INCREMENTAL LATTICE DETERMINIZATION FOR WFST DECODERS

Zhehuai Chen[1], *Mahsa Yarmohammadi*[2], *Hainan Xu*[2], *Hang Lv*[2,4]
Lei Xie[4], *Daniel Povey*[2,3], *Sanjeev Khudanpur*[2,3]

[1]SpeechLab, Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2]Center for Language and Speech Processing & [3]HLTCOE, Johns Hopkins University
[4]ASLP@NPU, School of Computer Science, Northwestern Polytechnical University

chenzhehuai@sjtu.edu.cn, {hanglv,lxie}@nwpu-aslp.org, {mahsa,hxu31,dpovey1,khudanpur}@jhu.edu

ABSTRACT

We introduce a lattice determinization algorithm that can operate incrementally. That is, a word-level lattice can be generated for a partial utterance and then, once we have processed more audio, we can obtain a word-level lattice for the extended utterance without redoing all the work of lattice determinization. This is relevant for ASR decoders such as those used in Kaldi, which first generate a state-level lattice and then convert it to a word-level lattice using a determinization algorithm in a special semiring. Our incremental determinization algorithm is useful when word-level lattices are needed prior to the end of the utterance, and also reduces the latency due to determinization at the end of the utterance.

Index Terms— ASR, WFST, Lattice, Determinization, Incremental, Latency.

1. INTRODUCTION

In Automatic Speech Recognition (ASR), lattices are representations of the most likely word-sequences that might correspond to a decoded utterance. Essentially a lattice is an acyclic graph with words on the edges, but in various settings they might come with additional information such as timing, phone-level or state-level alignment, acoustic and language model scores, and so on.

Exact lattice generation [1] is a method of generating word-level lattices by having the decoder first generate a state-level lattice, and then converting to a word-level lattice with a determinization algorithm using a special semiring that only preserves the best state alignment for each word-sequence. It is, in a certain sense, "exact" as it avoids approximations like the word-pair assumptions that are necessary in more traditional lattice generation methods [2].

The problem we are trying to solve is that if we are decoding a long utterance and want the word-level lattice to be available in real-time, standard determinization algorithms give us no way to re-use previously done work; we would have to redeterminize the whole state-level lattice each time we decode more of the utterance. Moreover, in real-time streaming scenarios we would incur a perceptible latency at the end of long utterances, by having to determinize the entire lattice. By doing it incrementally we can reduce this latency.

This work was partially supported by the IARPA MATERIAL program, NSF CRI Grant No 1513128, and an unrestricted gift from Google. The first author was supported by the National Key Research and Development Program of China under Grant No.2017YFB1002102, the China NSFC project (No. 61573241) and the Shanghai International Science and Technology Cooperation Fund (No. 16550720300).

The rest of the paper is organized as follows. Finite state machines and the notion of determinization are introduced in Section 2. The incremental determinization algorithm is proposed in Section 5 and applied to lattice processing in Section 6. Experiments conducted on the LibriSpeech corpus are described in Section 7, followed by conclusions in Section 8.

2. FINITE STATE ACCEPTORS AND TRANSDUCERS

Readers familiar with the literature on finite state acceptors (FSAs) and transducers (FSTs) may skip this section.

We give neither a precise nor a complete account of FSAs or FSTs or of determinization, but it will be helpful, for background, to give the reader some intuitions that will make it easier to read the more technical literature on this topic such as [3].

A finite state automaton (FSA) is like a directed graph with symbols on the edges (we call the edges "arcs" in this context), plus an initial state and final states. A weighted finite state automaton (WFSA) is the same, but with weights on the edges (and on the final states). A weighted finite state *transducer* (WFST) is like a weighted FSA but with two symbols on each edge (an "input" and "output" symbol). In all these cases, a special symbol ϵ (epsilon) is allowed where a normal symbol would normally occur; this means "there is no symbol here". An FSA is said to *accept* a sequence if there is a path from the initial state to a final-state with exactly that sequence on its arcs (not counting ϵ's).

2.1. Equivalence

Two FSAs are *equivalent* if they accept the same set of symbol sequences. WFSAs are equivalent if they accept the same symbol sequences with the same weights. For WFSTs, when testing equivalence we are not comparing sequences but pairs of sequences: (input-sequence, output-sequence). Equivalence is an important notion here because determinization must preserve equivalence.

2.2. Semirings

The word "semiring" is often encountered in the FST literature. Semirings can be thought of as a generalization of real-valued weights; a semiring is a set of objects that must contain two special elements, named 0 and 1; and two operators named addition ($+$) and multiplication ($\times$); and these must all satisfy certain axioms.

