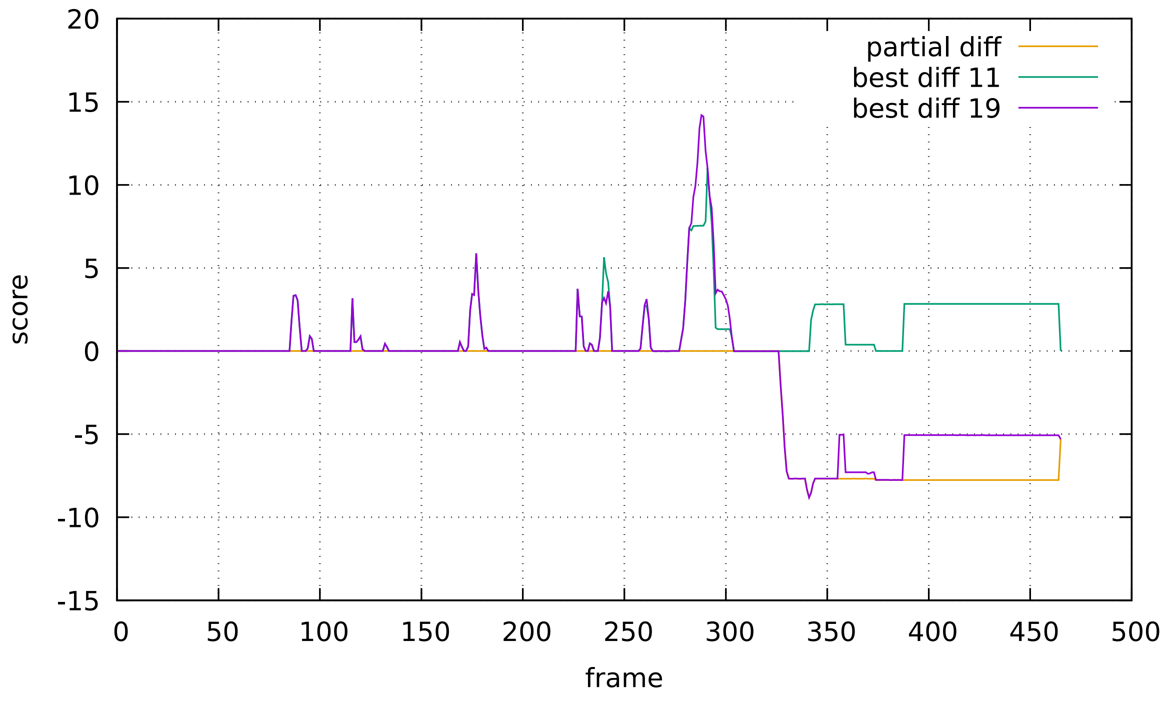


Figure 5.11: *Beam search: example utterance from the WSJ Nov'92 test set. We analyze partial scores of forward decoding for two different beam widths (beam 11.0 and 19.0). Looking at the score of the current best token for each frame, the absolute differences between beam 11.0 and beam 19.0 are small compared to the overall path score. Therefore, we show relative score differences:*
"partial diff" (yellow): the score difference of current best tokens, decoding with beam 19.0 and 11.0
"best diff 11" (green): the difference of the current best token (beam 11.0) and the partial score of the final best path (beam 11.0) at same frame - i.e. which is only known after finishing the decoding.
"best diff 19" (magenta): the difference of the current best token (beam 11.0) and the partial score of the final best path at beam 19.0.
We see ("partial diff"), that beginning around frame 325, the search with beam 19.0 found a better path, so the difference becomes negative. It is also observable ("best diff 11"), that most of the time, the current best partial score is also the score of the (future) best path, which means a small beam would be sufficient. Only at a few places, the path that is going to win, is off for a short time. Around frame 290, we miss the final winning path ("best diff 19"), if the beam is too small. Not immediately, but only after frame 325, this results in better overall scores ("best diff 19" vs. "best diff 11").

5.4.1 Tracking tokens with an arc-lattice

During decoding, we need to be able to identify which active tokens in our second-pass decoder correspond to paths in the first-pass lattice. One possible way to do this would be to designate a set of context-dependent HMM states (PDF-ids) on each frame that are "special" because they appear in the first pass lattices. However, we did not pursue this because it could lead to too many irrelevant tokens being kept in the beam. Instead, we chose to identify those paths through the second-pass decoding graph that correspond to paths in the first-pass lattice. We implemented this as a separate step, outside of the decoder code. It takes the standard output lattice from the first pass, and processes it into something we call an *arc-lattice*, whose symbols identify arcs (see below) in our second-pass decoding

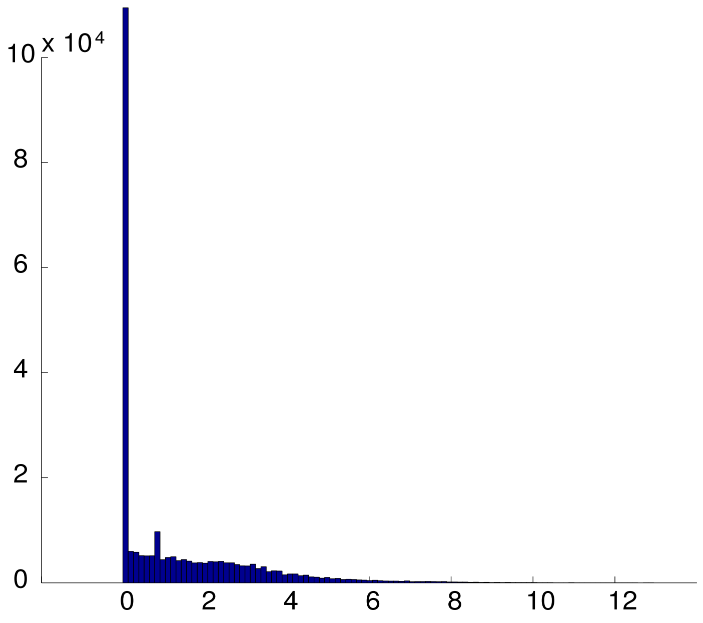


Figure 5.12: *Histogram of score differences: Shown are the scores of the current best partial path at each frame minus the partial score of the path that is going to be the final best path, not necessarily the correct one (decode beam 13.0, WSJ Nov'92 test set at WER 10.8%).*

graph HCLG_{2nd}. We explain the arc-lattice generation process below (Section 5.4.3).

The second-pass decoder, which we will refer to as our *tracking decoder*, is a lattice-generating decoder that takes an extra input[5], namely the arc-lattices for each utterance. Let a *token* be a record of a particular state in HCLG that is active on a particular frame. Our tracking decoder gives tokens an extra, Boolean property that identifies whether they are *tracked* or not. A tracked token is one that corresponds to a state in the arc-lattice. Tracked tokens are never pruned. Tracked tokens are also used to determine the pruning beam used on each frame.

5.4.2 Beam-width policy

For the second-pass decoding with the tracking decoder, we use the *tracked* tokens to determine the beam width to use for each frame. Here we describe the policy we use to set the beam width. The decoder has three configurable values that specify how it sets the frame-specific beam: the *beam*, the *max-beam* and the *extra-beam*. On a particular frame, let the score difference between the highest-score token and the lowest-score tracked token be D. Then the beam width on that frame is given by:

$$\max(\text{beam}, \min(\text{max-beam}, D + \text{extra-beam})).$$

Figure 5.13 illustrates the beam width policy. Unless otherwise specified we let *extra-beam* be zero and *max-beam* be large[6]; we try various values of the *beam* for our experiments here[7].

[5]Usually, inputs are the decoding graph $HCLG$, the acoustic model and the acoustic features.

[6]This is system-specific. We e.g. selected 100 for this task in Kaldi, although this may be too large.

[7]For a few utterances, the decoding does not terminate in a final state, when decoding with a small beam. This poses a problem for the reversal and the creation of the arc-lattice. In these cases, we used an increased final-beam to not prune away the path that leads to the final state.

