Annotating large lattices with the exact word error

Rogier C. van Dalen, Mark J. F. Gales

Department of Engineering, University of Cambridge, United Kingdom

rcv25@cam.ac.uk, mjfg@eng.cam.ac.uk

Abstract

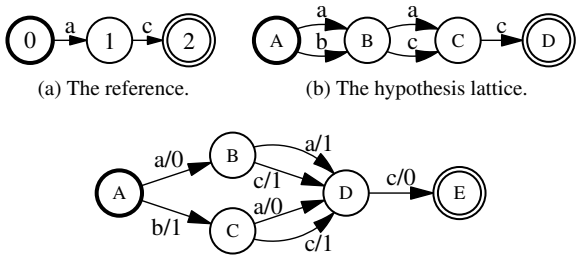
The acoustic model in modern speech recognisers is trained discriminatively, for example with the minimum Bayes risk. This criterion is hard to compute exactly, so that it is normally approximated by a criterion that uses fixed alignments of lattice arcs. This approximation becomes particularly problematic with new types of acoustic models that require flexible alignments. It would be best to annotate lattices with the risk measure of interest, the exact word error. However, the algorithm for this uses finite-state automaton determinisation, which has exponential complexity and runs out of memory for large lattices. This paper introduces a novel method for determinising and minimising finite-state automata incrementally. Since it uses less memory, it can be applied to larger lattices.

Index Terms: speech recognition, discriminative training, minimum Bayes risk

1. Introduction

Many operations in speech recognition can be elegantly described in terms of finite-state automata [1, 2, 3, 4]. However, optimisation algorithms do not always create the desired results. This paper focuses on determinisation, which has exponential space complexity. Even when the output automaton fits in memory, the intermediate representation may not. This paper therefore proposes an algorithm to incrementally determinise and minimise an acyclic automaton. It uses less memory by keeping intermediate automata minimised at all times.

The use case that this paper will consider is that of annotating lattices with the exact word (or phone) error. This problem comes up in training acoustic models. A commonly-used criterion is the minimum Bayes risk criterion [5] (MBR), which aims to minimise the expected word (or phone, or state) error. This involves a marginalisation over all word sequences, approximated with a lattice, of the weighted error for the word sequence. However, the algorithm for computing the word error for all word sequences in a lattice [6, 7] uses determinisation and in practice often runs out of memory. An approximation is therefore used that uses a fixed alignment of the words in the reference and the hypothesis lattice [5]. This may limit the applicability of the criterion used to the criterion purported to minimise. Also, as new models for speech recognition are becoming more complex and features richer, the fixed time alignments could restrict performance. For example, recent work has focussed on the need to re-align lattices every few iterations of training for HMMs with neural networks [8] and structured SVMs [9] or, for log-linear models, to use dense lattices to represent many alignments [10, 4]. This paper will therefore use the novel determinisation algorithm to compute the exact word or phone error for all the paths in a lattice.



(a) The reference. (b) The hypothesis lattice.

(c) The hypothesis lattice from figure 1b, annotated with the error. This changes the topology.

Figure 1: *Example reference and hypothesis automata.*

The word error between two sequences is defined as the minimum edit distance between the two. The *word error rate* used to assess speech recognisers is the word error divided by the length of the reference. The well-known algorithm for computing the minimum edit distance finds for the lowest-cost path in a two-dimensional state space. It can be expressed in terms of finite-state automata, as a shortest-distance calculation in a graph whose states are in the Cartesian product of the automata representing the two sequences. This algorithm can be generalised in two ways. One way is to find the combination of paths in the two automata that minimises the edit distance [6].