If the FST represents a Markov model and the weights represent probabilities, we would multiply the weights along paths and, if we encounter alternative paths with the same symbols on them, we would sum the weights. This corresponds to the "real semiring", where 0, 1, $+$ and $\times$ all have their everyday interpretation. If we were concerned about numerical overflow we might store these probabilities in log-space, and this is called the "log semiring". If we were using an FST to solve some kind of shortest-path problem, we would sum weights along paths and take the minimum if there were two paths with the same symbol sequence. This is called the "tropical semiring". There are also more complicated semirings involving symbol sequences ("gallic semiring") and so on; new semirings are easy to construct as the axioms are not hard to satisfy. See [4, 5] for the technical explanation.

2.3. Deterministic FSAs

The notion of being "deterministic" is defined for finite state acceptors, not transducers. When we speak of determinizing transducers it always involves some kind of trick to turn the transducer into an acceptor first, and the specific trick depends on the context. Generally it means either encoding the output symbol into the weight ("gallic semiring") leaving only the input symbol, or encoding symbol pairs into single symbols.

A finite state acceptor is deterministic if no state has two arcs leaving it with the same symbol. It is ϵ-free if it has no arcs with the symbol ϵ. These two notions are connected because in most of the situations where we want an automaton to be deterministic, we also want it to be ϵ-free.

2.4. Determinization algorithms

The task of *determinization*, whether in an FSA or WFSA context, is, for a given input, to find a deterministic equivalent. This is always possible for FSAs and for acyclic WFSAs, but there are some WFSAs that do not have a deterministic equivalent (c.f. the "twins property" [6]).

In [7], ϵ-removal is separated from determinization, i.e. we first remove arcs with ϵ (preserving equivalence, of course) and then determinize. There is older literature in which determinization and ϵ-removal are done as part of the same algorithm [3]. Removing ϵ's separately from determinization is very convenient for explanation and proof, but there are situations where it is not practical: for instance, in the "exact" lattice generation scenario in [1] there are too many ϵ's to remove without having the graph blow up enormously. The determinization algorithm used in the implementation of the "exact lattice generation" paper removes ϵ's as part of determinization.

Determinization algorithms operate on sets of states (or, in the weighted case, weighted sets of states). A set of states in the input FSA corresponds to a single state in the output FSA. The algorithm maintains a map from sets of states in the input FSA, to states in the output FSA; there is a queue of sets/output-states that have not been processed yet. Processing one of these sets involves enumerating all the symbols that could leave any state in the set, and for each symbol, finding the "successor set". If ϵ removal is done as part of the determinization algorithm, ϵ arcs have to be treated specially.

2.5. Pruned determinization

The task of pruned determinization is the same as that of determinization except we are not required to preserve paths in the output whose weight is too much worse than that of the best path. The motivation is to avoid the determinization "blowing up" the size of the lattice, which can occasionally happen in the ASR context [8]. The user would specify a beam (*lattice-beam*, in Kaldi), and possibly a maximum allowed number of states or arcs. Converting a normal determinization algorithm to a pruned determinization algorithm essentially means using a priority queue instead of a normal queue, and adding a stopping criterion. Computing the appropriate priorities requires working out in advance, for each state, the weight from that state to the end of the input WFSA, called the β score.

Before running the determinization algorithm we always prune the state-level lattice, removing any arcs and states that are not within *lattice-beam* of the best path.

2.6. Determinization for lattices

In "exact lattice generation" [1] we construct a word-level lattice by determinizing a state-level lattice. A state-level lattice has a state for each pair (decoding-graph-state, frame) that was not pruned away during decoding, and each arc spans at most one frame. When we determinize this, we determinize on the word level. During determinization we view the state-level lattice as an acceptor whose labels correspond to words (these will be ϵ for most arcs); the senone[1] labels are encoded into the weight. The specific semiring is described in [1]; the idea is to keep only the senones from the best path corresponding to each word sequence. We determinize this state-level lattice in an algorithm that also incorporates ϵ removal and pruning.

3. RELATED WORK

Previous work has addressed (offline) determinizing non-functional WFSTs such that a single best-scoring path of each input sequence along with the best output sequence is generated. These methods work by designing semirings with special binary operations, mapping the WFST into an equivalent WFSA in the designed semirings, applying WFSA determinization, and finally converting the results back to a WFST that preserves arc-level alignments. Povey et al. [1] described their method in the context of an exact ASR lattice generation task. In similar work, [9] and [10] used a special lexicographic semiring for the task of part-of-speech tagging disambiguation, and [11] used this semiring for hierarchical phrase-based decoding with push-down automata in statistical machine translation. In a related work, [12] presented a faster disambiguation algorithm showcased in the re-scoring task of a machine translation system with bilingual neural networks.