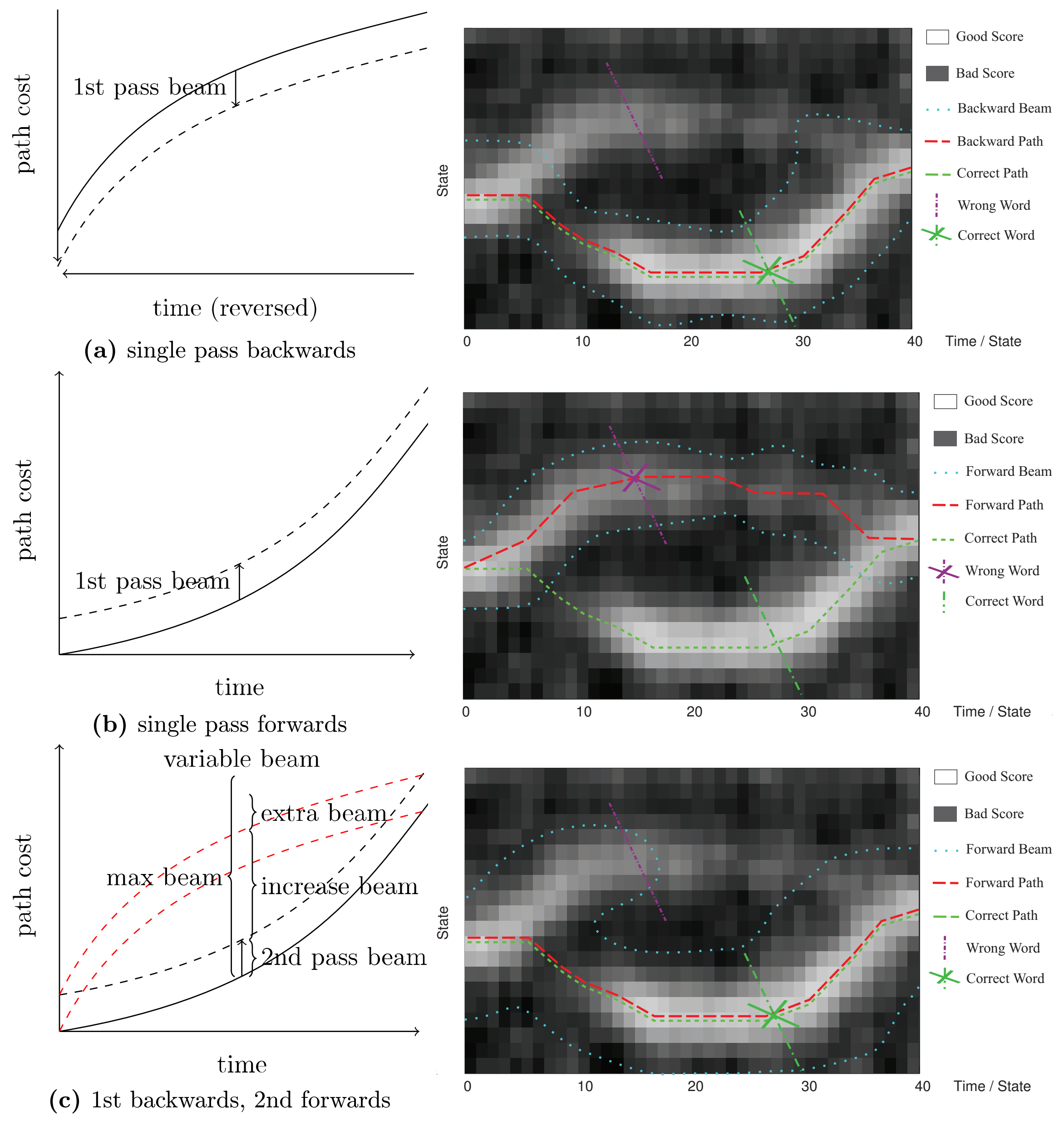


Figure 5.13: *Tracked decoding example illustrating the beam width policy. The illustration of forward and backward search is repeated from figure 5.2 [Nolden et al.(2013)].*
a) Single pass backward decoding in reversed time direction; shows the accumulated scores of the best path. Towards the left, the partial acoustic scores are worse, thus the accumulated log-scores increase faster.
b) Single pass forward decoding. At the beginning good acoustic scores, but towards the end the partial log-scores increase faster. The overall path is worse due to pruning.
c) Backward-forward tracked decoding: Paths in the first pass lattice (red) are time-reversed and tracked. Since the scores of the tracked tokens are farther off than the initial 2nd pass beam, the beam is increased to include all tracked tokens, plus an extra beam. If the beam exceeds the max-beam, it is not further increased, but all tracked tokens are still kept.

Regardless of the beam-width, we never prune away the *tracked* tokens. Note that even if we keep the beam equal to the single pass *beam* during the tracked second pass, our method is doing more than simply choosing the best path from two (forward and backward) passes, because for paths found by the first-pass search, it is possible to "recombine" with paths that were found by the second-pass search. Some parts of the utterance might have scores similar to figure 5.13, i.e. be advantageous for backwards decoding; other parts might have the opposite characteristic. If partial paths of tracked tokens and second-pass tokens meet in the same state, they can recombine and thus we would continue decoding the rest of the utterance with the maximum of the two partial scores (likelihoods). Therefore, the combined path can have a better score than either two single paths.

5.4.3 Generation of the arc-lattice

As mentioned above, the arc-lattice is a special kind of lattice that allows us to identify arcs in HCLG_{2nd} that were present in the first-pass lattice. This means there is a path in the lattice, that went through the corresponding state in HCLG_{1st} at the given time. The arc-lattice is an *acceptor* FST, i.e. it has only one symbol on each arc. These symbols correspond to arcs in HCLG_{2nd}. We first construct a mapping between integers and the individual arcs in HCLG_{2nd}; this involves creating tables for mapping pairs of (node, arc) to integers, because the product of (#states) $\times$ (maximum #arcs) may be greater than the 32-bit integer range.

We now describe how we create the arc-lattice. First, let us point out that the standard Kaldi lattices [Povey et al.(2012)] (and also HCLG) are WFSTs whose input symbols correspond to integers called *transition-ids* and whose output symbols correspond to words. The transition-ids may be mapped to *PDF-ids*, which correspond to context-dependent HMM-states (the transition-ids contain more information about the exact transition used, but this is not needed here). We first map the transition-ids in the input lattice to PDF-ids, and also map the input symbols of HCLG_{2nd} from transition-ids to PDF-ids. This is necessary because the order of self-loops versus "forward transitions" on the forward versus backward graphs differ, which makes the sequences of transition-ids differ even for paths that are "really" the same; this issue does not arise with PDF-ids. We then change the output symbols of HCLG_{2nd} (which were previously words) to symbols identifying the arc in HCLG_{2nd} (integer mapping to (node,arc) pair). Let the resulting FST be called HCLG_{arc}; it has the same structure as HCLG_{2nd} but different labels on the arcs.

After doing the symbol mappings described above, we reverse the first-pass lattice LAT_{1st} to retrieve the labels in reversed time order and obtain LAT_{rev}. We map the input labels from transition-ids to PDF-ids to correct for the self-loop order, "project it on the input", which means we keep only the input labels (PDF-ids) and then we remove the weights (they will be contained in HCLG_{2nd}) and remove epsilon arcs. Now, we can compose $LAT_{rev} \circ \text{HCLG}_{arc}$ to obtain a transducer from PDF-id sequences in the lattice (input) to sequences of symbols for HCLG_{2nd} arcs (output).