This paper will focus on the second way to generalise the minimum edit distance algorithm: finding the error not for one sequence, but for a whole lattice. Instead of a shortest-distance algorithm, this requires determinisation. An example is given in figure 1: the reference sequence is "a c" and the hypothesis lattice has four sequences. The word error for three of those sequence is 1 ("a a c", "a c c", "b a c"), and for the other sequence it is 2 ("b c c"). The automaton in figure 1c assigns the error to those four sequences. In this example, the topology of the automaton must be changed, so that each of the symbol sequences has a different path. This illustrates that in general, determinisation can lead to an exponential number of transitions. The standard determinisation algorithm shows exponential behaviour on normal lattices used for training speech recognisers.

This paper is organised as follows. Section 2 will discuss the minimum edit distance problem in terms of finite-state automata. Section 3 will introduce a semiring whose members are acyclic automata that are always determinised and minimised, and thus take as little memory as possible. By using them as the weights of another automaton, a general algorithm will then be introduced for determinising and minimising acyclic automata.

2. The minimum edit distance

The edit distance measures the similarity between two sequences of symbols as the number of operations required to

This work was supported by EPSRC Project EP/I006583/1 (Generative Kernels and Score Spaces for Classification of Speech) within the Global Uncertainties Programme and by a Google Research Award.

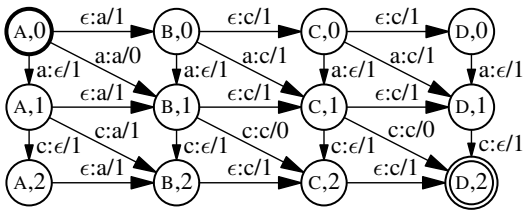


Figure 2: *The edit distance automaton for "a c" to "a c c". The shortest distance in this automaton is the edit distance.*

transform one string into another. The Levenshtein distance, used to evaluate speech recognisers, counts deletions, insertions, and substitutions. The optimal sequence of operations is found as the best path through both sequences at once. There is a standard dynamic programming algorithm for solving this (see e.g. [11]). To generalise it, the problem and algorithm will here be expressed in terms of finite-state automata [6].

Figure 2 shows an example automaton for determining the edit distance between "a c" (along the vertical axis) and "a c c" (along the horizontal axis). The states are in a product space of the states in figures 1a and 1b. Each transition stands for an edit operation. A vertical transition moves in the reference but not in the hypothesis: a deletion. The label is e.g. "a:$\epsilon/1$", with "a" in the reference, no symbol ("ϵ") in the hypothesis, and a cost of 1 for the deletion. The opposite, an insertion, is represented by a horizontal transition, with e.g. "ϵ:a/1". Diagonal transitions indicate a substitution (e.g. "c:a/1"), with a cost of 1 or a correct symbol (e.g. "a:a/0"), with a cost of 0. This automaton can be produced with a 3-way composition of the two input automata and a special transducer [6, 7], or with an ad hoc algorithm.

The lowest-cost path in this automaton corresponds to the minimum edit distance [6]. Weights on finite-state automata, here the cost of edit operations, must be in a *semiring* [12], which defines operations $\oplus$ and $\otimes$. $\oplus$ combines weights of competing paths, so the lowest cost should be selected: $x \oplus y \triangleq \min(x, y)$. $\oplus$ combines weights of consecutive paths, so costs should add up: $x \otimes y \triangleq x + y$. Using this semiring, the cost (sometimes "tropical") semiring, a shortest-distance algorithm finds the minimum edit distance between two sequences.

One generalisation of this algorithm, which this paper will not focus on, is to make the hypothesis a lattice (as in figure 1b), but run the same shortest-distance algorithm. This results in the minimum edit distance between the reference and the one best path in a hypothesis lattice: the oracle error.

3. Incremental determinisation and minimisation

To find the minimum edit distance for all paths in a lattice, the edit distance automaton in figure 2 must be determinised [6, 7]. The standard algorithm for determinisation of weighted finite-state automaton [13] uses a powerset construction: each state in the resulting automaton is a set of states in the original automaton with a weight for each of them. Each transition represents all transitions with the same symbol out of each of these states. The destination state is instantiated as the set of the original destination states and weights. The weights are normalised so that states are more likely to be re-used: if the state representing the exact same set of states and weights has been seen before, it is mapped to the same state in the resulting automaton.

