Algorithms for Acyclic Weighted Finite-State Automata with Failure Arcs

Anej Svete[1]   **Benjamin Dayan**[1]
Tim Vieira[2]   **Ryan Cotterell**[1]   **Jason Eisner**[2]
[1]ETH Zürich   [2]Johns Hopkins University
{asvete, bdayan}@ethz.ch
ryan.cotterell@inf.ethz.ch   {timv, jason}@cs.jhu.edu

Abstract

Weighted finite-state automata (WSFAs) are commonly used in NLP. Failure transitions are a useful extension for compactly representing backoffs or interpolation in n-gram models and CRFs, which are special cases of WFSAs. The pathsum in ordinary acyclic WFSAs is efficiently computed by the backward algorithm in time $\mathcal{O}(|E|)$, where E is the set of transitions. However, this does not allow failure transitions, and preprocessing the WFSA to eliminate failure transitions could greatly increase $|E|$. We extend the backward algorithm to handle failure transitions directly. Our approach is efficient when the average state has outgoing arcs for only a small fraction $s \ll 1$ of the alphabet Σ. We propose an algorithm for general acyclic WFSAs which runs in $\mathcal{O}(|E| + s|\Sigma||Q||\mathcal{T}_{\max}| \log |\Sigma|)$, where Q is the set of states and $|\mathcal{T}_{\max}|$ is the size of the largest connected component of failure transitions. When the failure transition topology satisfies a condition exemplified by CRFs, the $|\mathcal{T}_{\max}|$ factor can be dropped, and when the weight semiring is a ring, the $\log |\Sigma|$ factor can be dropped. In the latter case (ring-weighted acyclic WFSAs), we also give an alternative algorithm with complexity $\mathcal{O}(|E| + |\Sigma||Q| \min(1, s|\boldsymbol{\pi}_{\max}|))$, where $|\boldsymbol{\pi}_{\max}|$ is the size of the longest failure path.

https://github.com/rycolab/
failure-backward

1   Introduction

Weighted finite-state automata (WFSAs) are a common formalism in NLP. Many popular models are special cases, e.g., n-gram language models (Brown et al., 1992), conditional random fields (CRFs: Lafferty et al., 2001), maximum-entropy Markov models (McCallum et al., 2000), and semi-Markov models (Sarawagi and Cohen, 2004). In current practice, the weights in the WFSAs are often derived from a neural network, and neuralized WFSAs constitute the state of the art on a variety of common tasks in NLP (Rastogi et al., 2016; Schwartz et al., 2018; Lin et al., 2019; Jiang et al., 2021; Rijhwani et al., 2021; Alon et al., 2022). WFSAs are also increasingly being used for the design (Shen et al., 2019; Schwartz et al., 2018) and analysis (Peng et al., 2018; Hewitt et al., 2020; Hahn, 2020; Chiang and Cholak, 2022) of neural architectures.

Failure transitions are a useful augmentation of standard WFSAs. First introduced in the context of string matching (Aho and Corasick, 1975), they can be used to represent backoff n-gram language models (Allauzen et al., 2003), higher-order CRFs, and variable-order CRFs (VoCRFs; Vieira et al., 2016) in a more compact way. They represent "default" transitions out of states when no other transition is possible. For example, in backoff n-gram language models, a weighted failure transition from a higher-order history to a lower-order history (e.g., from a 4-gram to a 3-gram) is used to back off before reading a word that was rarely observed with the higher-order history, so that it was not worth including a dedicated transition for that word.

The **pathsum** computes the total weight of all the paths in a WFSA graph, where the weights may fall in any semiring.[1] Examples include finding the highest-weighted path for Viterbi decoding, computing the posterior marginals (inference) in hidden Markov models, and computing the normalizing constant in CRFs. The pathsum is particularly efficient to compute in *acyclic* WFSAs with the *backward algorithm*, whose runtime is $\mathcal{O}(|E|)$. However, the special semantics of failure transitions mean that the ordinary backward algorithm cannot be applied (nor can the forward algorithm). Failure transitions must first be replaced by normal ones (Alg. 2 below), resulting in the failure-expanded transition set $\overline{E}$, which can contain up to $|Q|^2|\Sigma|$ transitions. Replacing failure transitions, therefore, undoes the compaction

---

[1]We formally define the problem in §2.

afforded by them. This is especially expensive ($|\overline{E}| \gg |E|$) for backoff language models, for example, where each of the many 4-gram states only has explicit transitions in E for symbols a that were observed in training data to follow that 4-gram, but has transitions in $\overline{E}$ for every $a \in \Sigma$. For example, Penn Treebank tagging has $|\Sigma| = 36$ and Czech morphological tagging has $|\Sigma| > 1000$ (Hajič and Hladká, 1998). While Allauzen et al. (2003) present an $\mathcal{O}(n^2|\Sigma||Q|)$ method to preprocess a (possibly cyclic) n-gram language model WFSA with failure transitions such that the pathsum remains identical, their method only applies to the case of the tropical semiring.

In this paper, we study the problem of efficiently computing the pathsum in WFSAs with failure transitions over general semirings. We specifically focus on *acyclic* WFSAs,[2] introducing several algorithms, all based on the backward algorithm, that take advantage of the compact structure induced by the failure transitions. Our improvements are strongest for WFSAs that are *sparse* in a sense to be defined shortly. We summarise our contributions as follows:

- We present simple baseline algorithms using failure transition removal (§3.1) and memoization (§3.2).

- We present an algorithm for computing the pathsum of ring-weighted WFSAs, utilizing subtraction (§4).

- With some extra work to avoid subtraction (§5), we extend the algorithm to general semirings (§6).

2 Preliminaries

This section defines WFSAs, the pathsum problem, the backward algorithm, and failure transitions.

Definition 1. A **semiring** is a 5-tuple $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ where $\mathbb{K}$ is a set equipped with operations $\oplus$ and $\otimes$, s.t. $(\mathbb{K}, \oplus, \mathbf{0})$ is a commutative

---

[2]We have also worked out several novel algorithms for *cyclic* WFSAs with failure transitions. However, the 9-page format of EMNLP submissions meant that we had to save these for a future manuscript. Moreover, we view the split into acyclic and cyclic cases as natural, as our algorithms for cyclic automata are based on the more complex algorithm of Lehmann (1977). While the acyclic algorithms cannot be run on a backoff language model (one of our examples), they can be used, for example, to compute the total language model probability of all paths in an acyclic lattice that may have wildcard arcs and/or failure arcs. Even though the language model is cyclic, its intersection with the lattice becomes acyclic.

monoid, $(\mathbb{K}, \otimes, \mathbf{1})$ is a monoid, $\oplus$ distributes over $\otimes$, and $\mathbf{0}$ annihilates $\otimes$.

Definition 2. A **weighted finite-state automaton (WFSA)** is a 5-tuple $\mathcal{A} = \langle \Sigma, Q, E, \lambda, \rho \rangle$, where Σ is a finite alphabet, Q a finite set of states, E a collection of transitions in $Q \times \Sigma \times \mathbb{K} \times Q$, $\lambda : Q \to \mathbb{K}$ the initial-state weighting function, and $\rho : Q \to \mathbb{K}$ the final-state weighting function.