Lattice-determinization [Povey et al.(2012)] is an operation in a special semi-ring, that keeps only the best path for a symbol sequence (e.g. the best segmentation), but it is assigned the weight of all paths with that symbol sequence. We apply lattice-determinization on the resulting transducer to retain only the best path for each sequence of PDF-ids in the lattice. As a result, for each sequence of PDF-ids, we have a single path of HCLG_{2nd} arcs. Then, we project on the output, i.e. we keep only the output labels corresponding to arcs in HCLG$_{\text{2nd}}$, and we determinize again[8] – this time on the output labels, i.e. we keep only the best path for each arc sequence. The result is an acceptor lattice for HCLG$_{2nd}$ arcs which we call LAT_{arc}. Since the first-pass lattice contains the alignments (the sequence of PDF-ids), also the resulting arc-lattice contains timing information (it is a trellis). The timing information is represented in sequences of HCLG$_{\text{2nd}}$-arcs. For example, we see sequences of repeated arcs on self-loops, followed by a forward arc. Algorithm 2 summarizes the arc-lattice generation.

Algorithm 2 *Generation of arc-lattices (graph-state-lattices):*

```
1. Map $HCLG_{2nd}$ to PDF-to-Arc transducer $HCLG_{arc}$:
    (a) $HCLG_{2nd}$ : transduces PDF-ids into words
    (b) Encode $HCLG_{2nd}$ (node-id, arc-id) into output symbols.
    (c) Map input to be self-loop order independent.
2. Map first-pass lattice $LAT_{1st}$ to $LAT_{rev}$:
    (a) Map input (self-loops), project on input, remove weights.
    (b) Time reverse lattice and remove epsilons.
3. Compose: $LAT_{arc} = LAT_{rev} \circ HCLG_{arc}$:
    (a) Obtains sequences of $HCLG_{2nd}$ arcs for PDF sequence in lattice.
    (b) $det(LAT_{arc})$: Lattice-determinize (on PDF-ids) in special semi-ring
     $\rightarrow$ single $HCLG_{2nd}$ path left for each sequence of PDFs.
    (c) Project to $HCLG_{2nd}$ (node, arc) symbols, determinize again.
$\rightarrow$ The output is an acceptor lattice for $HCLG_{2nd}$ graph arcs.
```

During decoding, a token is *tracked* and never pruned if it was reached by a sequence of HCLG$_{\text{2nd}}$-arcs in the arc-lattice that correspond to a path in the first pass lattice. We could think of it in this way, that at each time step, there is a set of states, which we should keep in any circumstances. Since we explore backwards and with a wider beam than in the forward pass, it is possible that these states are reached by other paths than those used in the arc-lattice, and that these paths (and their corresponding tokens) have a better score than the those following the arc-lattice. In this case, the tokens recombine, i.e. we only keep the better token. We implemented it in this way, that the winning token inherits the status of being tracked, so that we still keep tracking the path.

It needs to be pointed out, that the implementation with the external creation of the arc-lattices is just one possibility. It would be also possible to compute a mapping of graph states between HCLG$_{\text{1st}}$ and HCLG$_{\text{2nd}}$ at the time of graph construction, and to provide this mapping together with the two graphs as input to the tracked decoding. This implementation has the advantage, that fewer changes to the decoder had to be made, and that the memory consumption is smaller.

[8]Using the standard determinization algorithm.

5.4.4 Experimental results

We tested the proposed forward-backward tracked decoding on the Wall Street Journal (WSJ) November'92 open-vocabulary test set (333 utterances) using a standard tri-phone HMM+GMM system (Kaldi recipe 'tri2a' [Povey et al.(2011)], trained on the 'si84' portion of WSJ). The experiments were conducted with the extended 146k vocabulary using the pruned tri-gram language model 'bd_tgpr' that was trained on all WSJ training texts. The lattices [Povey et al.(2012)] were generated with a lattice beam of 4.0.

We can detect and evaluate search errors by aligning the recognition outputs to a decoding with a very wide beam. We align the results of both forward and (reversed) backward decodings with the wide-beam-decoding. Table 5.1 shows an example of such an alignment. We implemented a 4D-Levenshtein edit-distance algorithm for that purpose. Table 5.2 confirms the intuition that forward and backward search errors are independent. With the help of the tracked forward-backward decoding, most of the search errors were eliminated.

```
f: BRIAN J. KILLING CHAIRMAN OF BELL - ATLANTA X.  INVESTMENT
     .   .     S        .     .   .   .    .     S       .
b: BRIAN J.  DAILY  CHAIRMAN OF BELL AND LAND  SIX INVESTMENT
     .   .     .        .     .   .   I    S     S       .
p: BRIAN J.  DAILY  CHAIRMAN OF BELL - ATLANTA ITS INVESTMENT
     .   .     .        .     .   .   .    .     .       .
w: BRIAN J.  DAILY  CHAIRMAN OF BELL - ATLANTA ITS INVESTMENT
r: BRIAN J.  KELLY  CHAIRMAN OF BELL - ATLANTIC'S  INVESTMENT
```

Table 5.1: *Analysis of search errors on the WSJ Nov'92 test set by aligning forward and backward search errors (with beam 11.0) against a decoding with a wide beam (29.0).*
Shown are the outputs of forward decoding (f), backwards decoding (b) and forward-backward 'ping-pong' decoding (p), aligned to a decoding with very wide beam (w) and reference transcription (r). The search errors are indicated by 'I' for insertion, 'S' for substitution and '-' for deletion.

beam width	forward errors	backward errors	co-occur	ping-pong
11.0	144	230	32	**14**
13.0	84	108	14	**6**

Table 5.2: *Analysis of search errors on WSJ Nov'92 test set by aligning against a wide beam (29.0). The co-occurrence of an error ('co-occur') means that both, forward and backward pass, made an error in the same alignment position. It does not necessarily mean that both produced the same error. With two-pass 'pingpong' decoding, all independent search errors were corrected (all those that are not co-occurring), and even a good portion of the co-occurring errors could be removed.*

We measured the total elapsed time for the two-pass forward and backward (tracked) decoding and relate it to the word error rate (WER). The real-time factor was measured on a single core of an Intel(R) CPU i5-2500 (3.3GHz, 8GB RAM). The results in figure 5.14 show, that for the lowest word error rates (WER$<$ 10.5), the two-pass tracked decoding runs about 2-3 times faster than the individual forward/backward passes at the same WER. This corresponds to the "more accurate" operating points of decoding where search errors are small. However, in this setup, the speed-ups are diminishing for operating points faster than $\approx$ 0.6 real-time using our method. The issue seems to be that if the beams are

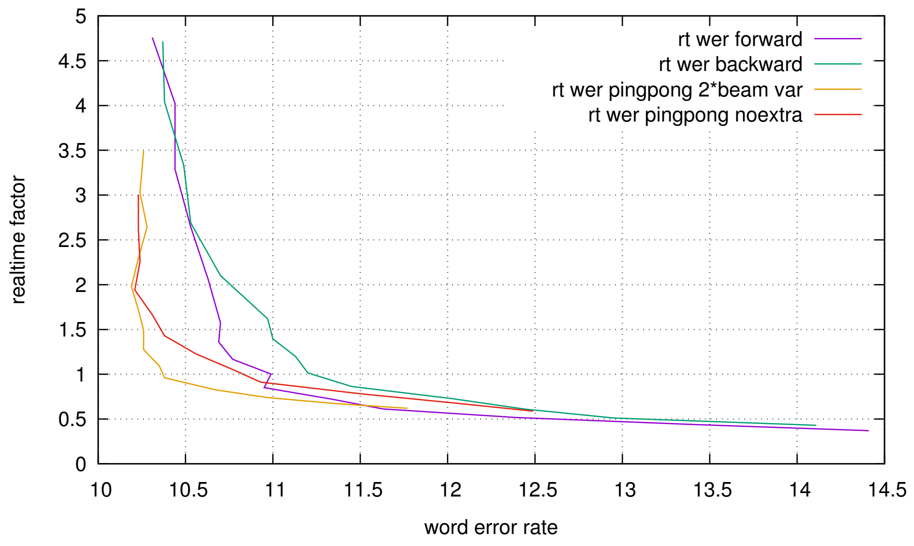


Figure 5.14: *Performance of tracked decoding: Shown are curves for word error rate vs. real-time factor on the WSJ Nov'92 test set. For single-pass decodings, the beam varies between 10-18, for the two-pass ('pingpong') decoding the beam varies between 7-13. We used extrabeam* $= 0$ *and found maxbeam* $= 2 \cdot$*beam as a good compromise between speed and accuracy. The lattice-beam is* 4.0*, but for beam* < 10.0 *we decrease it step-wise by* 0.5 *down to* 0.5*. As will be explained in section 5.4.5: We compare the variable-beam decoding ('2beam var', orange) to a decoding without generating extra tokens in the variable beam ('noextra', red) by setting maxbeam* $=$ *beam, which shows the additional benefit of the variable beam over just combining lattices of forward and backward passes.*

too narrow, the two decoding passes disagree substantially and too much effort is spent in decoding with a widened beam in areas that disagree. Also, [Nolden et al.(2013)] points out that a too narrow beam could lead to a degenerated search, where both passes produce the same errors (e.g. focussing on silence and noise models, which are symmetric). The WER curve in figure 5.14 is not always smooth, which points to the fact that fixing a search error does not necessarily mean fixing a word error.