Rybach et al. [8] presented an optimization method for determinization followed by minimization, that produces a deterministic minimal ASR lattice that retains all paths within specified weight and lattice size thresholds. To create compact lattices, they apply in sequence pruning, weighted determinization, and minimization.

Some related work proposed algorithms for incremental determinization of acyclic [13] and cyclic [14] finite automata. In the case that a non-deterministic finite automaton (NFA) is repeatedly extended, these algorithms make up the new deterministic finite automaton (DFA) as an extension of the previous DFA. Rather than starting from scratch the generation of the DFA equivalent to the new NFA, the algorithm applies the set of actions which are sufficient for transforming the previous DFA into the new DFA. As concluded in [14], if expanding automata includes ϵ-transitions, the proposed incremental determinization is not always the best choice as opposed to the regular determinization. In fact, the choice depends to a large extent on the specific application domain.

[1]context-dependent HMM state, or phone

4. INCREMENTAL DETERMINIZATION: OVERVIEW

Suppose we were to first cut the FSA to be determinized into multiple consecutive pieces, corresponding (for instance) to ranges of frames in a lattice. We would like to determinize these pieces separately and then somehow connect them together. We devised a way to connect the pieces together while keeping the result deterministic, by introducing special symbols at the points where we split apart the input FSA; the special symbols correspond to the states at the cut points.

When we determinize a new chunk, we also need to redo the determinization for a subset of the states of the previously determinized part–essentially, the states which have arcs that end in final-states and which are reachable from such states. These become part of the new piece to be determinized. This means that when the chunk size gets smaller than approximately the length of a word in frames, we start doing significant extra work.

5. INCREMENTAL DETERMINIZATION: SIMPLE CASE

To explain the basic principle of our algorithm, we consider a simplified case. Suppose we are determinizing an unweighted acceptor (FSA) F, and we want to do the determinization in two pieces.

We divide the states in the FSA into two nonempty sets $\mathcal{A}$ and $\mathcal{B}$, with the property that for any $s \in \mathcal{B}$, the destination-states of arcs leaving s are also in $\mathcal{B}$. (Intuition: "consecutive pieces"). An example of FSAs $\mathcal{A}$ and $\mathcal{B}$ are shown in Figure 1(a). We are going to first determinize the states in $\mathcal{A}$ and then those in $\mathcal{B}$. The algorithm inserts special symbols that we call *state labels* (because they identify states in FSTs) to make it possible to (mostly) separate the two pieces.

5.1. Determinizing the first half

We firstly construct the FSA A as follows. Its states are:

- All states $s \in \mathcal{A}$ (these are final if they were final in F).
- Those states in $\mathcal{B}$ to which an arc exists from a state in $\mathcal{A}$; these are not final.
- An extra final-state f.

For each state $s \in \mathcal{B}$ that is included in FSA A, we add an arc from s to the extra final-state f, with a specially created label that will identify the state s. We call these labels *state labels*. We then determinize FSA A; call the result $\det(A)$. We are using a notation where the det operator *includes* ϵ-removal, so the result is ϵ-free. An example of FSA A and $\det(A)$ are shown in Figure 1(b).

5.2. Determinizing the second half

We next construct the FSA B. The construction of B is a little more complicated than that of A because we need to include some information from $\det(A)$ as well as from F. For purposes of exposition, we assume that the state labels used for $\det(A)$ do not overlap with those in F or A.

Define a *redeterminized state* as a state in $\det(A)$ that has an arc with a *state label* leaving it, or which is reachable from such a state. A *non-redeterminized state* is a state in $\det(A)$ that is not a *redeterminized state*.