To improve readability, we render a transition (q, a, w, q') as $q \xrightarrow{a/w} q'$. We further define $E(q) \stackrel{\text{def}}{=} \{e \mid \exists a, w, q' : e = q \xrightarrow{a/w} q' \in E\}$ as the set of outgoing transitions of $q \in Q$, and $E(q, a)$ as those labeled with $a \in \Sigma$. $\Sigma(q) \stackrel{\text{def}}{=} \{a \mid E(q, a) \neq \varnothing\}$ denotes the set of transition labels in $E(q)$.

Importantly, we will assume that the graph (Q, E) is acyclic (see footnote 2). Less importantly, our definition of WFSAs does not allow ε-transitions, assuming that they have been eliminated in advance (Mohri, 2002a), which is easy in the acyclic case. Our runtime analyses assume for simplicity that *(i)* the graph is connected (implying $|E| \geq |Q| - 1$) and *(ii)* that for each $q, q' \in Q$, E contains at most one transition $q \xrightarrow{a/w} q'$ for any $a \in \Sigma$. This can always be achieved by replacing "parallel" transitions $\left\{ q \xrightarrow{a/w_i} q' \mid i \right\} \subseteq E$ with $q \xrightarrow{a/\oplus_i w_i} q'$.

Definition 3. A **path** π in a WFSA $\mathcal{A}$ is a sequence of consecutive transitions in E,
$q_0 \xrightarrow{a_1/w_1} q_1 \cdots q_{N-1} \xrightarrow{a_N/w_N} q_N$. $\mathrm{p}(\pi) \stackrel{\text{def}}{=} q_0$ and $\mathrm{n}(\pi) \stackrel{\text{def}}{=} q_N$ refer to the initial and final states of π, respectively. $\Pi(\mathcal{A})$ denotes the set of all paths in $\mathcal{A}$.

Definition 4. The **inner path weight** is defined as $\mathrm{w_I}(\pi) \stackrel{\text{def}}{=} \bigotimes_{n=1}^{N} w_n$ and the (full) **path weight** as $\mathrm{w}(\pi) \stackrel{\text{def}}{=} \lambda(\mathrm{p}(\pi)) \otimes \mathrm{w_I}(\pi) \otimes \rho(\mathrm{n}(\pi))$.

Definition 5. The **pathsum** of $\mathcal{A}$ is defined as
$$\mathbf{Z}(\mathcal{A}) \stackrel{\text{def}}{=} \bigoplus_{\pi \in \Pi(\mathcal{A})} \mathrm{w}(\pi). \tag{1}$$

The problem of computing the pathsum is sometimes also referred to as the generalized shortest-distance problem (Mohri, 2002b).

Definition 6. The **backward value** $\beta(q)$ of a state $q \in Q$ is the sum of the inner weights of all paths π starting at q right-multiplied by $\rho(\mathrm{n}(\pi))$, i.e.,
$$\beta(q) \stackrel{\text{def}}{=} \bigoplus_{\substack{\pi \in \Pi(\mathcal{A}), \\ \mathrm{p}(\pi) = q}} \mathrm{w_I}(\pi) \otimes \rho(\mathrm{n}(\pi)). \tag{2}$$

We extend this definition to state–symbol pairs $(q, a) \in Q \times \Sigma$ as

$$\beta(q, a) \stackrel{\text{def}}{=} \bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q'). \tag{3}$$

The value $\beta(q, a)$ can be seen as the result of restricting the paths contributing to $\beta(q)$ to those starting with $a \in \Sigma$.

For $S \subseteq \Sigma$ we also define

$$\beta(q, S) \stackrel{\text{def}}{=} \bigoplus_{a \in S} \beta(q, a). \tag{4}$$

Notice that $\beta(q) = \rho(q) \oplus \beta(q, \Sigma(q))$.

Naïvely computing the pathsum by enumerating all $\pi \in \Pi(\mathcal{A})$ in an acyclic WFSA would result in an exponential runtime. However, algebraic properties of semirings allow for faster algorithms (Mohri, 2002b). An example is the backward algorithm, a dynamic program which computes backward values and the pathsum in acyclic WFSAs in time $\mathcal{O}(|E|)$. It exploits the fact that, in acyclic WFSAs, Q can always be topologically sorted and the backward values can be computed in *reverse* topological order. This guarantees that the backward values of q's children will have been computed by the time we expand q, meaning that $\beta(q)$ can be computed as

$$\beta(q) \leftarrow \rho(q) \oplus \bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q') \tag{5}$$

The pseudocode is given in Alg. 1. All our algorithms are based on the backward algorithm.

Algorithm 1

1: **def** Backward($\mathcal{A}$):
2:    **for** $q \in$ ReverseTopological($\mathcal{A}$) :
3:       $\beta(q, \Sigma(q)) \leftarrow \bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q')$
4:       $\beta(q) \leftarrow \rho(q) \oplus \beta(q, \Sigma(q))$
5:    **return** $\bigoplus_{q \in Q} \lambda(q) \otimes \beta(q)$ ▷ *Equals* $\mathbf{Z}(\mathcal{A})$

2.1 Failure Transitions

We consider an extension of WFSAs where any state can have a single *fallback state* q^ϕ.

Definition 7. *A WFSA with failure transitions (WFSA-ϕ) is a 6-tuple $\mathcal{A} = \langle \Sigma, Q, E, \lambda, \rho, \phi \rangle$, where ϕ is a **failure function**—a partial function that maps some states $q \in Q$ to their **fallback state** $\phi(q) = q^\phi$.*

Fallback states can be represented by transitions

$q \xrightarrow{\phi/\mathbf{1}} q^\phi$ with a special meaning:[3] they are only traversed upon reading a symbol $a \notin \Sigma(q)$ and thus represent a default option used when no ordinary transition is available.[4] This formalization means that every state has at most one fallback state.

We do not include ϕ in Σ or ϕ-transitions in E. We denote the set of ϕ-transitions as E^ϕ and assume that $E \cup E^\phi$ still forms an acyclic graph.

ϕ-transitions can be explicitly represented in a normal WFSA by *expansion* of ϕ-transitions.

Definition 8. *Given an acyclic WFSA-ϕ $\mathcal{A} = \langle \Sigma, Q, E, \lambda, \rho, \phi \rangle$, we introduce the recursively defined **failure-expanded transition set** as follows*

$$\overline{E}(q, a) \stackrel{\text{def}}{=} \begin{cases} E(q, a) & \textbf{if } a \in \Sigma(q) \\ \overline{E}(q^\phi, a) & \textbf{if } q \text{ has a } \phi \text{ arc} \\ \varnothing & \textbf{otherwise} \end{cases} \tag{6}$$

and the set $\overline{E} \subseteq Q \times \Sigma \times \mathbb{K} \times Q$ as the union of these sets over Q and Σ.

$\overline{E}(q, a)$ is well-defined due to the assumed acyclicity of E^ϕ. It may be empty. $\overline{E}$ captures all "indirect" transitions which can be made across arbitrarily long paths of only ϕ-transitions. $\overline{\Sigma}(q)$, analogously to $\Sigma(q)$, denotes the set of outgoing symbols for $q \in Q$ in the failure-expanded WFSA-ϕ.