5.4.5 Importance of beam parameters

The proposed decoder has several parameters to tune: forward beam, backward beam, lattice-beam, extra-beam and max-beam. This section analyzes the importance and typical settings for those parameters. Since the WER-RTF curves for single-pass forward and backward decodings are similar, we typically set the forward beam and backward beam to the same value. In the backward pass (tracked decoding), we have three types of tokens:

- Tokens that are generated in the normal way within the narrow beam.
- Tracked tokens, which are never pruned.
- Extra tokens, which are generated due to the increased variable beam, which is the difference between the best token and the worst tracked token plus extra beam.

Looking at the different components of the beam-width-policy (section 5.4.2), there seem to be two strategies one could pursue: either track many tokens and try to combine good forward and backward paths, while limiting the generation of extra tokens, or just track

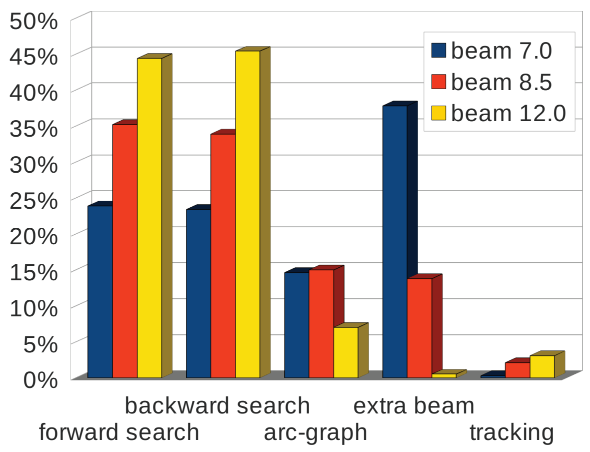


Figure 5.15: *Profiling the tracked two-pass decoding on a single core CPU. Shown is the percentage of time spent in different parts of the algorithm at three operating points (beam 8.5 as optimal, others as not optimal). The first pass is the lattice-generating 'forward search' (which is also our single-pass baseline) and the second pass can be seen as consisting of a) normal backward decoding (column 'backward search'), b) generating the arc-lattice ('arc-graph'), c) additionally tracking tokens from the first pass ('tracking') and d) generating extra tokens within the increased variable beam ('extra beam'). The acoustic scores were not cached between the two passes. The contributions of 'arc-graph' and 'tracking' (together $< 20\%$) could be possibly optimized by a better implementation, but the two individual passes constitute a lower bound (around 70% of the time is spent there). For beam 7.0, we used lattice-beam 1.0, and got 11.36% WER at 0.85 RTF. For beam 8.5: lattice-beam 4.0, 10.38% WER at 1.06 RTF. For beam 12.0: lattice-beam 4.0, 10.23% WER at 2.63 RTF.*

few tokens and generate many extra tokens up to the variable beam difference. To analyze the importance of the extra tokens, we can compare the proposed tracked decoding using the variable beam (which is the distance of the best active token to the worst tracked token plus the extra-beam) to a decoding without generating extra tokens. We can achieve this by limiting the beam to $maxbeam = beam$ and thus effectively disabling the variable beam. Since tokens 'tracked' by the first-pass lattice are kept anyway, this effectively corresponds to combining the lattices of the forward and backward pass. Figure 5.14 ('2beam' vs. 'noextra') shows that creating extra tokens within the variable beam gives a substantial improvement on top of that. This shows that the extra tokens are important, especially for the operating points with low WERs.

To get an insight on the optimal size of the forward/backward beam, we profiled the tracked decoding in figure 5.15. We observe, that the time spent in the two individual decoding passes (without tracking / extra tokens) is the dominant factor - thus we want to keep this value small. However, if we reduce the beam too much, we observe (figure 5.14 for error rates $> 11.5\%$) that the two-pass decoding is no longer better than the single-pass decoding. From figure 5.15 we see, that for narrow beam widths, most of the time is consumed in the generation of extra tokens, which effectively means decoding with a higher beam. Below a certain beam width (11% in figure 5.14) the error rates in the single passes grow rapidly with only little RTF to gain. This means that the divergence between the best paths from forward and backward decoding is too big, so that the algorithm has to increase the variable beam a lot to track the first pass tokens. The max-beam parameter limits the variable beam, so that in these situations, the decoding is not slowed down too much (at the cost of higher WER).

With an optimal setting of the beam, we can reach a significant WER decrease by just generating a small amount of extra tokens in the variable beam ('extra beam' in figure 5.15, optimal around beam 8.5). This point corresponds to the turning point in figure 5.14 (around RTF 1.0) - it is the 'sweet spot'. Above that, though little time needs to be spent for tracking and for generating extra tokens, too much time is spent in the individual forward/backward decodings, and the overall RTF increases rapidly.

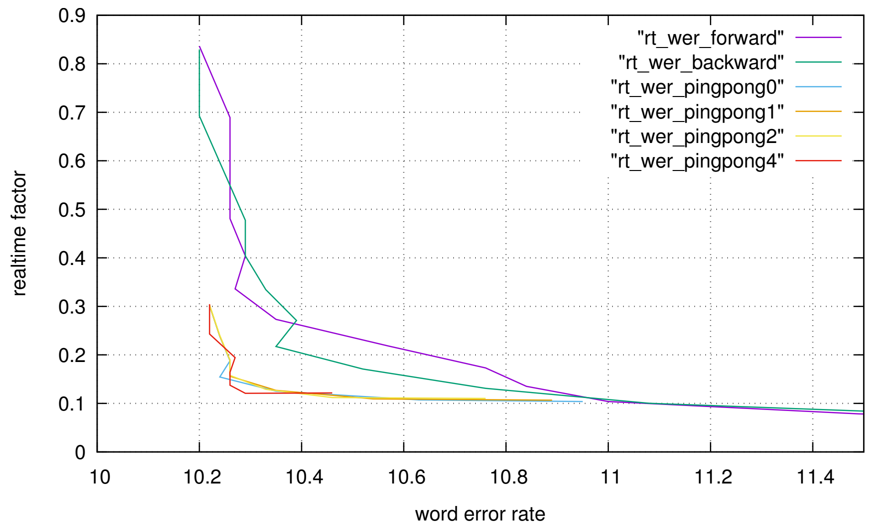


Figure 5.16: *Testing the extra-beam: WER vs. real-time factor on WSJ Nov'92 test set using the bi-gram LM with 5k vocabulary. We set the parameters to lattice-beam 6.0 and max-beam 100.0 and varied the extra-beam from 0.0 to 4.0 'pingpong0...4'. All settings of extra-beam resulted in very similar curves.*

In an experiment using a smaller vocabulary, we tested the influence of the extra-beam parameter. Figure 5.16 suggests, that this parameter doesn't have any significant influence. It seems that increasing this parameter has a similar effect to simply decoding with a wider beam. Therefore, we set the extra-beam to zero in the further experiments (also in figure 5.14). Now, in figure 5.17, we investigate the effect of the lattice-beam. We can see, that pruning the lattice with different beams and generating the corresponding arc-lattice has mainly the effect, that larger lattices result in higher RTF, visible in the area with the higher WER. Thus, we want to make the lattices as small as possible. However, for the most accurate operating points with low WER, we want to have a wider lattice that is more likely to contain the best path. Figure 5.17 suggests that it seems to be a good strategy to increase the lattice-beam linearly with the beam. We can set an upper bound of 4.0, which is enough to get good results in re-scoring the lattice.

