The hypothesis lattice.

A
b/1
a/0 a/1
\( b \)
\( a \)
C
\( b \)
\( a \)
B

Resent many alignments \([10, 4]\). This paper will therefore use HMM of training for has focussed on the need to re-align lattices every few iterations. This 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 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.

The edit distance measures the similarity between two sequences of symbols as the number of operations required to convert one into the other. The word error is defined as the number of insertions, deletions, or substitutions required to transform one sequence into another. This can be computed using dynamic programming algorithms such as the Needleman-Wunsch algorithm or the Wagner-Fischer algorithm.

This paper will consider is that of annotating lattices with the exact word 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 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.

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
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 (along the horizontal axis). The states are in a product space of edit distance between “a c” (along the vertical axis) and “a c c” (see e.g. [11]). To generalise it, the problem and algorithm will here be expressed in terms of finite-state automata [6].

The operation \( \oplus \) combines weights of competing paths, so the lowest cost should be selected: \( x \oplus y \equiv \min(x, y) \). \( \otimes \) combines weights of consecutive paths, so costs should add up: \( x \otimes y \equiv x + y \). Using this semiring, the cost (sometimes “tropical”) semiring, a shortest-distance algorithm finds the minimum edit distance between two sequences.

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 and minimised. The standard algorithm for determinisation of weighted finite-state automata [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: \( a \sim b \) 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].

The cost semiring, the automaton semiring must have operations \( \otimes \) and \( \oplus \), and identities \( \top \) and \( \bot \) defined. Since the sum over all paths of the automaton semiring should recreate its host automaton, the operations are defined as follows. The operation \( \otimes \) 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,

\[
\begin{align*}
\frac{1}{a} \oplus \frac{2}{b} & = \frac{1}{a} \bowtie \frac{2}{b} = \frac{1}{b/1} \\
\end{align*}
\]

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 \( \oplus \)-product) of the weights of the two automata. For example,

\[
\frac{1}{a} \otimes \frac{2}{b} = \frac{1/2}{a b}.
\]
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 $\mathcal{B}$ and $\mathcal{T}$ must be defined so that adding $\mathcal{B}$ does
nothing, and multiplying by $\mathcal{T}$ does nothing. They are therefore
defined as $\mathcal{T} \doteq \{0\}$ and $\mathcal{B} \doteq \{\infty\}$.

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 $q = (f,T)$ is defined by a final
weight $f$ and a set of outgoing transitions $T$. Each transition
t $t = (k,a)$ 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 $A_0$ (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
$a_f = \{0,\{0,0\}\}$. The automaton $A_1$ can be expressed as
$(2,\{\infty,\{(b,a)\}\})$. $\infty$ is the final weight of the first state.

The following will discuss the operations $\text{Union}(a_1,a_2)$,
for implementing $\oplus$, and $\text{Concatenate}(a_1,a_2)$, for $\otimes$, but
first building blocks $\text{Normalise}(a)$ and $\text{Denormalise}(a)$.

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 $\text{Normalise}(a)$ 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:

$$
\text{Normalise}((s,(f,T))) \doteq (s + c, (f - c, T_{-c})).
$$

where $T_{-c} \doteq \{(k, (s' - c, q)) | (k, (s', q)) \in T\}$;

e $\doteq \min((f) \cup \{s' | (k, (s', q)) \in T\});$

$$
\text{Normalise}(\emptyset) \doteq \emptyset.
$$

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

The operation $\text{Denormalise}(a)$ 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 $\text{Union}$
and $\text{Concatenate}$ can be implemented recursively.
$\text{Union}((s_1,q_1),(s_2,q_2))$ 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, $\text{Denormalise}$ can be used. Then,

$$
\text{Union}(0, (f_2,T_2), (0, (f_3, T_3))) = \text{Normalise}(0, (\min(f_1,f_2,f_3), T'))
$$

where the new transition set $T'$ contains all the symbols $k$ from
the left- and right-hand arguments, with merged automata $a_k$. It is
defined $T' \doteq \{(k,a_k) | k \in T_1 \cup T_2 \}$, where

$$
a_k \doteq \begin{cases}
\text{Union}(a(t),a(u)) & \text{if } 3t \in T_1 \& 3u \in T_2 \\
a(t) & \text{s.t. } k(t) = k(u) = k; \\
a(u) & \text{if } 3u \in T_2 \& k(t) = k;
\end{cases}
$$

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 $\text{Union}$ and $\text{Concatenate}$
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, $\text{Union}$ 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 hypoth-
thesis lattice, the “output” symbols are used, with their weights.
For example, a label “ace/1” is converted into \(1_c \oplus 1_a \oplus 1_e\),
which assigns a weight only to the empty sequence, is produced
for a label without a symbol, like “ace/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.

Figure 5: Standard and incremental error-marking algorithms.

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 increases roughly quadratically with this, but the output roughly 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