Definition 9. *We define the **average out-symbol fraction** s of a WFSA as*

$$s = \underset{q \in Q}{\text{mean}} \frac{|\Sigma(q)|}{|\Sigma|} = \frac{\sum_{q \in Q} |\Sigma(q)|}{|Q||\Sigma|}. \tag{7}$$

$s \in [0, 1]$ *is a measure of completeness of the WFSA. We correspondingly define $\overline{s}$, the equivalent in the failure-expanded transition set $\overline{E}$.*

We say informally that a WFSA is Σ-**sparse** if $s \ll 1$, so on average $|\Sigma(q)| \ll |\Sigma|$. Intuitively, this means that the average state only has outgoing transitions on a few distinct symbols. We will show that the runtime tradeoff between our baseline pathsum algorithm MemoizationBackward (Alg. 3) and later algorithms depends on the *difference* between s and $\overline{s}$. Our algorithms are efficient when $s \ll \overline{s}$: intuitively in the regime where failure expansion would add outgoing transitions for many new symbols.

---

[3]We use ϕ to refer to both the function and the symbol.

[4]A ϕ-transition from a non-final state could be used at the end of the input string, to try to transition to a final state and accept the string. But this case does not arise in our WFSA formalization, where every state q is final with an explicit final-state weight $\rho(q)$ (possibly $\mathbf{0}$). Separately, if we were to extend our WFSA formalization to allow ε-transitions, then a ϕ-transition could never be used at a state that also had an ε-transition, since the ε-transition would always be available.

Correcting Eq. (5) to take ϕ-transitions into account, the backward values in a WFSA-ϕ can be computed as

$$\beta(q) \leftarrow \rho(q) \oplus \bigoplus_{q \xrightarrow{a/w} q' \in \overline{E}} w \otimes \beta(q'). \qquad (8)$$

Importantly, the following equality holds

$$\beta(q, a) = \begin{cases} \displaystyle\bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q') & \textbf{if } a \in \Sigma(q) \\ \beta(q^\phi, a) & \textbf{otherwise} \end{cases} \qquad (9)$$

This follows straight from the definition of ϕ. It states that the backward values of the state-symbol pairs (q, a) in WFSA-ϕ equal the ones in a normal WFSA if an a-labeled transition can be taken; if not, the backward value is inherited from the fallback state, since the **1**-weighted ϕ-transition is taken.

Connected components of the graph formed by ϕ-transitions of a WFSA-ϕ are *trees* (specifically, anti-arborescences) since a state can have at most one outgoing ϕ-transition and the WFSA is acyclic. This motivates the following definition.

Definition 10. *Let $\mathcal{A}$ be an acyclic WFSA-ϕ. A **failure tree** $\mathcal{T}$ is a connected component of the graph formed by ϕ-transitions of $\mathcal{A}$.*

An example of a failure tree $\mathcal{T}$ is shown in Fig. 1a. We write $|\mathcal{T}|$ for the number of states in $\mathcal{T}$, with $|\mathcal{T}_{\max}|$ being the number of states in the largest failure tree and $|\pi_{\max}|$ the number of states in the longest failure path.

$\mathcal{T}_q$ denotes the failure tree containing $q \in Q$. We write $q \prec q'$ to say that q is a proper ancestor of q' in $\mathcal{T}_q$, i.e., there is a non-empty ϕ-path from q to q'.

3 Expanding Failure Transitions

The pathsum of a WFSA-ϕ can be naïvely computed by replacing the ϕ-transitions with normal ones according to the semantics of the ϕ-transitions and running the backward algorithm on the expanded WFSA. Before introducing our contributions, we present this method for pedagogical purposes. While this solution is near-optimal for non-Σ-sparse WFSAs, it can be improved for certain Σ-sparse WFSAs.
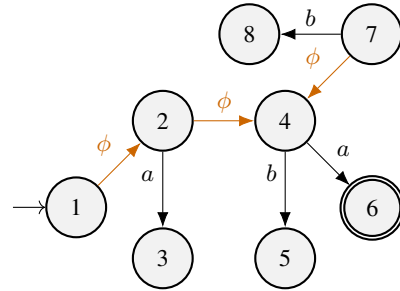
3.1 Expanding Failure Transitions

Failure expansion is a transformation of an acyclic WFSA-ϕ which replaces the ϕ-transitions while retaining acyclicity. See Alg. 2 for the pseudocode, Fig. 1b for an example of failure expansion, and App. A for an example of how the backward algorithm operates in this setting.
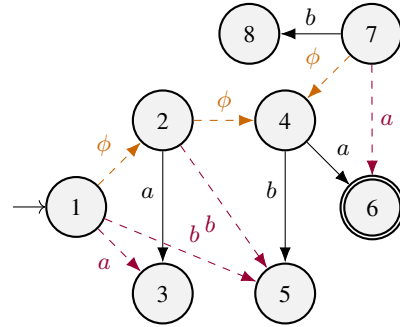

Algorithm 2

1: **def** FailureExpansion($\mathcal{A}$):
2:    $\overline{E} \leftarrow E$ ▷ *Will be updated*
3:    **for** $q \in$ ReverseTopological(E^ϕ) :
4:       $\overline{E} \leftarrow \overline{E} \cup \{q \xrightarrow{a/w} q' \mid$
5:          $q^\phi \xrightarrow{a/w} q' \in \overline{E}, a \notin \Sigma(q)\}$
6:    **return** $\overline{E}$

(a) Example of a failure tree. Its root is node 4.

(b) To expand failure transitions, the dashed transitions are added and the ϕ-transitions are removed.

In deterministic WFSA-ϕ's, failure expansion adds $\mathcal{O}(|Q||\Sigma|)$ new transitions, since each of the $|Q|$ states q gains new transitions to each of q^ϕ's children. More precisely, q gains up to $|\overline{\Sigma}(q) \setminus \Sigma(q)|$ transitions, where $\overline{\Sigma}(q) \overset{\text{def}}{=} \{a : \overline{E}(q^\phi, a) \neq \varnothing\}$ denotes the set of symbols on the outgoing transitions from q^ϕ in the failure-expanded automaton. In non-deterministic WFSA-ϕ's, however, q can gain up to $|Q||\overline{\Sigma}(q) \setminus \Sigma(q)|$ transitions, since q^ϕ might have up to $|Q|$ a-labeled transitions $\forall a \in \Sigma$. This results in $\mathcal{O}(|Q|^2|\Sigma|)$ new transitions. Following the derivation in App. B, the runtime of the backward algorithm on the ϕ-expanded WFSA-ϕ is therefore $\mathcal{O}(|\overline{E}|) = \mathcal{O}(|E| + |Q|^2(\overline{s} - s)|\Sigma|)$ in the general case and $\mathcal{O}(\overline{E}) = \mathcal{O}(|E| + |Q|(\overline{s} - s)|\Sigma|)$ in deterministic WFSAs.

3.2 Decomposing the Backward Values

The algorithms we present in later sections sidestep the need to materialize all additional transitions replacing the failure transition. They are based on

a decomposition of the backward values into two components: the local and the failure component. Using Eq. (4), we can split $\beta(q)$ into

$$\beta(q) = \rho(q) \oplus \beta(q, \Sigma) \tag{10}$$

$$\beta(q, \Sigma) = \underbrace{\beta(q, \Sigma(q))}_{\text{local}} \oplus \underbrace{\beta(q, \Sigma \smallsetminus \Sigma(q))}_{\text{failure}}. \tag{11}$$

The two terms on the right hand-side of Eq. (11) can be further expanded as

$$\beta(q, \Sigma(q)) = \bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q') \tag{12}$$

$$\beta(q, \Sigma \smallsetminus \Sigma(q)) = \bigoplus_{b \in \Sigma \smallsetminus \Sigma(q)} \beta(q^\phi, b) \tag{13}$$

except that the second term is $\mathbf{0}$ if q has no failure transition (in which case q^ϕ is not defined). $\beta(q, \Sigma(q))$ is exactly the quantity computed by Alg. 1 on line 3; our modifications never change this computation. Rather, all of our algorithms seek to simplify the computation of $\beta(q, \Sigma \smallsetminus \Sigma(q))$.