Any state in $\det(A)$ which has an arc with a *state label* entering it will be final and will only have arcs with *state labels* entering it. Call this a *splice state* (because it can be thought of as the place

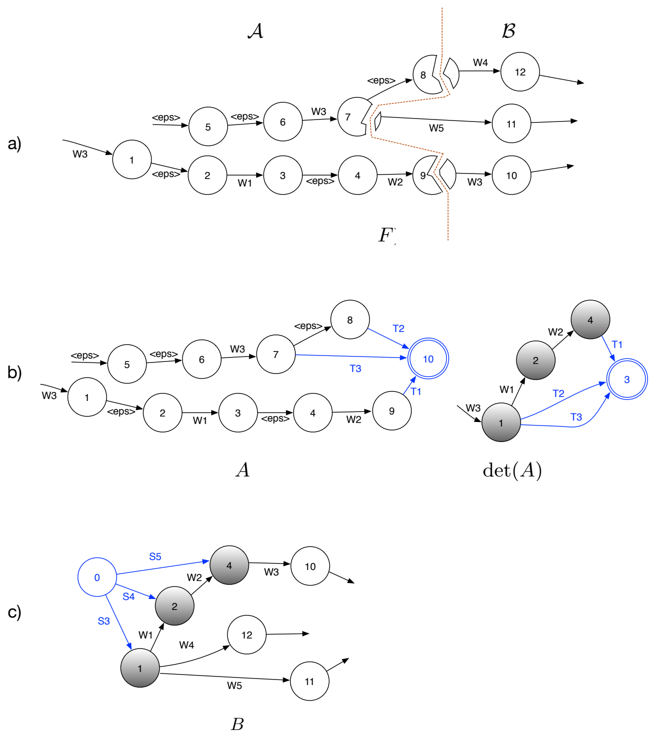

Fig. 1. Examples of incremental determinizing FSA F in Section 5. The arc symbols include words (W1, W2, …), state labels (T1, T2, T3) in FSA A and state labels (S3, S4, S5) in FSA B. The states of gradient colors are redeterminized states.

where we splice $\det(A)$ and B together). *Splice states* will also be *redeterminized states*.

The states in the FSA B are:

- An initial state i.
- Each of the *redeterminized states* in $\det(A)$ mentioned above which is not also a *splice state*; these are final if they were final in $\det(A)$.
- All states in $\mathcal{B}$; these are final if they were final in F.

We add arcs to B as follows:

- All arcs leaving states in $\mathcal{B}$.
- Arcs from i to each *redeterminized state* that is either initial in $\det(A)$ or that has an arc entering it from a state that is not a *redeterminized state*. We put labels on these arcs (*state labels*) which identify the destination *redeterminized states*.
- All arcs leaving *redeterminized states* that are not *splice states*; but if the destination state was a *splice state*, we make the arc in B end at the state in $\mathcal{B}$ identified by the label on the arc; and the label on the arc in B will be ϵ.

An example of FSA B is shown in Figure 1(c). Then of course we determinize B to get $\det(B)$.

5.3. Connecting the pieces together

We connect $\det(A)$ and $\det(B)$ together as follows to construct a spliced-together FST C. Again, assume the state labels of $\det(A)$ and $\det(B)$ do not overlap.

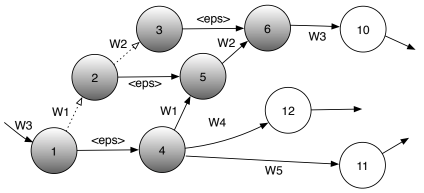

Fig. 2. The resultant FSA C after connecting the pieces in Figure 1(b) and (c) together. The states of gradient colors are redeterminized states before connecting. The dotted arcs are deleted after connecting.

The states of C consist of:

- All *non-redeterminized states* in $\det(A)$.
- All states in $\det(B)$ which are not initial.
- Only if the initial state of $\det(A)$, $i[\det(A)]$, is a *redeterminized state*, we include $i[\det(B)]$ (this is a pathological case where we do not re-use any work).

All arcs in $\det(B)$ except those leaving its initial state are included in C. All arcs in $\det(A)$ that begin and end at *non-redeterminized states* are included in C. For each arc leaving a *non-redeterminized state* n in $\det(A)$ and entering a *redeterminized state* r, if there is an arc from $i[\det(B)]$ with a label corresponding to r on it and entering some state s, then we include in C an arc from n to s with label ϵ.

We let the initial state of C be the initial state of $\det(A)$ if it is not a *redeterminized state*; otherwise we use the initial state of $\det(B)$. An example of FSA C is shown in Figure 2.

5.4. Removing epsilons

The final result $\det(F)$ is obtained by removing ϵ's from C; this is quite straightforward to do since there will not be successive ϵ's.

6. APPLICATION TO INCREMENTAL LATTICE DETERMINIZATION

Above we explained the basic idea of algorithm for an unweighted acceptor. There are a few other issues that come into play in incremental lattice determinization.