The problem with applying this general algorithm for the edit distance problem is memory use. Because all states in the original automaton are inserted into at least one state in the resulting automaton, the amount of memory required is at least the size of the automaton in figure 2. In theory, it would be possible to use the knowledge that the resulting automaton is acyclic to reduce this, but in general this is a hard problem, requiring garbage collection that can deal with cycles as well as problem-specific reference counting to determine whether a state in the resulting automaton may yet be revisited.

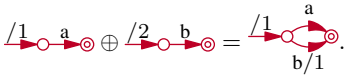
An option that may seem attractive is to prune the edit distance automaton. It is true that many paths turn out to be redundant. However, a pruning algorithm that removes paths with high cost will prune not only redundant paths, but also legitimate paths in the lattice where the recogniser was badly wrong. That is not a good idea for discriminative training. Therefore, this paper aims to annotate lattices with the exact error.

Instead of optimising standard determinisation to acyclic automata, this paper uses a simpler approach: the weights are replaced by equivalent weights in a different semiring. This semiring is the space of determinised and minimised acyclic automata. A standard shortest-distance algorithm then recreates the original automaton in determinised and minimised form.

3.1. The automaton semiring

Minimisation of weighted automata [14, 15, 16] works by normalising the weights (also known as "weight pushing"), and then merging states with the same *suffix*, the same labels and weights following. Assuming that there are no arcs with empty symbol sequences, there is only one possible topology for one suffix, and the state with this suffix can be shared. An important insight since many values in the automaton semiring will be in memory at once, they can all share the same states, so they are jointly minimised. Each automaton is also kept determinised. This means that each state has at most one outgoing transition with a given symbol, and therefore that for each symbol sequence there is only one path. Since deterministic automata have only one start state, and most automata will have a start weight, it will be indicated on an unconnected arrow:  is an automaton that assigns 1 to the symbol sequence "a" (as usual, a weight $/0$ is not written). The rest of this paper will use the cost semiring and operations specific to this. However, this is trivial to generalise to any weakly left divisible semiring [12].

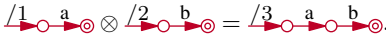
Like the cost semiring, the automaton semiring must have operations $\otimes$ and $\oplus$, and identities $\overline{1}$ and $\overline{0}$ defined. Since the sum over all paths of the automaton semiring should recreate its host automaton, the operations are defined as follows. The operation $\oplus$ is used to combine the weights of two competing paths. It is therefore defined as the union of its arguments, which assigns any sequence the minimum value (the $\oplus$-sum) of what its arguments assign to that sequence. For example,



(1a)

The left argument assigns 1 to "a", and the right one 2 to "b". The result assigns those weights to both. To keep the automaton normalised, its weights have been pushed to the front.

The operation $\otimes$, which combines consecutive arcs along a path, concatenates two automata. This means that the concatenation of each string of the left-hand automaton with each string of the right-hand automaton is assigned the sum (the $\otimes$-product) of the weights of the two automata. For example,

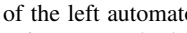


(1b)

The resulting automaton assigns a weight of 3 to "a b". To keep the automaton normalised, the weights are pushed to the front.

The values $\overline{0}$ and $\overline{1}$ must be defined so that adding $\overline{0}$ does nothing, and multiplying by $\overline{1}$ does nothing. They are therefore defined as $\overline{1} \triangleq$  $=$  and $\overline{0} \triangleq$  .