Eq. (13) makes it possible to avoid failure expansion by storing not only $\beta(q)$ but also the values $\beta(q, a)$ at each state q. Since q^ϕ will then memoize all needed $\beta(q^\phi, b)$ values, the sum (13) becomes easy to compute for any q that may back off to q^ϕ. Passing the summand $\beta(q^\phi, b)$ back to q is cheaper than passing back all of the arcs $q^\phi \xrightarrow{b/w} q' \in \overline{E}$ that contribute to that summand, as Alg. 2 does: a non-deterministic WFSA may have multiple such arcs. The pseudocode of this modification is presented in Alg. 3. Notice the additional term $\beta(q, \Sigma \smallsetminus \Sigma(q))$ on line 10 in Alg. 3, which was not needed in the backward algorithm for ordinary WFSAs. See App. A for a guided example on a small WFSA.

In the general case of non-deterministic WFSAs, failure expansion may have to loop over as many as $|Q||\Sigma \smallsetminus \Sigma(q)|$ transitions at each state q. Alg. 3 reduces this to a loop over $|\Sigma \smallsetminus \Sigma(q)|$ symbols, which is $(\overline{s} - s)|\Sigma|$ on average. The full complexity of Alg. 3 is then $\mathcal{O}(|E| + (\overline{s} - s)|\Sigma||Q|)$ (similarly to App. B).

The shortcoming of Alg. 3 is that $(\overline{s} - s)|\Sigma|$ may still be large. The terms $\beta(q^\phi, b)$ must be individually copied back to q as $\beta(q, b)$ for each $|\Sigma \smallsetminus \Sigma(q)|$. Our proposed algorithms in the following subsections avoid the overhead incurred by this copying.

4 An Algorithm with Subtraction

Alg. 3 computes $\beta(q)$ in part by summing up to $|\Sigma \smallsetminus \Sigma(q)|$ values passed back from q^ϕ. This section presents a more efficient algorithm for *ring-*

Algorithm 3

```
1: def MemoizationBackward(A):
2:     for q ∈ ReverseTopological(A) :
3:         for a ∈ Σ(q) :
4:             β(q, a) ← ⊕_{q --a/w--> q'∈E} w ⊗ β(q')
5:         β(q, Σ ∖ Σ(q)) ← 0
6:         if q has a fallback state :
7:             for b ∈ Σ ∖ Σ(q) :
8:                 β(q, b) ← β(q^φ, b)
9:                 β(q, Σ ∖ Σ(q)) ⊕= β(q^φ, b)
10:        β(q, Σ) ← β(q, Σ(q)) ⊕ β(q, Σ ∖ Σ(q))
11:        β(q) ← ρ(q) ⊕ β(q, Σ)
12:    return ⊕_{q∈Q} λ(q) ⊗ β(q)
```

---

weighted Σ-sparse WFSAs. As rings allow *subtraction*, we can compute the failure term as follows:

$$\beta(q, \Sigma \smallsetminus \Sigma(q)) = \beta(q^\phi, \Sigma) \ominus \beta(q^\phi, \Sigma(q)) \tag{14}$$

Recall that $\beta(q^\phi, \Sigma(q)) \stackrel{\text{def}}{=} \bigoplus_{a \in \Sigma(q)} \beta(q^\phi, a)$ by (4). Thus Eq. (14) effectively uses $|\Sigma(q)|$ subtractions (for $a \in \Sigma(q)$), whereas Eq. (13) used $|\Sigma \smallsetminus \Sigma(q)|$ additions (for $b \in \Sigma \smallsetminus \Sigma(q)$). In the runtime analysis, these subtractions are already covered by the $\mathcal{O}(|\Sigma(q)|)$ runtime needed for the $|\Sigma(q)|$ additions in Eq. (12). Overall, Eqs. (11), (12) and (14) compute $\beta(q, \Sigma)$ in Eq. (11) by combining $\oplus$ and $\ominus$ to *replace* just $|\Sigma(q)|$ of the summands of $\beta(q^\phi, \Sigma)$—namely, those overridden at q.

But how fast is it to *find* the subtrahends $\beta(q^\phi, a)$ for $a \in \Sigma(q)$? Eagerly storing $\beta(q, a)$ (if non-$\mathbf{0}$) for *every* $q \in Q, a \in \Sigma$ (in case it is needed during backoff) would allow constant-time lookup, but doing so would require copying $\beta(q^\phi, b)$ backward to $\beta(q, b)$ for all $b \in \Sigma \smallsetminus \Sigma(q)$, just as in Alg. 3, which would incur the same complexity of $\mathcal{O}(|\Sigma \smallsetminus \Sigma(q)|)$. So instead of computing and storing the full set of $\beta(q^\phi, a)$ values $\forall a \in \Sigma$, we will compute on demand only the ones that need replacement. This involves following ϕ-arcs forward until we find an a arc, or run out of ϕ arcs, or encounter a memo because $\beta(q^\phi, a)$ was already needed by a different ancestor of q^ϕ in its failure tree. The full algorithm is presented as Alg. 4.[5]

---

[5]In practice, it is handy for the summation in RingBackward line 2 to visit the states Q in reverse topological order, as was done explicitly in Alg. 3. This slightly simplifies implementation of $\beta(q, \Sigma)$ since whenever that function is called, the call is guaranteed to be memoized already. But this guarantee cannot apply to the $\beta(q, a)$ function, as it is called only for those a where it is needed. In particular, when $\beta(q^\phi, a)$ is

Algorithm 4

1: **def** RingBackward($\mathcal{A}$):
2:    **return** $\bigoplus_{q \in Q} \lambda(q) \otimes \beta(q)$

3: **def** $\beta(q)$:
4:    **return** $\beta(q) \leftarrow \rho(q) \oplus \beta(q, \Sigma)$

5: **def** $\beta(q, \Sigma)$:   ▷ *Memoizes its result*
6:    $\beta(q, \Sigma(q)) \leftarrow \bigoplus_{a \in \Sigma(q)} \beta(q, a)$
7:    **if** q has no fallback state : **return** $\beta(q, \Sigma(q))$
8:    $\beta(q^\phi, \Sigma(q)) \leftarrow \bigoplus_{a \in \Sigma(q)} \beta(q^\phi, a)$
9:    **return**
      $\beta(q^\phi, \Sigma) \oplus \big(\beta(q, \Sigma(q)) \ominus \beta(q^\phi, \Sigma(q)) \big)$

10: **def** $\beta(q, a)$:   ▷ *Memoizes its result*
11:    **if** $a \in \Sigma(q)$: **return** $\bigoplus_{q \xrightarrow{a/w} q' \in E} w \otimes \beta(q')$
12:    **else if** q has a fallback state : **return** $\beta(q^\phi, a)$
13:    **else return** 0

4.1 Runtime

The runtime of Alg. 4 is on the order of the number of calls to line 10, plus $|E|$ to cover all the sums in line 11 (which executes at most once for each q, a pair, thanks to memoization). Every $a \in \Sigma(q)$ results in two such calls, at lines 6 and 8; there is also a possible recursive call at line 12 if $a \in \Sigma(q')$ for at least one proper ancestor $q' \prec q$ in the failure tree (thanks to memoization, this happens at most once per q, a pair, even if there are multiple choices of q'). Thus, the overall runtime is
$$\mathcal{O}\left(|E| + \sum_{q \in Q} |\hat{\Sigma}(q)| \right), \text{ where } \hat{\Sigma}(q) \subseteq \Sigma \text{ is defined}$$
as $\bigcup_{q' \preceq q} \Sigma(q')$. A looser bound written in terms of s is $\mathcal{O}(|E| + |\Sigma||Q| \min(1, s|\boldsymbol{\pi}_{\max}|))$.[6]

We will revisit ring-weighted WFSA-ϕ's in §6.4.