One obvious issue is that we are splitting the state-level lattice into not two, but many chunks, and for interior chunks we need to put the special *state label* symbols mentioned above at both the beginning and end of the chunk. We may also want to remove non-coaccessible states (states that cannot reach a final-state) after appending chunks together; these can arise due to pruning in Section 2.5. We always do it once after the end of the utterance [2]. Another option is pruning the appended lattice with the same beam in Section 2.5, called the *final pruning after determinization*

6.1. Handling weights

The FSTs we deal with in lattice generation are weighted, so there are certain extra details that we need to take care of with regard to the weights, when constructing the combined FST (for instance, taking final-probabilities into account when splicing the determinized pieces together). This is fairly straightforward for those familiar with FSTs; the reader may look at our code for more details[3].

The lattice determinization algorithm works in a semiring where both probabilities and alignment information are encoded into the weight, but when we store the state-level lattice we do not use this weight format; we put the alignment information on the input label. This requires certain format changes in the application of the algorithm above: for instance, when we include parts of $\det(A)$ in the FST B mentioned above we need to change the format from Kaldi's "compact lattice" format (alignments in weights) to state-level lattice format (alignments in ilabels).

6.2. Pruned determinization

Another issue that we need to bear in mind is that our determinization algorithm actually uses pruning, as mentioned in Section 2.5. This means that we need to take some extra care, when determinizing chunks of the state-level lattice, to get the initial and final costs right. (By "initial costs" we actually mean the probabilities on the arcs from the initial state in non-initial chunks– the arcs that have *state labels* on them). The initial costs would correspond to the "forward cost" of those states in the lattice, and the final cost would correspond to the "backward cost", computed backward from the most recently decoded frame using β scores in Section 2.5. After determinizing the chunks we then need to cancel out these costs, which is possible by referring to the *state labels*.

6.3. Deciding the chunks

We need to decide how to split up the state-level lattice into chunks to be determinized. Essentially this is a user-level decision, but for our experiments here we configure it with a *determinize-period* (which is the number of frames we wait between forming chunks, e.g. 20).

There is also another parameter we use for our experiments here, a *determinize-delay* which is the number of most-recently-decoded frames that we exclude from being determinized (e.g. 10). The reason for the *determinize-delay* parameter is that in the most recent frames, there will tend to be a large number of states within the pruning beam, which will make determinization slower. The following paragraph explains the background to this, specifically: why only recent frames have a large number of states active.

As mentioned in Section 2.5 we prune the state-level lattice before determinization. (This is a pre-existing feature of our decoders, not something new). If we are not yet at the end of the utterance, before doing this pruning we make each currently-active state final with a cost that is the negative of the forward-cost of that state, so each currently-active state is on a path with the same cost as the best path. This guarantees that the pruning process will not remove any path which will *later* have a cost that is worse than the best cost by more than *lattice-beam*, as done in [15].

6.4. Adaptive chunking

A straightforward schedule would be determinizing the chunks at fixed intervals of *determinize-period* defined in Section 6.3. However, it is more computationally efficient if we divide chunks so that at the end of chunks there are fewer active states, resulting in less information to keep track of and less computational overhead. For this reason, we follow an "adaptive chunking" method to utilize future information from backward costs to the lattice pruning phase.

[2]The frequency can be controlled by users.

[3]This should be part of Kaldi by the time of publication; search the code for 'incremental'.

We pre-define a threshold, *determinize-max-active*. For every fixed interval of *determinize-period* frames, we pick the longest chunk possible whose last frame contains no more active states than this threshold, and determinize up to there only; we skip determinization for this step if no such chunk could be found. In either case, the remaining unprocessed frames will be added to the subsequent chunk. For the finalizing step, we process all the unprocessed frames till the end. In the worst case scenario when this threshold was never met before, all of the frames will be determinized in this last step, however, this rarely happens for a reasonably-chosen *determinize-max-active* value.

6.5. Partial determinization

Determinization for ASR lattices can occasionally blow up [8]; and we do not want those pathological cases to cause our incremental determinization algorithm to become very slow. Therefore we make an option available to our algorithm to avoid re-determinizing states that are too far back in the past. This is in addition to the pruned determinization mentioned above. With reference to the notation in Section 5, we make a small change to the structure of $\det(A)$, namely: for any state s in $\det(A)$ that is more than *redeterminize-max-frames* earlier than the last frame in $\det(A)$, and that has arcs leaving it that have *state labels* and also arcs that have normal word labels, we do the following procedure. We add a new state, put an ϵ transition from s to that new state, and move the arcs with *state labels* from s to that new state. This avoids the need to redeterminize too many states that are far in the past. We remove the ϵ later on; the result is a lattice that is not completely deterministic, meaning a small percentage of states have more than one arc with the same label on. (Most algorithms on lattices will work on incompletely-determinized lattices.)