Finally, after tuning all other parameters, we investigate different settings of the max-beam parameter. Figure 5.18 suggests, that the exact setting of the parameter max-beam doesn't influence the potential speed-up of the technique (the 'sweet-spot'), but mainly influences the shape of the curve from the 'sweet spot' towards the higher WER. Using no limit for the beam even for huge divergences between forward and backward pass seems wasteful. Therefore, it seems to be reasonable (figure 5.18) to increase the max-beam slowly with increasing beam. Once a reasonable beam has been reached, the divergence between forward and backward passes gets smaller, and the max-beam is no longer needed.

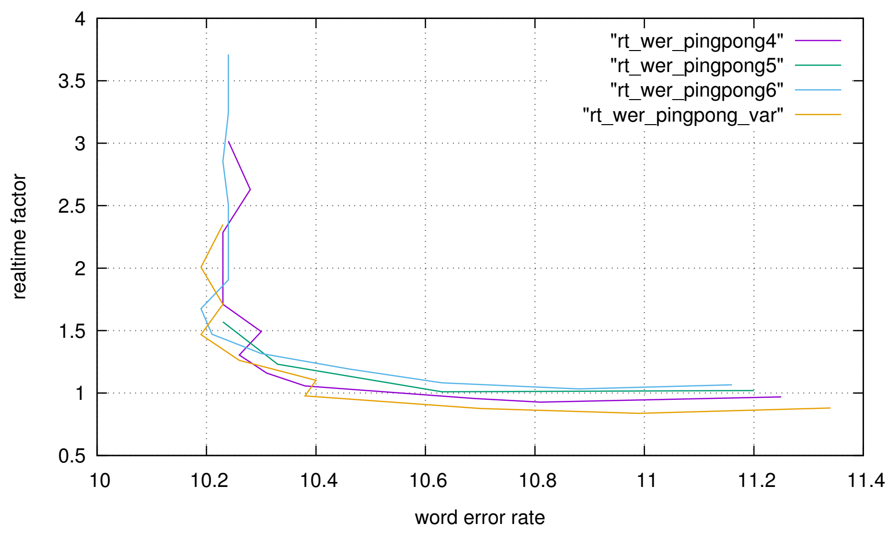


Figure 5.17: *Analyzing the effect of the lattice-beam on WSJ Nov'92 test set using the big bi-gram LM with 147k vocabulary. We set the parameters to extrabeam* $= 0.0$ *and maxbeam* $= 100.0$ *and varied the lattice-beam from 4.0 to 6.0 (curves 'pingpong{4,5,6}'). Then, we tried increasing the lattice beam linearly from 0.5 to 5.0 (curve 'pingpong_var'), i.e. we started with lattice-beam 0.5 at beam 6.5 and increased it until we had 5.0 at beam 11.0.*

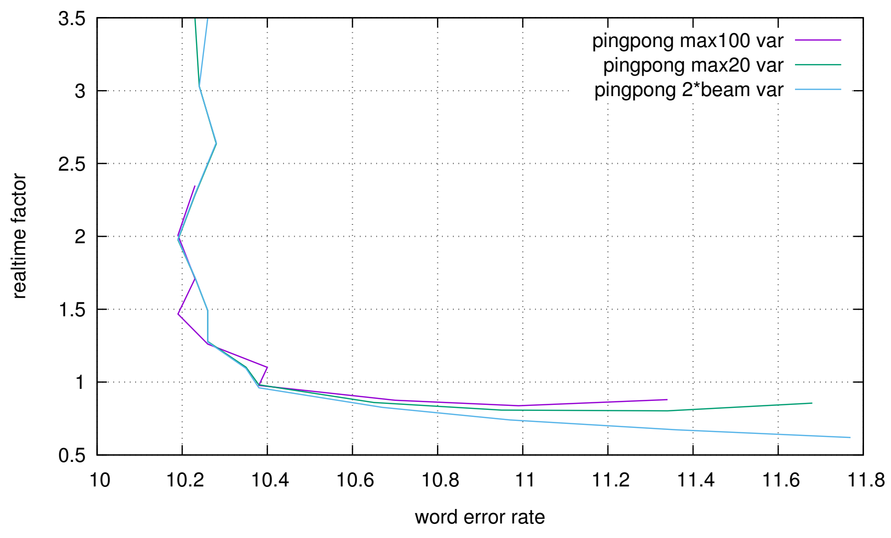


Figure 5.18: *Analyzing the effect of different max-beam settings on WSJ Nov'92 test set using the big bi-gram LM with 147k vocabulary. As already explored, we set the parameters to extrabeam* $= 0.0$ *and linearly increased the lattice-beam from 0.5 to a maximum of 4.0. Now, we compare three strategies of setting max-beam: a) using a fixed max-beam of 100.0 b) using a fixed max-beam of 20.0 c) changing the max-beam linearly with the beam:* $maxbeam = 2 \cdot beam$. *We also tried* $maxbeam = beam$, *which had slightly worse performance for WER* > 11.0. *We see, that using a fixed maxbeam leads to a slight increase of RTF for the lowest WER, which indicates, that too many extra tokens are generated due to the variable beam.*

5.5 Conclusions

We proposed how to integrate information from two symmetric decoding passes, decoding forwards and then backwards in time. In order to implement this we needed to construct reverse decoding networks that assign exactly the same scores as the forward decoding.

We explored two implementations, one approach using an incremental decoding that can be easily parallelized, and another approach that allows for a more fine-grained steering of the beam by tracking the paths from a first-pass lattice in the second pass. More specifically, in the second pass of tracked decoding, we modify the pruning behavior of the decoder to treat specially tokens that were part of successful paths in the first pass, and to increase the decoding beam for parts of the utterance where the forward and backward decoding disagree. Our decoding method results in a roughly two to three-fold speed-up.

The proposed speed-up method can be applied in any ASR based technology, for example in the fast generation of lattices for audio indexing. The tracked decoding could be used to generate lattices that contain certain desired paths (e.g. the reference forced alignment for discriminative training).

Our algorithms uses the WFST approach [Mohri et al.(2008)] to speech recognition. For the tracked decoding, other speed-up techniques such as acoustic look-ahead [Nolden et al.(2011)] and various types of fast acoustic score computation are also applicable. We expect that those methods can combined with the technique describe here and bring complementary speed-ups.

Chapter 6

Conclusions

6.1 Summary of the findings

In this thesis, we have introduced the idea of symmetrically decoding forwards and backwards in time. For tasks like LVCSR decoding, the search space cannot be explored exhaustively. For some tasks, the pruned backward search is more efficient than the the forward search. Moreover, we showed experimentally, that the search errors of forward and backward search are mutually independent. Forward search prunes based on the “history” and backward search prunes based on the “future”. To be able to concentrate on search errors rather than on modeling errors, we require both decoding passes to be symmetric – i.e. both models are equally powerful and are constructed to assign exactly the same probabilities to hypotheses (paths, word sequences). The symmetry of both passes allows us to compare the recognition results of forward and backward decoding. Each difference detects a search error. We have shown, that for most of the time frames in beam search decoding, a very narrow beam is sufficient to keep the final best path. Therefore, we are able to decode with a variable beam width – we use a small baseline beam and only increase it in places, where the forward and backward searches disagree.