5 Incrementally Modified Aggregator

The point of Alg. 4 line 9 is to replace some summands of $\beta(q^\phi, \Sigma)$ to get $\beta(q, \Sigma)$. When no subtraction operator $\ominus$ is available (e.g., if $\oplus = \max$), we can use an **aggregation data structure** that is designed to efficiently replace individual summands in a sum without using subtraction. For example, a **Fenwick tree** (Fenwick, 1994) can replace a summand and recompute the sum in $\mathcal{O}(\log N)$

time, where N is the number of summands. (Fenwick trees are similar to binary heaps; they are reviewed in App. C.) Here we merely give the interface to aggregators:

1: **class** Aggregator():   ▷ *We use γ to refer to an aggregator instance*
2:    **def** set(a: Σ, v: $\mathbb{K}$)   ▷ *Updates $\gamma(a) \leftarrow v$*
3:    **def** get(a: Σ) $\to \mathbb{K}$ ▷ *Returns $\gamma(a)$ (default 0)*
4:    **def** value() $\to \mathbb{K}$ ▷ *Returns $\bigoplus_{a \in \Sigma} get(a)$*
5:    **def** undo(n: $\mathbb{N}$)   ▷ *Reverts the last n updates*

We will represent each sum $\beta(q, \Sigma)$ in Alg. 4 as the total value of an aggregator that stores summands $\beta(q, a)$ for $a \in \Sigma$. In principle, this aggregator could be obtained by copying the aggregator for $\beta(q^\phi, \Sigma)$ and then modifying some summands (see line 9). However, aggregators are not constant-size data structures, so creating all of these slightly different aggregators would be expensive.

Instead, our strategy will be to use just a *single* aggregator, for the "current" state q, and make small modifications as we visit different states q'. More precisely, we have one aggregator γ per failure tree, first created at the tree's root. When we step backwards in the failure tree, say from q^ϕ to q, we modify "just a few" summands in γ so that $\beta(q, a)$ replaces $\beta(q^\phi, a)$ for $a \in \Sigma(q)$. This is fast if $\Sigma(q)$ is small. We can now obtain $\beta(q, \Sigma)$ as the aggregator's new total value. To visit other ancestors of q^ϕ, we must first move forward to q^ϕ again, which we do by reverting the modifications.[7]

Definition 11. *Aggregator γ **represents** $q \in Q$ if*
$$\gamma(a) = \beta(q, a), \ \forall a \in \Sigma$$
γ will be updated to represent different states in the failure tree at different times. When γ represents q, it holds that $\beta(q) = \rho(q) \oplus \gamma.\text{value}()$, by (10).

Updates are carried out by the methods in Alg. 5, which move backward and forward in a failure tree. When γ represents q^ϕ, we can call Visit(γ, q) to update γ so that it represents q. At any later time when γ again represents q, we can call Leave(γ, q) to undo this update, so that γ again represents q^ϕ.

Each Visit(γ, q) or Leave(γ, q) call runs in time $\mathcal{O}(|\Sigma(q)| \log |\Sigma|)$, since it sets $|\Sigma(q)|$ values in γ.[8]

---

called in Alg. 4, the memo (for that particular a) may not yet exist and will be filled in on demand.

[6]Clearly $\sum_{q \in Q} |\hat{\Sigma}(q)| \leq |\Sigma||Q|$ since $|\hat{\Sigma}(q)| \leq |\Sigma|$. Also, $\sum_{q \in Q} |\hat{\Sigma}(q)| \leq \sum_{q \in Q} \sum_{q' \preceq q} |\Sigma(q')| = \sum_{q' \in Q} \sum_{q \succeq q'} |\Sigma(q')| \leq \sum_{q' \in Q} |\boldsymbol{\pi}_{\max}||\Sigma(q')| = |\boldsymbol{\pi}_{\max}| \sum_{q' \in Q} |\Sigma(q')| = |\boldsymbol{\pi}_{\max}| s|\Sigma||Q|$.

[7]Any data structure can support a undo method. Any update method begins by pushing a sentinel onto an *undo stack*; whenever it writes a new value to a memory cell, it pushes an operation that would write the old value to that memory cell. undo(n) simply pops and applies these operations in reverse order, until it has popped n sentinels. Reverting updates in this way takes no more time than the original updates.

[8]A tighter analysis allows us to reduce this runtime to

Algorithm 5

1: **def** Visit(γ, q): ▷ *update γ that represented q^ϕ to represent q*
2:    **for** $a \in \Sigma(q)$:
3:      γ.set$(a, \beta(q, a))$ ▷ *Use the memoizing $\beta(q, a)$ from Alg. 4*
4: **def** Leave(γ, q): ▷ *update γ that represented q to represent q^ϕ*
5:    undo$(|\Sigma(q)|)$ ▷ *revert all the updates made by Visit*

Note that Visit(γ, q) accomplishes the same goal as Alg. 4 line 9, but with an extra runtime factor of $\mathcal{O}(\log|\Sigma|)$ to avoid subtraction. It may still be faster than the $\mathcal{O}(|\Sigma \setminus \Sigma(q)|)$ runtime of Alg. 3 line 10, when $|\Sigma(q)|$ is quite small relative to $|\Sigma|$.

6 A General Backward Algorithm for Acyclic WFSAs with Failure Transitions

Alg. 6 is our most general version of the backward algorithm for computing the pathsum of an acyclic WFSA-ϕ. It makes use of the Aggregator and pseudocode from the previous section.