3.1.1. *Implementation*

The data structures used to implement the automata and the operations on them are best expressed mathematically. An automaton $a = (s, q)$ consists of an initial weight s, which can be extracted with $s(a)$, and a state q, which can be extracted with $q(a)$. A state $q = (f, \mathcal{T})$ is defined by a final weight f and a set of outgoing transitions $\mathcal{T}$. Each transition $t = (k, a)$ consists of a symbol k, which can be extracted with $k(t)$, and an automaton a attached to it, which can be extracted with $a(t)$. For example, the automaton  $=$ (weights of 0 are normally not shown explicitly) that only assigns 0 to empty sequences has start and final weight 0 and can be written $a_f = (0, (0, \emptyset))$. The automaton  can be expressed as $(2, (\infty, \{(\mathrm{b}, a_f)\}))$. ∞ is the final weight of the first state.

The following will discuss the operations $\textsc{Union}(a_l, a_r)$, for implementing $\oplus$, and $\textsc{Concatenate}(a_l, a_r)$, for $\otimes$, but first building blocks $\textsc{Normalise}(a)$ and $\textsc{Denormalise}(a)$.

Since the states should be re-used as much as possible, they should be stored normalised. The standard normalisation used in minimisation algorithms is to push weights from all paths forward [15, 16]. The function $\textsc{Normalise}(a)$ ensures that the outer level of weights from state $q(a)$ is normalised. In the cost semiring, the smallest of the final weight f and the weights s' of the automata following the state is made 0 by subtracting the residual c from all weights, and adding it to start weight s. For readability, the following definition expands the argument:

$$\textsc{Normalise}((s, (f, \mathcal{T}))) \triangleq \big(s + c, (f - c, \mathcal{T}_{-c})\big), \quad \text{(2a)}$$

$$\text{where } \mathcal{T}_{-c} \triangleq \{(k, (s' - c, q)) \mid (k, (s', q)) \in \mathcal{T}\};$$

$$c \triangleq \min(\{f\} \cup \{s' \mid (k, (s', q)) \in \mathcal{T}\});$$

$$\textsc{Normalise}(\overline{0}) \triangleq \overline{0}. \quad \text{(2b)}$$

The resulting automaton assigns the same weights to the same sequences as the original automaton does.

The operation $\textsc{Denormalise}(a)$ performs the opposite operation. By adding s to f and all s', it produces an automaton $a' = (0, q')$ with the start weight 0, that is equivalent to a.

Since the automata are acyclic, the operations Union and Concatenate can be implemented recursively. $\textsc{Union}((s_l, q_l), (s_r, q_r))$ returns an automaton that assigns to each symbol sequence the minimum of the weights that the two arguments assign. Without loss of generality, the initial weights of the automata can be assumed 0, and the states unnormalised. Otherwise, Denormalise can be used. Then,

$$\textsc{Union}((0, (f_l, \mathcal{T}_l)), (0, (f_r, \mathcal{T}_r))) = \textsc{Normalise}((0, (\min(f_l, f_r), \mathcal{T}'))), \quad \text{(3a)}$$

where the new transition set $\mathcal{T}'$ contains all the symbols k from the left- and right-hand arguments, with merged automata a_k. It is defined $\mathcal{T}' \triangleq \{(k, a_k) \mid k \in \mathcal{T}_l \cup \mathcal{T}_r\}$, where

$$a_k \triangleq \begin{cases} \textsc{Union}(a(t), a(u)) & \text{if } \exists t \in \mathcal{T}_l, \exists u \in \mathcal{T}_r \\ & \text{s.t. } k(t) = k(u) = k; \\ a(t), & \text{if } \exists t \in \mathcal{T}_l \text{ s.t. } k(t) = k; \\ a(u), & \text{if } \exists u \in \mathcal{T}_r \text{ s.t. } k(u) = k. \end{cases} \quad \text{(3b)}$$

Figure 3: *The edit distance automaton from figure 2 with labels in the automaton semiring.*

The $\textsc{Concatenate}(a_l, a_r)$ operation returns an automaton that assigns to the concatenation of each string of the left-hand automaton with each string of the right-hand automaton the sum of the weights of the two automata, like in (1b).