We do not show any experiments with partial determinization because we were unable to find a scenario where it showed a clear benefit (pathological blowup of determinization of ASR lattices is quite rare).

7. EXPERIMENTAL RESULTS

7.1. Experimental Setup

In the LibriSpeech Corpus [16], we used a time-delay deep neural network (TDNN) model trained by lattice-free maximum mutual information (LF-MMI) criterion with the same setup in [17]. Evaluation is carried out on *dev_other* and *test_other* sets in LibriSpeech [4]. The 3-gram language model in this corpus pruned with threshold 3×10^{-7} using SRILM [18] is used. The decoding baseline is the Kaldi offline decoder [19], *latgen-faster-mapped*. We use beam pruning and histogram pruning in decoding, and the lattice pruning interval is 20. In the proposed method, we use *determinize-delay=20*, *determinize-period=20*, *determinize-max-active=50* (defined in Section 6.3) and we do the *final pruning after determinization* by default.

To evaluate the precision of the decoder, 1-best results and lattice quality are both examined. We report word error rate (WER) for the former, and lattice Oracle WER (OWER) [20] along with lattice density (measured by arcs/frame) [21] for the latter. To show the efficiency of the algorithm, we measure real-time factor (RTF) to evaluate the decoding speed. Because of the focus of this work, we report RTFs excluding AM inference time [5], denoted as search RTF (no AM). One of the main goals of this work is to reduce online streaming latency. The practical latency can be related to many factors [6]: model latency during inference [22], WFST decoding speed [23], and lattice processing here. As this work is mainly focused on the lattice processing, we report *Latency* as the time (measured by millisecond, *ms*) taken after decoding the last frame until the decoding is done. This includes the time spent in final lattice pruning and determinization and excludes the time spent in model inference and decoding.

[4]These are more challenging test sets compared to *dev_clean* and easier to cause search errors in decoding algorithms.

Table 1. Incremental determinization vs. baseline determinization

Test set	Incremental?	WER (%)	Oracle WER (%)	Lattice Density	Latency (ms)	RTF (no AM)
dev_other	×	10.88	1.80	25.55	37.32	0.76
	√	10.88	1.82	26.57	13.28	0.76
test_other	×	11.31	1.91	27.71	36.91	0.71
	√	11.31	1.92	28.92	14.32	0.70

7.2. Performance

Table 1 compares incremental determinization with the baseline decoder with offline lattice determinization [1] on two test sets. The incremental determinization algorithm is with the default settings mentioned in Section 7.1. Both decoders get the same 1-best result since the incremental determinization only affects the lattice processing. The proposed method obtains moderately larger lattice densities with similar Oracle WER.

Incremental determinization significantly reduces the *Latency* defined in Section 7.1. Our profiling shows that lattice determinization of the baseline decoder causes around 60% of this utterance-end latency, while that of the proposed method causes less than 10%.

Although the utterance-end latency improvements may seem fairly modest (a factor of 3), the time taken for determinization in the baseline decoder is proportional to the utterance length, so the difference would be much more significant for long utterances (average length here was 6 seconds).

7.3. Analysis

In this section, we further analyze this algorithm from several perspectives. Experiments reported are run on the *test_other* set in this section.

7.3.1. The effect of Adaptive Chunking

Table 2 shows the effect of adaptive chunking (*AC*) and the final pruning after determinization (*FP*) in our incremental determinization algorithm. By comparing rows 1, 2 and rows 3, 4, we see that adaptive chunking always helps to reduce lattice density while having similar OWERs, versus the non-adaptive, fixed *determinize-delay* and *determinize-period* counterparts, regardless of whether final pruning is used.

Adaptive chunking also reduces RTF. This is probably because it leads to fewer tokens being active on the boundaries between chunks,

[5] The AM inference RTF is 0.34 in our setup.

[6]Our preliminary experiments on typical online acoustic models and WFST decoder show that lattice processing takes up more than half of the overall latency.

Table 2. The effect of Final Pruning After Determinization (*FP*) and Adaptive Chunking (*AC*)

FP	*AC*	WER (%)	Oracle WER(%)	Lattice density	Latency (ms)	RTF (no AM)
×	×	11.31	1.86	54.73	18.48	0.83
×	√	11.31	1.88	36.91	13.98	0.70
√	×	11.31	1.94	31.14	21.71	0.84
√	√	11.31	1.92	28.92	14.32	0.70

reducing the number of *redeterminized-states*. It also improves latency, which is surprising, because it will cause the last chunk that we determinize to be longer (on average).