Algorithm 6

1: **def** GeneralBackward$(\mathcal{A})$:
2:    **for** $q \in$ ReverseTopological$(\mathcal{A})$:
3:      $\mathcal{T} \leftarrow \mathcal{T}_q$ ▷ *failure tree containing q*
4:      **if** q has no fallback state : ▷ *q is root of failure tree*
5:        $\gamma_\mathcal{T} \leftarrow$ **new** Aggregator() ▷ *New empty aggregator*
6:        Visit$(\gamma_\mathcal{T}, q)$ ▷ *Initialize $\gamma_\mathcal{T}$*
7:        $q_\mathcal{T} = q$ ▷ *Remember the state represented by $\gamma_\mathcal{T}$*
8:      **while** $q_\mathcal{T}$ is not a descendant of q in $\mathcal{T}$:
9:        Leave$(\gamma_\mathcal{T}, q_\mathcal{T})$; $q_\mathcal{T} \leftarrow q_\mathcal{T}^\phi$ ▷ *Descend in $\mathcal{T}$*
10:      Visit$^+(\gamma_\mathcal{T}, q, q_\mathcal{T})$; $q_\mathcal{T} \leftarrow q$ ▷ *Ascend in $\mathcal{T}$*
11:      ▷ *Now $\gamma_\mathcal{T}$ represents q (thanks to all of the above)*
12:      $\beta(q) \leftarrow \rho(q) \oplus \gamma_\mathcal{T}$.value()
13:    **return** $\bigoplus_{q \in Q} \lambda(q) \otimes \beta(q)$
14: **def** Visit$^+(\gamma, q, q')$: ▷ *update γ that represented q' to represent q*
15:    **if** $q^\phi \neq q'$: Visit$^+(\gamma, q^\phi, q')$
16:    Visit(γ, q)

Like Alg. 3, this computes $\beta(q)$ at all states in reverse topological order. However, it attempts to share work among states q in the same failure tree $\mathcal{T}$, by having them share an aggregator $\gamma_\mathcal{T}$ that currently represents some state $q_\mathcal{T} \in \mathcal{T}$ (in the sense of definition 11). The algorithm updates the aggregator to represent q, by descending in the failure tree until it reaches a common descendant, and then ascending again until it reaches q.

$\mathcal{O}(\log \max_{q \in Q} |\overline{\Sigma}(q)|)$; see App. C for details.

To make line 8 efficient, we preprocess each failure tree by visiting its states in depth-first order and annotating each state with the time interval during which it is on the stack.[9] The loop at line 8 continues until the $q_\mathcal{T}$ interval contains the q interval.

6.1 Runtime

As in Alg. 4 (see §4.1), $\mathcal{O}(|E|)$ runtime is needed to sum over the non-failure transitions from each state. The rest of the runtime is dominated by the calls to Visit and Leave. Recall from §5 that visiting or leaving q takes time $\mathcal{O}(|\Sigma(q)| \log|\Sigma|)$. Since a state can be left at most once for each time it is visited, it suffices to count just the visits.

The number of visits to each state depends on the (reverse) topological order used at line 2. In the *best case*, q iterates over the states of each failure tree in depth-first order, starting at the root. Then Visit is called only on the current iterate q—either as a root (line 6) or as a parent (line 16). Since each state is Visited exactly once, the total runtime is $\mathcal{O}(|E| + \sum_{q \in Q} |\Sigma(q)| \log|\Sigma|)$. In the *worst case*, however, each q at line 2 is far in the failure tree from the previous one, forcing $q_\mathcal{T}$ to descend all the way to the root and then ascend again to q. This means line 16 visits all states q' for which $q \preceq q'$. The total runtime is therefore $\mathcal{O}(|E| + \sum_{q' \in Q} |\Sigma(q')| \operatorname{ancs}(q') \log|\Sigma|)$, where $\operatorname{ancs}(q') \stackrel{\text{def}}{=} |\{q : q \preceq q'\}|$ is the number of ancestors of q' in the failure tree. Renaming the summation variable, we get $\mathcal{O}(|E| + \sum_{q \in Q} |\Sigma(q)| \operatorname{ancs}(q) \log|\Sigma|)$.[10]

We can get a simpler but looser worst-case bound by increasing $\operatorname{ancs}(q)$ to $|\mathcal{T}_{\max}|$, the maximum size of any failure tree. Rewriting this in terms of s, we have bounded the runtime by $\mathcal{O}(|E| + s|\Sigma||Q||\mathcal{T}_{\max}|\log|\Sigma|)$, where, however, in the best case we avoid the $|\mathcal{T}_{\max}|$ factor.

The worst-case behavior is illustrated by Fig. 2, where the only possible topological order is $1, 2, 3, 4, 5, \ldots$. When line 2 iterates over state 5 immediately after state 4, the aggregator must transition $4 \xrightarrow{\text{Leave}} 2 \xrightarrow{\text{Leave}} 1 \xrightarrow{\text{Visit}} 3 \xrightarrow{\text{Visit}} 5$. Note that this involves 2 Visits, as 2 is the height of state 5.

---

[9] During depth-first search, the "clock" is a counter that starts at 0 and advances by 1 on each recursive call or return. Thus, the "time interval" is a pair of small integers.

[10] This can be slightly improved by noting that line 16 never Visits the root of the failure tree, so the $\sum_{q \in Q}$ can omit the root. (Thus, the runtime is $\mathcal{O}(|E|)$ on a WFSA without failure arcs.) Although the root is still Visited once at line 6, the cost of that visit can be folded into the $|E|$ term, since the initial creation of the aggregator at the root state q can be accomplished in time only $\mathcal{O}(|\Sigma(q)|)$ without the log factor (see App. C).
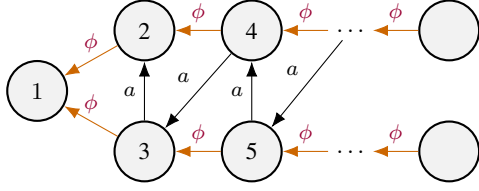
Figure 2: A WFSA-ϕ fragment in which Alg. 6 would perform a large number of updates over the ϕ-transitions.

If the a arcs were not present in Fig. 2, however, then $1, 2, 4, \ldots, 3, 5, \ldots$ would also be a topological order, which achieves the best-case behavior of visiting each state only once. Indeed, many topological orders would be available—some more efficient than others.

6.2 Topological Sorting Heuristics

It is desirable to choose a good topological order when one is available. In particular, the "best-case" scenario above is achieved under this condition:

Definition 12. *Let $\mathcal{A}$ be an acyclic WFSA-ϕ. Given a reverse topological order of the states, we say that q **completely precedes** q' if q and all its failure-tree ancestors precede q' and all its failure-tree ancestors. We say that the order is **compatible** with the failure trees of $\mathcal{A}$ if whenever q, q' are in the same failure tree*[11] *but have disjoint sets of ancestors, either q completely precedes q' or vice-versa.*

To put this another way, a compatible order of the WFSA states may jump back and forth among failure trees, as needed to achieve a topological ordering, but each failure tree's states will appear in some depth-first order starting at the tree's root, which ensures that each state is Visited just once.

In some backoff architectures such as variable-order conditional random fields (Vieira et al., 2016), it is easy to find a compatible order. In these WFSAs, each failure tree is associated with a position in a fixed input sentence. Simply visit the failure trees from right to left, enumerating each one's states in depth-first order starting at the root.

For the general case, we have developed an topological sorting algorithm that will find a compatible order when one exists. We begin with Kahn's (1962) agenda-based algorithm for finding a reverse topological order. It places all states onto a very simple priority queue in which "ready" states are prioritized at the front of the queue. The next state q to enumerate is obtained by popping this queue, and then the parents of q (that is, its immediate predecessors in the WFSA graph) decrement their counts of unenumerated children (i.e., immediate successors). If a parent state's count reaches 0, then it becomes ready and moves to the front portion of the queue. If the algorithm ever pops a non-ready state, then it throws an exception saying that the WFSA was cyclic.