One possible realization of the variable beam width decoding is to run the forward and backward passes in parallel, and to iteratively refine the decoding (by increasing the beam width) in places, where both passes disagree. We showed that, for about 50% of the utterances, the results already match after the first iteration. For the remaining utterances, the stretches of mis-matching words (in average 1.5 per utterance) can be decoded in parallel. This approach is very similar to chunk based decoding and is a high-level technique that can be applied additionally to other coarse-grained and fine-grained parallelization techniques.

Another realization of the variable beam width is the tracked decoding presented in this thesis, which runs forward and backward decoding sequentially. During the second pass (tracked decoding, backwards), we are able to identify which active tokens correspond to paths that were present in the first-pass lattice. These are called tracked tokens and they are never pruned, regardless of the beam width. We track tokens with an acceptor lattice of graph-states of the backward decoding graph, which is generated from the first pass lattice with a series of WFST operations. Tracked tokens are used to determine the variable pruning beam for each frame. In places where disagreement is detected, the beam is increased to include all tracked tokens. Otherwise, in the second pass, the same narrow beam is used that was used in the first pass.

Even if we don’t increase the beam in the second pass, our method is doing more than simply choosing the best path from the two passes because it is possible to “recombine” partial paths from the first-pass and second-pass search (effectively combining the forward and backward lattices). On top of that, the variable beam leads to the generation of extra partial hypotheses in areas where both passes disagree, which gives an additional speed-up.

Tracked decoding leads to a 2-3 times speed-up compared to a single pass forward decoding. Since most of the time is spent in the forward and backward decoding with the narrow beam, this beam determines the possible speed-up. It should be small enough to decode at least two times faster than the original single pass, and it should be wide enough to allow for a reasonable comparison of the forward and backward search results, i.e. either of the two passes should obtain a solution, that is at least partly correct. If we decrease the beam below a critical threshold the speed-up vanishes, since an excessive amount of extra tokens are generated. Thus, we introduce an upper limit to the variable beam, which becomes effective in the areas of higher word error rates. We show that the main tuning parameters, which are the log beam width and the maximum number of active tokens for the histogram pruning, are dependent on each other.

Reversal of the recognition network

To construct the backward recognition network, it is not sufficient to apply WFST reversal to the forward network, since this will result in highly non-deterministic structures. It is necessary to construct reverse models for each component separately and to compose the components in the same way as in the forward network. It turned out that the transducers for HMM structure, context-dependency and pronunciation lexicon are rather easy to reverse, however, the reversal of the LM transducer is difficult. The stochasticity of outgoing arcs will not be satisfied when reversing the model, i.e. the optimal weight distribution for backward search is different from the one used in forward search. Therefore, we have to apply weight pushing to the reversed components. Our approach to the construction of backward recognition networks is not limited to static network decoders. Since all components are reversed individually, no change is necessary when dynamically composing the components in a dynamic network decoder.

To represent N-gram LMs as WFSTs, an approximate structure is necessary, since a fully connected model is prohibitive. When representing back-off arcs as either failure arcs or epsilon arcs, we actually violate the assumptions of the WFST algorithms. Either, when using failure arcs, the semi-ring concept is changed and a new class of algorithms is needed. On the other hand, when approximating back-offs using epsilon arcs, non-determinism is introduced. If the weights are taken from back-off LMs, the weight of cycles can be greater than one and results in an infinite total weight. A general weight pushing algorithm is based on the shortest path algorithm in the given semi-ring. The (log) probability semi-ring is not closed (due to cycles), therefore an approximate iterative weight pushing algorithm is used as the standard weight pushing (e.g. in OpenFST), whose convergence depends on the weight in a loop, which must be smaller than one. However, this is not the case for WFSA resulting from back-off LMs and the weight pushing algorithm will not converge.

We presented an alternative weight pushing algorithm, which will always converge. Similar to the power method for finding the dominant eigenvector of a matrix, we use the Perron theorem to obtain the dominant right eigenvector of the transition matrix of an ergodic WFST. This vector represents the minimum distance towards the final states (stationary state distribution), which we can use as the potential function in re-weighting. This results in pushing the weights towards the initial state and making the WFSA output stochastic. More precisely, the outgoing arcs sum to the same quantity for all states, which means that the total weight, causing the standard algorithm to fail, is now uniformly "smeared" all over the WFSA. Our algorithm is in practice an order of magnitude faster than the more generic conventional weight pushing algorithm.

The most difficult component to reverse is the WFST resulting from the back-off LM. We require that it assigns exactly the same probabilities as the forward LM. To guarantee an optimal search, the backward WFST should also be deterministic, stochastic and of minimal size. Thus, simple WFST reversal is not sufficient. We derive the construction of the backward LM satisfying these requirements, which is valid when using exact back-off models using failure arcs, and also when approximating them with epsilon arcs.

The constructive approach to obtain the backward LM consists of applying the N-gram probabilities with a delay, and to switch the functions of labeled word arcs and back-off arcs. We also explain the origin of missing N-grams, and how to represent them correctly in the backward LM. With the help of a series of weight pushing operations and representation changes of the probabilities, where each step guarantees WFSA equivalence, we show that our LM reversal algorithm can also be derived step by step. By applying the constraint that the joint word probabilities should be the same for the forward and backward LM for all N-gram orders, we are able to show that the same algorithm can be derived from Bayes' rule. The application of weight pushing to the resulting backward LM is crucial for optimal performance. We compared this 'exact' LM with a backward LM resulting from training on the reversed training texts. The performance of both is very similar, except for low word error rates, where the exact model performs better – more closely to the forward LM.

6.2 Future work

The proposed speed-up method can be applied in any ASR based technology, as e.g. in the fast generation of lattices for audio indexing and the tracked decoding could be used to generate lattices that contain desired paths, such as the forced-alignment reference for the discriminative training of acoustic models. Additionally to decoding forwards and backwards in time, depending on the task, there might be other ways of decoding, which could result in independent search errors, and thus lead to additional speed-ups.

The alternative weight pushing algorithm was derived under certain assumptions. In particular, we assume that all arcs in the WFST are of the same type. However, there are "emitting" arcs with a word label, and "non-emitting" arcs representing e.g. the back-off arcs. An open problem is to derive a weight pushing algorithm respecting the special semantics of back-off arcs. Under this correct interpretation, if the back-off LM was correctly normalized, the total weight of the transducer will be one, and we avoid the negative log-probabilities resulting from pushing weights greater than one. The original Kaldi recipe for the construction of recognition networks [Povey et al.(2011)] used the assumption, that all components are stochastic, which eliminates the necessity for weight pushing. We want to find a derivation for the exact LM reversal, which directly produces a properly normalized stochastic WFST.

There is some inconsistency between the algorithms for decoding graph construction, which usually assume the log-semi-ring, and the decoding algorithms, which use the tropical semi-ring. Together with different interpretations of the failure/epsilon arcs, this opens several dimensions of design choices, and the different options should be systematically explored to find a consistent framework for decoding graph construction that results in an optimal decoding. When using epsilon arcs for back-offs, the WFSTs resulting from back-off LMs introduce non-determinism to the graph, which results in multiple evaluations of the same models during decoding. It is not possible to apply determinization on the LM transducer, since this would lead to a fully connected N-gram, which is not feasible. However, in a preliminary experiment, we showed, that after the composition with the lexicon transducer, it is possible to apply another slightly modified determinization step, which respects the special semantics of failure arcs. The resulting transducer is bigger, but still managable. After this step, the transducer is deterministic, and no special arcs are needed (e.g. failure arcs) to correctly represent the back-off LM - i.e. the resulting transducer is consistent with the log-semi-ring. Therefore, the resulting WFST LG is either already stochastic, or can be normalized with the weight pushing in the log-semi-ring. Thus, many of the problems to which we point in this thesis could be solved.

Bibliography

[Abo-Gannemhy et al.(2010)] W. Abo-Gannemhy, I. Lapidot, and H. Guterman, “Speech recognition using combined forward and backward Viterbi search.” in *IEEE Convention of the Electrical and Electronic Engineers in Israel*, 2010.