The strategy for finding this automaton is to treat two parts of the left automaton separately: the final weight and the outgoing arcs. The left automaton assigns empty sequences cost $s_l + f_l$ (which may be ∞), which is added to the cost that the right automaton assigns to sequences. The outgoing arcs of the left automaton are traversed recursively. The results of the two parts are then combined by calling Union. Expanding the arguments, $\textsc{Concatenate}(a_l, a_r)$ can be written as

$$\textsc{Concatenate}((s_l, (f_l, \mathcal{T}_l)), (s_r, q_r)) = \textsc{Normalise}\big(\textsc{Union}\big((s_l + f_l + s_r, q_r), (s_l, (0, \mathcal{T}'))\big)\big), \quad \text{(4)}$$

where $\mathcal{T}' \triangleq \{(k, \textsc{Concatenate}(a, a_r)) \mid (k, a) \in \mathcal{T}_l\}$.

The expressions in (3) and (4) are recursive, and in an eagerly-evaluated language (like C++) they will take time in the order of the number of paths of the input automata. However, the fact that these functions do not have any side effects can be exploited by applying *memoisation*, which stores the result of a function the first time it is called, and afterwards returning the stored value. It is useful to store the arguments normalised, by subtracting the smaller of the start weights. Using memoisation lowers the time complexity of Union and Concatenate to the order of the number of states in the automata.

Another opportunity for optimisation is when either the left or the right argument is returned exactly. By memoising weight thresholds for when this happens, Union can be made to run faster, but using no less memory.

3.2. Determinisation and minimisation

The complete algorithm for determinisation and minimisation works as follows. An automaton is constructed with values in the automaton semiring as its labels, representing the symbols and weights of interest. Then, a shortest-distance algorithm is applied. Since the semiring has been described as modelling the suffixes of states, not the prefixes, the shortest-distance algorithm should work backwards from the final state.

Figure 3 illustrates the normal weighted automaton in figure 2 can be converted into one with weights in the automaton semiring. Since the interest here is in the symbols in the hypothesis lattice, the "output" symbols are used, with their weights. For example, a label "a:c/1" is converted into  , which assigns a weight only to the empty sequence, is produced for a label without a symbol, like "a:ϵ/1".

The shortest-distance algorithm in progress is illustrated in figure 4. For simplicity, the example has only one path in the

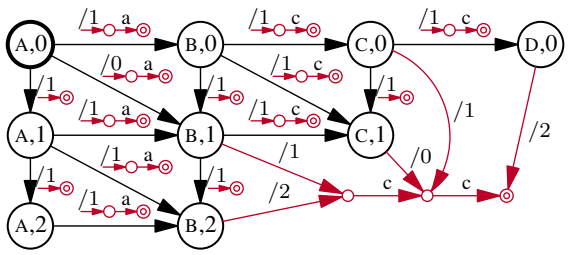


Figure 4: *A shortest-distance algorithm on the automaton in figure 3. The resulting determinised and minimised automaton is forming as the accumulated weights of the algorithm.*

hypothesis lattice, so the resulting automaton also has one path. The states from the automaton whose transitions have been processed have been removed from the graph. For the states on the frontier, the automaton-valued shortest distances to the final state have been computed. They are drawn as red automata starting from the states on the frontier. For example, from state "B,1" the automaton semiring has cost 1 for "c c". Once the algorithm has completed, the shortest distance to "A,0" will assign a cost of 1 to "a c c", which is the correct result.

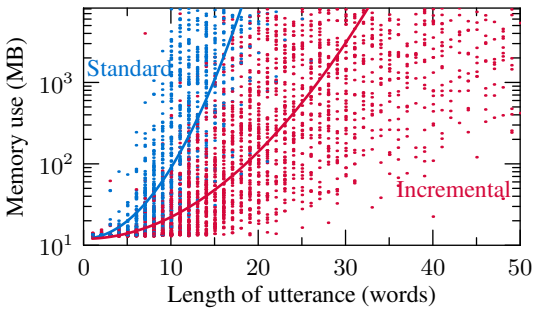
In general, this algorithm finds a determinised and minimal equivalent automaton for any acyclic automaton. The advantage compared to the standard determinisation algorithm is that by computing the output incrementally, peak memory use is lower.