Our approach is to modify Kahn's algorithm so as to break ties. Once q's children have been enumerated, Kahn's algorithm is allowed to enumerate q at any time, but our modified version prefers to wait until it would be possible to enumerate q and (eventually) its failure-tree ancestors with a single Visit each. Unfortunately, this test is expensive,[12] so using it would not actually speed up Alg. 6. We therefore omit the details here.

In practice, we recommend using a greedy version of the above algorithm. We do wait to enumerate q until q can be enumerated with a single Visit, but we no longer worry about its ancestors. This greedy heuristic is still guaranteed to find a compatible order if the WFSA has the special property that there are no paths between states in the same failure tree (other than ϕ-paths). Variable-order CRFs do have this property. Fig. 2 does not.

Specifically, we say that a not-yet-enumerated state $q \in \mathcal{T}$ is **cheap** if it is a ϕ-parent of the current $q_{\mathcal{T}}$ (that is, $q^{\phi} = q_{\mathcal{T}}$), so that Alg. 6 only has to call $\text{Visit}(\gamma_{\mathcal{T}}, q)$ to update $q_{\mathcal{T}} \leftarrow q$. Modify Kahn's algorithm to prioritize cheap ready states ahead of expensive ready states.[13] Modify Alg. 6 to repeatedly descend at the end of the main loop until $q_{\mathcal{T}}$ has at least one unenumerated ϕ-parent,[14] ensuring that there is a new cheap state in $\mathcal{T}$. The hope is that this cheap state will become ready while it is still cheap (indeed, it may already be ready).

6.3 Copying Aggregators

Long Leave–Visit paths can trigger many updates to aggregator $\gamma_{\mathcal{T}}$ due to Such paths can be shortened by splitting the failure tree into multiple smaller trees, each with its own aggregator. When we Visit a

---

[11]Remark: If we dropped the condition that q, q' be in the same failure tree, then we would get a stronger compability criterion that would let all failure trees share a single aggregator.

[12]It must look ahead to determine whether any failure-tree ancestor of q would have to wait for a state in some other part of the same failure tree. As far as we know, this requires computing the transitive closure of the WFSA.

[13]Each state (in particular $q_{\mathcal{T}}$) maintains its own agenda of unenumerated ϕ-parents, with the ready ones at the front.

[14]We may descend all the way to the root of the failure tree without ever achieving this condition—but this implies that the failure tree has been fully enumerated, so there is no need to further update $\gamma_{\mathcal{T}}$ and $q_{\mathcal{T}}$, as they will not be used again.

state q, we can choose to *copy* the aggregator from q^ϕ and update only the copy, leaving the old aggregator at q^ϕ. While this incurs a one-time copying cost, we can now split off the failure subtree rooted at q into its own failure tree. Enumerating states in this subtree will now never require visiting q's descendants. The effect is to reduce $|\mathcal{T}_{\max}|$ in the analysis of §6.1. App. E presents

- a dynamic splitting heuristic that is sensitive to the actual toposort order (§6.2)
- an static splitting algorithm that uses dynamic programming to choose the *optimal set* of split states to minimize a worst-case bound
- runtime analysis of an idealized case to show how Alg. 6 uses copying to gracefully degrade into Alg. 3 as the WFSA becomes denser

6.4 The Ring Case

In the case of a ring, it is possible to implement a faster aggregator. The aggregator still stores N summands and their total, but no partial sums. It can replace a summand in time $O(1)$ rather than $O(\log N)$, by subtracting off the old summand from the total and adding the new one. This eliminates the $\log|\Sigma|$ factor from the runtimes in §6.1.

The resulting bound $\mathcal{O}(|E| + s|\Sigma||Q||\mathcal{T}_{\max}|)$ for Alg. 6 is still worse than §4.1's bound of $\mathcal{O}(|E| + |\Sigma||Q|\min(1, s|\boldsymbol{\pi}_{\max}|))$ for Alg. 4. However, the former becomes better when a compatible order is known and the $|\mathcal{T}_{\max}|$ can be dropped.

It is more instructive to compare the tighter bounds of $\mathcal{O}(|E| + \sum_{q'\in Q}|\Sigma(q')|\operatorname{ancs}(q'))$ for Alg. 6 and $\mathcal{O}\big(|E| + \sum_{q\in Q}|\hat{\Sigma}(q)|\big)$ for Alg. 4. If a compatible order is known, $\operatorname{ancs}(q')$ can be dropped and the former is better. If not, then either runtime could be better. The former effectively charges each state q for all of the out-symbols at all of its ϕ-descendants q',[15] while the latter charges q for all of the *distinct* out-symbols at its *ϕ-ancestors* q'. The reason for the difference: Both algorithms override a descendant's a-arc with an ancestor's a-arc, but to find these descendant-ancestor pairs, Alg. 6 loops over a-arcs at the descendant (pushing subtrahends up from below) while Alg. 4 loops over a-arcs at the ancestor (pulling subtrahends up from above). When different descendant-ancestor paths overlap, the former algorithm shares work between them if the Visit order is good, while the latter shares work between them via memoization.

---

[15]Since $\sum_{q'\in Q}|\Sigma(q')|\operatorname{ancs}(q') = \sum_{q'\in Q}\sum_{q\leq q'}|\Sigma(q')| = \sum_{q\in Q}\sum_{q'\geq q}|\Sigma(q')|$.

Alg.	$\mathcal{O}$-Cost	Use case
Alg. 1	$(\bar{s}-s)|Q||\Sigma||Q|$	never
Alg. 3	$(\bar{s}-s)|\Sigma||Q|$	–
Alg. 4	$|\boldsymbol{\pi}_{\max}|s|\Sigma||Q|$	$s \ll \frac{\bar{s}-s}{|\boldsymbol{\pi}_{\max}|}$
Alg. 6$^+$	$C_U s|\Sigma||Q|$	$s \ll \frac{\bar{s}-s}{C_U}$
Alg. 6$^-$	$|\mathcal{T}_{\max}|C_U s|\Sigma||Q|$	$s \ll \frac{\bar{s}-s}{|\mathcal{T}_{\max}|C_U}$
App. E	$\sqrt{C_U}s|\Sigma||Q|$	$s \ll \frac{(\bar{s}-s)^2}{C_U}$

Table 1: Runtime of computing the failure term by the different algorithms. The "use case" column indicates when an algorithm has better complexity than the baseline algorithm, Alg. 3. C_U is the update complexity of the aggregator interface: $\log|\Sigma|$ in the general case (via a Fenwick tree) and 1 in the ring case of §6.4 (via subtraction). Alg. 6$^+$ is the runtime for WFSA-ϕ's such as VoCRFs where a compatible state order is known, whereas Alg. 6$^-$ is the general worst-case runtime.

7 Comparison of Algorithms

This work proposed multiple algorithms for computing the pathsum of an acyclic WFSA-ϕ. They are all alternatives to running the backward algorithm (Alg. 1)—or its simple improvement by aggregation (Alg. 3)—after explicitly expanding failure transitions (Alg. 2). This section summarizes the improvements.

As mentioned in §3.2, we never change the way the local component of the backward values is computed. All algorithms we consider therefore retain the $\mathcal{O}(|E|)$ complexity term from expanding the non-ϕ transitions. What differs is the method for computing the failure term $\beta(q, \Sigma\smallsetminus\Sigma(q))$—the contribution of the paths starting at q that take q's failure transition. Table 1 compares this term's runtime complexity for all the algorithms discussed.