[Agarwal et al.(2014)] A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, M. Padmilac, H. Parthasarathi, B. Peng, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, Y. Wang, H. Wang, K. Yao, D. Yu, Y. Zhang, and G. Zweig, “An introduction to computational networks and the computational network toolkit.” Tech. Rep. MSR-TR-2014-112, August 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=226641

[Aho and Corasick(1975)] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search.” *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[Allauzen et al.(2003)] C. Allauzen, M. Mohri, and B. Roark, “Generalized algorithms for constructing statistical language models.” in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ser. ACL ’03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 40–47.

[Allauzen et al.(2004)] C. Allauzen, M. Mohri, M. Riley, and B. Roark, “A generalized construction of integrated speech recognition transducers.” in *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, 2004. (ICASSP ’04)*, vol. 1, May 2004, pp. I–761–4 vol.1.

[Austin et al.(1991)] S. Austin, R. Schwartz, and P. Placeway, “The forward-backward search algorithm.” in *Proc. ICASSP*, 1991, pp. 697–700.

[Bellman(1952)] R. Bellman, “On the theory of dynamic programming.” *Proceedings of the National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.

[Berger et al.(1996)] A. L. Berger, V. J. D. Pietra, and S. A. D. Pietra, “A maximum entropy approach to natural language processing.” *Computational Linguistics*, vol. 22, number 1, pp. 39–71, 1996.

[Berman and Shaked-Monderer(2012)] A. Berman and N. Shaked-Monderer, “Non-negative matrices and digraphs.” in *Computational Complexity*, R. A. Meyers, Ed. Springer New York, 2012, pp. 2082–2095.

[Cardinal et al.(2013)] P. Cardinal, P. Dumouchel, and G. Boulianne, “Large vocabulary speech recognition on parallel architectures.” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 11, pp. 2290–2300, Nov 2013.

[Chong et al.(2009)] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, “A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit.” in *Interspeech 2009, 10th Annual Conference of the International Speech Communication Association*, September 2009.

[Cormen et al.(2009)] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition.* MIT press, 2009.

[Davidson et al.(2014)] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-efficient parallel GPU methods for single-source shortest paths.” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* IEEE, 2014, pp. 349–359.

[Dixon et al.(2009)] P. Dixon, T. Oonishi, and S. Furui, “Fast acoustic computations using graphics processors.” in *ICASSP 2009. IEEE International Conference on Acoustics, Speech and Signal Processing*, April 2009, pp. 4321–4324.

[Fiscus(1997)] J. G. Fiscus, “A post-processing system to yield reduced word error rates: Recognizer Output Voting Error Reduction (ROVER).” in *Proceedings 1997 IEEE Workshop on Automatic Speech Recognition and Understanding*, Dec 1997, pp. 347–354.

[Gibbons(1985)] A. Gibbons, *Algorithmic graph theory.* Cambridge University Press, 1985.

[Grinstead and Snell(1997)] C. M. Grinstead and J. L. Snell, *Introduction to Probability*, 2nd ed. American Mathematical Society, GNU General Public License, July 1997.

[Hannemann et al.(2013)] M. Hannemann, D. Povey, and G. Zweig, “Combining Forward and Backward Search in Decoding.” in *Proc. ICASSP 2013*, 2013, pp. 6739–6743. [Online]. Available: http://www.fit.vutbr.cz/research/view˙pub.php.en?id=10324

[Horowitz et al.(2005)] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, “Scaling, power, and the future of CMOS.” in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International.* IEEE, 2005, pp. 7–pp.

[Jouvet and Fohr(2013b)] D. Jouvet and D. Fohr, “Combining Forward-based and Backward-based Decoders for Improved Speech Recognition Performance.” in *Proc. Interspeech 2013 - 14th Annual Conference of the International Speech Communication Association*, 2013.

[Jouvet and Fohr(2014)] ——, “About Combining Forward and Backward-Based Decoders for Selecting Data for Unsupervised Training of Acoustic Models.” in *Proc. Interspeech 2014*, 2014, pp. 815–819.

[Jouvet and Fohr(2013a)] ——, “Analysis and Combination of Forward and Backward Based Decoders for Improved Speech Transcription.” in *Text, Speech, and Dialogue*, ser. Lecture Notes in Computer Science, I. Habernal and V. Matoušek, Eds.

Springer Berlin Heidelberg, 2013, vol. 8082, pp. 84–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40585-3˙12

[Katz(1987)] S. M. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer." in *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35(3), 1987, pp. 400–401.

[Kim et al.(2012)] J. Kim, J. Chong, and I. Lane, "Efficient On-The-Fly Hypothesis Rescoring in a Hybrid GPU/CPU-based Large Vocabulary Continuous Speech Recognition Engine." in *Proc. Interspeech*, 2012, pp. 1183–1186.

[Kneser and Ney(1995)] R. Kneser and H. Ney, "Improved backing-off for M-gram language modeling." in *Proc. ICASSP-95, International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, May 1995, pp. 181–184.

[Kuich and Salomaa(1986)] W. Kuich and A. Salomaa, "Semirings, Automata, Languages." in *EATCS Monographs on Theoretical Computer Science*, vol. No. 5. Springer-Verlag, Berlin, Germany, 1986.

[Lee and Kawahara(2009)] A. Lee and T. Kawahara, "Recent development of open-source speech recognition engine Julius." in *Proc. APSIPA Annual Summit and Conference*, 2009.

[Lee et al.(1998)] A. Lee, T. Kawahara, and S. Doshita, "An efficient two-pass search algorithm using word trellis index." in *Proc. ICSLP*, 1998.

[Lehmann(1977)] D. J. Lehmann, "Algebraic structures for transitive closure." *Theoretical Computer Science*, vol. 4, pp. 59–76, 1977.

[Li et al.(2009)] T. Li, W. Xu, J. Pan, and Y. Yan, "Improving automatic speech recognizer of voice search using system combination." in *Sixth International Conference on Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09*, vol. 4, Aug 2009, pp. 477–480.

[Lowerre(1976)] B. Lowerre, "The Harpy Speech Recognition System." Ph.D. dissertation, Carnegie Mellon University, 1976.

[Maleki et al.(2014)] S. Maleki, M. Musuvathi, and T. Mytkowicz, "Parallelizing Dynamic Programming Through Rank Convergence." in *Proc. ACM PPoPP'14*. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), February 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=208241

[Meyer and Sanders(2003)] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm." *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003, 1998 European Symposium on Algorithms.

[Mohri(2002)] M. Mohri, "Semiring frameworks and algorithms for shortest-distance problems." *Journal of Automata, Languages and Combinatorics*, vol. 7, pp. 321–350, March 2002.

[Mohri(1997)] ——, "Finite-state transducers in language and speech processing." *Computational linguistics*, vol. 23, no. 2, pp. 269–311, 1997.

[Mohri and Riley(2001)] M. Mohri and M. Riley, "A Weight Pushing Algorithm for Large Vocabulary Speech Recognition." in *Proc. Eurospeech 2001, 7th European Conference on Speech Communication and Technology*, 2001.

[Mohri et al.(2008)] M. Mohri, F. C. N. Pereira, and M. Riley, "Speech recognition with weighted finite-state transducers." in *Handbook on Speech Processing and Speech Communication, Part E: Speech recognition*, L. Rabiner and F. Juang, Eds. Heidelberg, Germany: Springer-Verlag, 2008, p. 31.

[Murveit et al.(1993)] H. Murveit, J. W. Butzberger, V. V. Digalakis, and M. Weintraub, "Large-vocabulary dictation using SRI's decipher speech recognition system: Progressive search techniques." in *Proc. ICASSP Vol. 2*, 1993, pp. 319–322.

[Nguyen et al.(1993)] L. Nguyen, R. Schwartz, F. Kubala, and P. Placeway, "Search algorithms for software-only real-time recognition with very large vocabularies." in *Proceedings of the Workshop on Human Language Technology*, 1993, pp. 91–95.