4. Results

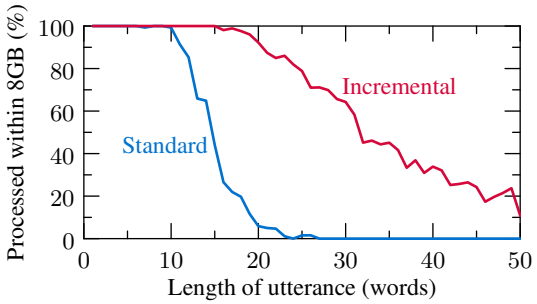
To test the new algorithm for error-marking lattices, a realistic training set with long utterances was chosen. The training set was 34 hours of randomly selected shows from the '96 release of the Hub4 broadcast news, LDC97S44 [17]. The hypothesis lattices (sometimes called "denominator lattices" in a reference to the CML training criterion) are produced for minimum phone error training in the standard setup at the Department of Engineering at Cambridge, which is similar to [18].

Two algorithms are compared, both exact. Both find the phone error, which is usually used for training speech recognisers, for all paths in the hypothesis lattice. The cost metric uses monophone identities, which is realistic but in preliminary experiments increased time and space required compared to using the triphones the lattices contain. The standard algorithm is implemented in C++ using the OpenFst toolkit [19]. It uses lazy composition to create the automaton in figure 2, but uses standard determinisation, which instantiates all states anyway. The implementation of the incremental algorithm uses the author's *Flipsta* C++ library for finite-state automata [20], open-sourced under the permissive Apache License [21], which allows more lazy operations. The algorithm lazily constructs an automaton like in figure 2, and transforms it on the fly to one like in figure 3. It provides a single-source shortest-distance algorithm that exploits the acyclicity by releasing memory for computed distances as soon as they are not needed any more.

Both algorithms were run constrained to 8 gigabytes of memory, and fail if they try to use more. A separate process was run for each utterance, measuring memory use with GNU *time*. Figure 5a shows peak memory use for the two algorithms; each dot represents an utterance (drawn alternating between the two algorithms). The horizontal axis has the number of words in the utterance. The number of states in the edit distance automaton increases roughly quadratically with this, but the output roughly



(a) Memory use of error-marking lattices, up to 8 GB.



(b) Percentage of lattices that can be error-marked within 8 GB of memory.

Figure 5: *Standard and incremental error-marking algorithms.*

linearly. The curves in the graph indicate the result of linear regression (in the log-space the diagram is drawn in) on the exponent of the square of the utterance length, fitted on lengths such that 99 % of the processes completed. It is clear that the standard algorithm uses much more space than the novel incremental algorithm. Since the vertical axis is logarithmic, at 32 words, where the incremental algorithm needs around 8 gigabytes, the curve for the standard algorithm suggests it would require an infeasible 8.5 petabytes of RAM.

Figure 5b illustrates the range of practical use of the two algorithms, by displaying the percentage of utterances that can be error-marked within 8 GB (the story would be similar for any other amount of memory available in real computers). The standard algorithm error-marks 99 % of utterances of up to 11 words and then quickly becomes unable to finish the computation. The incremental algorithm, however, can deal with much longer sentences: it error-marks 99 % of utterances of up to 21 words.

This improvement makes it much more feasible to error-mark lattices. For some corpora, a 20-word limit may be sufficient, or automatic segmentation can be used. It must be noted that here no pruning is used, and no admissible heuristic can be found. Pruning based on the error is hazardous because it may strip away legitimate high-error paths, which should be retained for training. It may, however, be possible to prune conservatively based on lattice arc timings.