7.3.2. *The effect of determinize-delay*

Figure 3 shows the effect of the *determinize-delay* defined in Section 6.3, when both adaptive chunking and final pruning are turned off. It shows that without the delay (*determinize-delay* = 0), the lattice density and RTF are significantly larger. This is expected, because without the benefit of some future frames for pruning, we have to determinize state-level lattices with too many states. The frame shift at the neural net output is 30ms, so the default *determinize-delay* of 20 corresponds to a delay of 0.6 seconds. This contributes to the latency between decoding a frame and when the determinized partial lattice can be obtained (applicable while we are decoding an utterance, not at the final frame), so for applications where a determinized lattice is desired in real time, *determinize-delay* is a key parameter that trades off between latency and real-time factor. Caution: this is not the same as the latency referred to in the tables, which is always utterance-end latency.

7.3.3. *The effect of determinize-period*

Figure 4 shows the effect of varying *determinize-period*, with *determinize-delay* fixed at 20, and without final lattice pruning. Larger *determinize-period* reduces lattice density because of better pruning and fewer chunks to be processed. (The difference in lattice density would be greatly reduced if we used final pruning). As with *determinize-period*, larger *determinize-delay* will increase the average within-utterance latency.

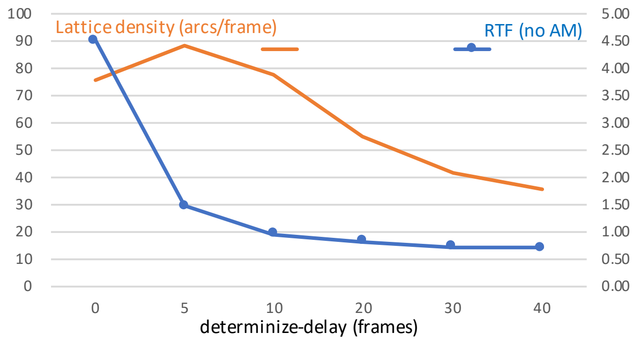

Fig. 3. Lattice density and RTF (no AM) vs. *determinize-delay* (OWER=1.87).

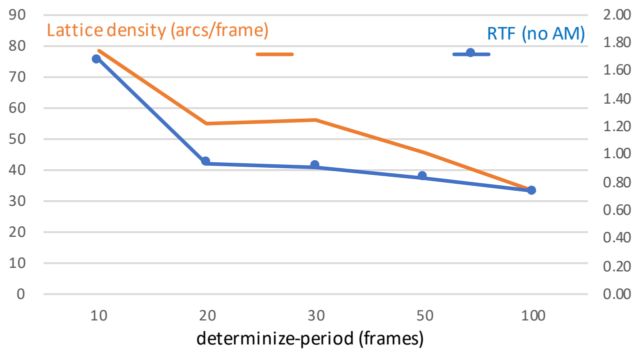

Fig. 4. Lattice density and RTF (no AM) vs. *determinize-period* (OWER=1.87).

7.3.4. *The effect of redeterminize-max-frames*

Redeterminize-max-frames is a parameter that can be set to avoid redeterminizing already-determinized states that are too far back in time. Our default value of infinity means that we will determinize as many states as necessary to ensure fully deterministic output. This parameter was introduced mostly as a mechanism to limit the effect of pathological data or models on determinization time.

Table 3 shows the effect of setting this parameter to finite values. As we decrease it, lattice density increases and Oracle WER improves slightly, and the proportion of states in the lattice that are deterministic decreases. (The nondeterminism of the lattices may or may not be a problem, depending on the downstream task.)

Table 3. The Effect of *redeterminize-max-frames*.

Redet. Frames	Oracle WER(%)	Lattice density	*Det. Portion (%)*
3	1.83	67.78	96.29
5	1.84	63.81	97.79
10	1.85	59.07	99.26
20	1.86	56.21	99.93
Inf.	1.86	54.73	100

8. CONCLUSION

We have introduced a lattice determinization algorithm that can operate on an input FSA that arrives incrementally, as in real-time decoding for ASR. (Note: this is different from on-demand determinization, which is where the input is all known at once but the states of the output are obtained only as requested). Our algorithm is useful when word-level lattices are needed prior to the end of the utterance, and it also reduces the latency due to determinization at the end of the utterance.

Future work may include the combination of the proposed method and other speedups of ASR decoding, e.g. GPU WFST decoding [23] and phone synchronous decoding [24].