[Nolden et al.(2011)] D. Nolden, R. Schlüter, and H. Ney, "Acoustic look-ahead for more efficient decoding in LVCSR." in *Proc. Interspeech*, 2011.

[Nolden et al.(2012)] ——, "Extended search space pruning in LVCSR." in *Proc. ICASSP*. IEEE, 2012.

[Nolden et al.(2013)] ——, "Efficient nearly error-less LVCSR decoding based on incremental forward and backward passes." in *Proceedings ASRU 2013, IEEE Workshop on Automatic Speech Recognition and Understanding*, 2013, pp. 1–6.

[Ortmanns et al.(1996)] S. Ortmanns, H. Ney, and A. Eiden, "Language-model look-ahead for large vocabulary speech recognition." in *Proc. ICSLP 96, Fourth International Conference on Spoken Language Processing*, vol. 4, Oct 1996, pp. 2095–2098.

[Parihar and Hansen(2008)] N. Parihar and E. Hansen, "A lexical-tree division-based approach to parallelizing a cross-word speech decoder for multi-core processors." in *EUSIPCO 2008, 16th European Signal Processing Conference*, Aug 2008, pp. 1–5.

[Paul and Baker(1992)] D. B. Paul and J. M. Baker, "The Design for the Wall Street Journal-based CSR Corpus." in *DARPA Speech and Language Workshop*. Morgan Kaufmann Publishers, 1992.

[Phillips and Rogers(1999)] S. Phillips and A. Rogers, "Parallel speech recognition." *International Journal of Parallel Programming*, vol. 27, no. 4, pp. 257–288, 1999.

[Povey et al.(2011)] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The Kaldi speech recognition toolkit." in *Proc. ASRU*. IEEE, 2011.

[Povey et al.(2012)] D. Povey, M. Hannemann, G. Boulianne, L. Burget, A. Ghoshal, M. Janda, M. Karafiat, S. Kombrink, P. Motlicek, Y. Quian, N. Thang Vu, K. Riedhammer, and K. Vesely, "Generating exact lattices in the WFST framework." in *Proc. ICASSP*. IEEE, 2012, pp. 4213–4216.

[Soltau and Saon(2009)] H. Soltau and G. Saon, “Dynamic network decoding revisited.” in *Proc. ASRU 2009, IEEE Workshop on Automatic Speech Recognition Understanding*, Nov 2009, pp. 276–281.

[Steinbiss et al.(1994)] V. Steinbiss, B.-H. Tran, and H. Ney, “Improvements in beam search.” in *Proc. ICSLP*, vol. 94, no. 4, 1994, pp. 2143–2146.

[Stolcke(1998)] A. Stolcke, “Entropy-based pruning of backoff language models.” in *Proceedings DARPA Broadcast News Transcription and Understanding Workshop.* Morgan Kaufmann, February 1998, pp. 270–274.

[Tang and Cristo(2008)] M. Tang and P. D. Cristo, “Backward viterbi beam search for utilizing dynamic task complexity information.” in *Proc. Interspeech*, 2008, pp. 2090–2093.

[van Hamme and van Aelten(1996)] H. van Hamme and F. van Aelten, “An adaptive-beam pruning technique for continuous speech recognition.” in *Proc. ICSLP 96, Fourth International Conference on Spoken Language Processing*, vol. 4, Oct 1996, pp. 2083–2086.

[Viterbi(1967)] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm.” *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, April 1967.

[You et al.(2009)] K. You, J. Chong, Y. Yi, E. Gonina, C. J. Hughes, Y.-K. Chen, W. Sung, and K. Keutzer, “Parallel scalability in speech recognition.” *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 124–135, November 2009.

[Young et al.(1989)] S. J. Young, N. Russell, and J. Thornton, *Token passing: a simple conceptual model for connected speech recognition systems.* Cambridge University Engineering Department Cambridge, UK, 1989.

[Young et al.(2006)] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. A. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland, “The HTK Book. Revised for HTK Version 3.4.” 2006.

Appendix A

Scripts and executables in the Kaldi toolkit

Most of the algorithms and recipes described in this thesis have been integrated into the Kaldi toolkit. The master script invoking the scripts for single pass backward and two-pass tracked decoding can be found in `egs/wsj/s5/local/run_fwdbwd.sh`.

However, since the `utils/` directory is linked to all experiment directories `egs/`, the described scripts can be accessed from all recipes. During the preparation of the training/decoding directories, the first step is to reverse the lexicon. This is done with providing the `--reverse` option to `utils/prepare_lang.sh`. For the preparation of the decoding directory, we use the script `utils/reverse_lm.sh`, which creates a new `lang_test/` directory with the reversed LM transducer. It is very similar to the normal `utils/prepare_lang_test.sh`, i.e. creating the LM WFST with `src/bin/arpa2fst`, however, the heart of it is a call to `utils/reverse_arpa.py`, which takes as input a textual LM in ARPA format and outputs the exactly reversed LM in ARPA format. In this python script, we first read the ARPA file, add missing N-grams (section 4.3) and in a second pass we create the backward LM.

At the end of `utils/reverse_lm.sh`, we apply the alternative weight pushing algorithm to make the WFST stochastic. One particularity is that `arpa2fst` doesn't support the representation of back-off arcs of missing N-grams (section 4.3) in the backward LM. Therefore, we have to manually remove these arcs. To make a sanity check that everything went right, we can use the script `utils/reverse_lm_test.sh`, which generates random word sequences from the forward LM, reverses them and checks, that they are assigned the same scores in the forward and backward LM (including different ways of backing-off).

The last step towards the creation of a backward recognition network is to compose the $HCLG$ transducer from the individual components with the script `utils/mkgraph.sh`, which also has an option `--reverse`. After the lexicon transducers and LM transducers are already reversed, the only thing left to do is the reversal of the HMM transducer – the `--reverse` option is passed further to the executable `src/bin/make-h-transducer`. The relevant source code is actually in `src/hmm/hmm-utils.cc`. Here, the context window into the decision tree is reversed (section 5.2.1), and the individual (context-phone) HMMs are reversed and pushed, before composing them as H_a transducer.

The forward/backward decoding is done with the script `steps/decode_fwdbwd.sh`. In case of a simple backward decoding (using `src/gmmbin/gmm-latgen-faster`) we use the `--reverse` option, and the only two things that need to be changed compared to a forward decoding is the time reversal of the acoustic features with the executable `src/featbin/reverse-feats` (where the code is actually in `src/feat/feature-functions.cc`), and the reversal of the decoded text in the scoring script `steps/score_kaldi.sh` (called by `local/score.sh`).

To use the tracked decoding, we run the first pass as just described (`--beam` is the baseline beam width and `--latbeam` is the lattice pruning beam), and then we decode in the opposite direction, using the `--first_pass` option as an additional input, followed by the first pass decoding directory, from which we take the lattices.

The script `steps/decode_fwdbwd.sh` has two additional options `--extra_beam` and `--max_beam`, which set the extra beam and the beam limit (section 5.4.2), respectively. If the `--first_pass` option is given, we convert the first pass lattice to the graph-arc acceptor lattice (section 5.4.3) and use the executable `src/gmmbin/gmm-latgen-tracking` to perform the tracked decoding. All the necessary source code for the arc-lattice generation is in `src/latbin/lattice-arcgraph.cc`, which compiles also to the corresponding executable.

In `src/decoder/lattice-tracking-decoder.{cc,h}` is the implementation of the tracking decoder. The arc-lattice is read as standard WFSA. The central method `LatticeTrackingDecoder::Decode()` gets it as an input parameter. Each token (represented as a `struct Token`) has an additional component, the state in the arc-lattice. The main methods inside `Decode()`, that realize the decoding and tracking are `ProcessEmitting()` and `ProcessNonemitting()`, which are called alternately. The beam width policy (section 5.4.2) is implemented in the method `GetCutOff()`, where the variable `extra_cutoff` is computed, as well as in `ProcessNonemitting()`.