5. Conclusion

This paper has introduced a novel method for determinising and minimising acyclic automata that uses less memory. It has been used to mark lattices with the exact word or phone error. This increases the length of the utterances that can be processed within memory twofold. This makes it much more feasible to use a training criterion based on the exact error in practice.

6. References

[1] 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.

[2] B. Hoffmeister, G. Heigold, R. Schlüter, and H. Ney, "WFST enabled solutions to ASR problems: Beyond HMM decoding," IEEE *Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 2, pp. 551–564, Feb 2012.

[3] D. Povey, M. Hannemann, G. Boulianne, L. Burget, A. Ghoshal, M. Janda, M. Karafiat, S. Kombrink, P. Motlicek, Y. Qian, K. Riedhammer, K. Vesely, and N. T. Vu, "Generating exact lattices in the WFST framework," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2012.

[4] R. C. van Dalen, A. Ragni, and M. J. F. Gales, "Efficient decoding with generative score-spaces using the expectation semiring," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2013.

[5] D. Povey, "Discriminative training for large vocabulary speech recognition," Ph.D. dissertation, Cambridge University, 2003.

[6] M. Mohri, "Edit-distance of weighted automata: General definitions and algorithms," *International Journal of Foundations of Computer Science*, vol. 14, no. 06, pp. 957–982, 2003.

[7] G. Heigold, W. Macherey, R. Schlüter, and H. Ney, "Minimum exact word error training," in *Proceedings of the Automatic Speech Recognition and Understanding Workshop*, Nov. 2005, pp. 186–190.

[8] H. Su, G. Li, D. Yu, and F. Seide, "Error back propagation for sequence training of context-dependent deep networks for conversational speech transcription," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2013.

[9] S.-X. Zhang and M. J. F. Gales, "Structured support vector machines for noise robust continuous speech recognition," in *Proceedings of Interspeech*, 2011.

[10] A. Ragni and M. J. F. Gales, "Inference algorithms for generative score-spaces," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2012, pp. 4149–4152.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[12] M. Mohri, "Semiring frameworks and algorithms for shortest-distance problems," *Journal of Automata, Languages and Combinatorics*, vol. 7, no. 3, pp. 321–350, 2002.

[13] ——, "Weighted automata algorithms," in *Handbook of Weighted Automata*, M. Droste, W. Kuich, and H. Vogler, Eds. Springer, 2009, pp. 213–254.

[14] D. Revuz, "Minimization of acyclic deterministic automata in linear time," *Theoretical Computer Science*, vol. 92, no. 1, pp. 181–189, 1992.

[15] M. Mohri, "Minimization algorithms for sequential transducers," *Theoretical Computer Science*, vol. 234, pp. 177–201, 2000.

[16] J. Eisner, "Simpler and more general minimization for weighted finite-state automata," in *Proceedings of the Joint Meeting of the Human Language Technology Conference and the North American Chapter of the Association for Computational Linguistics (*HLT-NAACL*)*, Edmonton, 2003, pp. 64–71.

[17] D. Graff, J. Garofolo, J. Fiscus, W. Fisher, and D. Pallett, "1996 English broadcast news speech (HUB4), LDC97S44," Philadelphia, 1997.

[18] M. J. F. Gales, D. Y. Kim, P. C. Woodland, H. Y. Chan, D. Mrva, R. Sinha, and S. E. Tranter, "Progress in the CU-HTK broadcast news transcription system," IEEE *Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 5, pp. 1513–1525, 2006.

[19] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "OpenFst: A general and efficient weighted finite-state transducer library," in *Proceedings of the International Conference on Implementation and Application of Automata, (CIAA 2007)*, ser. Lecture Notes in Computer Science, vol. 4783. Springer, 2007, pp. 11–23. [Online]. Available: http://openfst.org/

[20] R. C. van Dalen, "Flipsta finite-state automaton library," https://github.com/rogiervd/flipsta, 2015.

[21] The Apache Software Foundation, "Apache license, version 2.0," http://www.apache.org/licenses/LICENSE-2.0, 2004.