9. REFERENCES

[1] Daniel Povey, Mirko Hannemann, Gilles Boulianne, Lukáš Burget, Arnab Ghoshal, Miloš Janda, Martin Karafiát, Stefan

Kombrink, Petr Motlíček, Yanmin Qian, et al., "Generating exact lattices in the WFST framework," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 4213–4216.

[2] Xavier Aubert and Hermann Ney, "Large vocabulary continuous speech recognition using word graphs," in *1995 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1995, pp. 49–52.

[3] Mehryar Mohri, "Weighted automata algorithms," in *Handbook of weighted automata*, pp. 213–254. Springer, 2009.

[4] Werner Kuich and Arto Salomaa, *Semirings, Automata, Languages*, Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2012.

[5] Jonathan Golan, *Semirings and their Applications*, Springer Science & Business Media, 2013.

[6] Cyril Allauzen and Mehryar Mohri, "Efficient algorithms for testing the twins property," *Journal of Automata, Languages and Combinatorics*, vol. 8, no. 2, pp. 117–144, 2003.

[7] Takaaki Hori and Atsushi Nakamura, "Speech recognition algorithms using weighted finite-state transducers," *Synthesis Lectures on Speech and Audio Processing*, vol. 9, no. 1, pp. 1–162, 2013.

[8] David Rybach, Michael Riley, and Johan Schalkwyk, "On lattice generation for large vocabulary speech recognition," in *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2017, pp. 228–235.

[9] Izhak Shafran, Richard Sproat, Mahsa Yarmohammadi, and Brian Roark, "Efficient determinization of tagged word lattices using categorial and lexicographic semirings," in *2011 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2011, pp. 283–288.

[10] Richard Sproat, Mahsa Yarmohammadi, Izhak Shafran, and Brian Roark, "Applications of lexicographic semirings to problems in speech and language processing," *Computational Linguistics*, vol. 40, no. 4, pp. 733–761, 2014.

[11] Cyril Allauzen, Bill Byrne, Adrià de Gispert, Gonzalo Iglesias, and Michael Riley, "Pushdown automata in statistical machine translation," *Computational Linguistics*, vol. 40, no. 3, pp. 687–723, 2014.

[12] Gonzalo Iglesias, Adrià de Gispert, and Bill Byrne, "Transducer disambiguation with sparse topological features," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015, pp. 2275–2280.

[13] Gianfranco Lamperti and Michele Scandale, "From diagnosis of active systems to incremental determinization of finite acyclic automata," *AI Communications*, vol. 26, no. 4, pp. 373–393, 2013.

[14] Simone Brognoli, Gianfranco Lamperti, and Michele Scandale, "Incremental determinization of expanding automata," *The Computer Journal*, vol. 59, no. 12, pp. 1872–1899, 2016.

[15] Andrej Ljolje, Fernando Pereira, and Michael Riley, "Efficient general lattice generation and rescoring," in *Sixth European Conference on Speech Communication and Technology*, 1999.

[16] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur, "LibriSpeech: an ASR corpus based on public domain audio books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 5206–5210.

[17] Daniel Povey, Vijayaditya Peddinti, Daniel Galvez, Pegah Ghahremani, Vimal Manohar, Xingyu Na, Yiming Wang, and Sanjeev Khudanpur, "Purely sequence-trained neural networks for ASR based on lattice-free MMI," in *Interspeech*, 2016, pp. 2751–2755.

[18] Andreas Stolcke, "SRILM - an extensible language modeling toolkit," in *Seventh international conference on spoken language processing*, 2002.

[19] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al., "The Kaldi speech recognition toolkit," in *2011 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2011.

[20] Björn Hoffmeister, Tobias Klein, Ralf Schlüter, and Hermann Ney, "Frame based system combination and a comparison with weighted ROVER and CNC," in *Ninth International Conference on Spoken Language Processing*, 2006.

[21] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al., "The HTK book," *Cambridge university engineering department*, vol. 3, pp. 175, 2002.

[22] Vijayaditya Peddinti, Yiming Wang, Daniel Povey, and Sanjeev Khudanpur, "Low latency acoustic modeling using temporal convolution and LSTMs," *IEEE Signal Processing Letters*, vol. 25, no. 3, pp. 373–377, 2018.

[23] Zhehuai Chen, Justin Luitjens, Hainan Xu, Yiming Wang, Daniel Povey, and Sanjeev Khudanpur, "A GPU-based WFST decoder with exact lattice generation," in *Interspeech*, 2018, pp. 2212–2216.

[24] Zhehuai. Chen, Yimeng Zhuang, Yanmin Qian, and Kai Yu, "Phone synchronous speech recognition with CTC lattices," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 1, pp. 90–101, 2017.