Maintaining perspective, the benefits of our more sophisticated pathsum algorithms over the basic Alg. 3 only make an actual impact if Alg. 3's failure component complexity $\mathcal{O}((\bar{s}-s)|\Sigma||Q|)$ is dominant over the local component $\mathcal{O}(|E|)$, where $|E| \geq s|\Sigma||Q|$. In particular, reducing the failure component is only helpful if $\bar{s} \gg s$, so that expanding failure transitions would make the graph denser.

8 Conclusion

We presented two new algorithms for more efficiently computing the backward values and pathsum of a sparse acyclic semiring-weighted FSA with ϕ-transitions, using the observation that a ϕ-transition from q to q^ϕ means that $\beta(q)$ is a sparsely modified version of $\beta(q^\phi)$. We characterized when the new algorithms are asymptotically faster.

Limitations

This section addresses two main limitations of our work: the assumptions made on the structure of the WFSAs and the applicability of the proposed algorithms in real scenarios.

Acyclicity assumption. We only consider acyclic WFSAs. While this covers interesting use cases such as CRFs, other commonly used instances of WFSAs also contain cycles, e.g., n-gram language models. Furthermore, all our novel algorithms actually assume that $E \cup E^\phi$ is acyclic, whereas failure expansion only requires that the resulting $\overline{E}$ is acyclic. The former is a strictly stronger condition—see Fig. 5 below for an example WFSA-ϕ where $E \cup E^\phi$ is not acyclic, but $\overline{E}$ is.

Applicability. As seen above, the runtime of Alg. 6 depends on the size of failure trees, with complexity $\mathcal{O}(|E| + s|\Sigma||Q||\mathcal{T}_{\max}| \log |\Sigma|)$. In practice, failure trees may be large, or s may be large, which could result in our algorithms performing worse than the naïve approaches.[16] To see this, consider higher-order CRFs with backoff, a useful formalism for sequence tagging in NLP (Vieira et al., 2016), which can be encoded as WFSAs. They were the initial motivation for our proposed algorithms. Although these backoff CRFs do admit a compatible topological order that allows us to avoid the $|\mathcal{T}_{\max}|$ factor (§6.2), we inspect them as an example of how large $|\mathcal{T}_{\max}|$ can be.

An order-n CRF tagging a sequence of length ℓ can be represented with a WFSA-ϕ in form of a lattice of ℓ layers. The layers include *tag sequences* of length $\leq n$, meaning that, given a set of tags Σ, each layer contains states representing histories $h \in \{\epsilon\} \cup \Sigma \cup \cdots \cup \Sigma^n$. This results in $\mathcal{O}(|\Sigma|^n)$ states per layer. Backoff transitions in such models encode transitions to lower-order histories (transitioning from a history of length k to one of length $k - 1$) whenever a transition to a history of the same order is not possible. It is easy to see that each history of order k could have up to Σ incoming ϕ-transitions, connecting it to a large failure tree, which is exponential in size w.r.t. n.

Ethics Statement

We are not aware of any specific social risks created or exacerbated by this work.

---

[16]The hybrid approach proposed in App. E aims at improving this shortcoming but requires additional knowledge of the WFSA's topology.

References

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Communications of the Association for Computing Machinery*, 18(6).

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2003. Generalized algorithms for constructing statistical language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Uri Alon, Frank Xu, Junxian He, Sudipta Sengupta, Dan Roth, and Graham Neubig. 2022. Neuro-symbolic language modeling with automaton-augmented retrieval. In *Proceedings of the International Conference on Machine Learning*.

Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jenifer C. Lai, and Robert L. Mercer. 1992. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4).

David Chiang and Peter Cholak. 2022. Overcoming a theoretical limitation of self-attention. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3).

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8.

Jan Hajič and Barbora Hladká. 1998. Tagging inflective languages: Prediction of morphological categories for a rich structured tagset. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and International Conference on Computational Linguistics*.

John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. 2020. RNNs can generate bounded hierarchical languages with optimal memory. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Chengyue Jiang, Zijian Jin, and Kewei Tu. 2021. Neuralizing regular expressions for slot filling. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Arthur B. Kahn. 1962. Topological sorting of large networks. *Commmunications of the ACM*, 5(11):558–562.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine Learning*, pages 282–289.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1).

Chu-Cheng Lin, Hao Zhu, Matthew R. Gormley, and Jason Eisner. 2019. Neural finite-state transducers: Beyond rational relations. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Proceedings of the International Conference on Machine Learning*, pages 591—598.

Mehryar Mohri. 2002a. Generic ε-removal and input ε-normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(1):129–143.

Mehryar Mohri. 2002b. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3).

Hao Peng, Roy Schwartz, Sam Thomson, and Noah A. Smith. 2018. Rational recurrences. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Pushpendre Rastogi, Ryan Cotterell, and Jason Eisner. 2016. Weighting finite-state transductions with neural context. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Shruti Rijhwani, Daisy Rosenblum, Antonios Anastasopoulos, and Graham Neubig. 2021. Lexically aware semi-supervised learning for OCR post-correction. *Transactions of the Association for Computational Linguistics*, 9.

Sunita Sarawagi and William W. Cohen. 2004. Semi-Markov conditional random fields for information extraction. In *Advances in Neural Information Processing Systems*, volume 17.

Roy Schwartz, Sam Thomson, and Noah A. Smith. 2018. Bridging CNNs, RNNs, and weighted finite-state machines. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. 2019. Ordered neurons: Integrating tree structures into recurrent neural networks. In *International Conference on Learning Representations*.
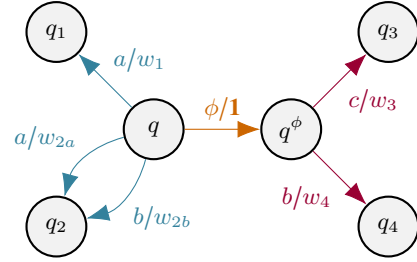
(a) A fragment inside of a WFSA-ϕ graph.
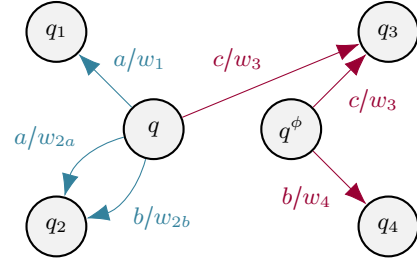
(b) Failure-expanded version of the fragment from Fig. 3a.

Figure 3: WFSA-ϕ and WFSA examples discussed in App. A.

Tim Vieira, Ryan Cotterell, and Jason Eisner. 2016. Speed-accuracy tradeoffs in tagging with variable-order CRFs and structured sparsity. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

A Algorithm Demonstrations

Consider the WFSA-ϕ fragment in Fig. 3a and the version in Fig. 3b that is produced by failure expansion (Alg. 2). This section demonstrates how different algorithms we discuss operate to compute the value $\beta(q)$.

The normal backward algorithm (Alg. 1) on the failure-expanded version would compute $\beta(q)$ as

$$\beta(q) \leftarrow w_1 \otimes \beta(q_1) \oplus w_{2a} \otimes \beta(q_2)$$
$$\oplus w_{2b} \otimes \beta(q_2) \oplus w_3 \otimes \beta(q_3).$$

The version that memoizes out-symbol sums (Alg. 3) would compute $\beta(q)$ as

$$\beta(q) \leftarrow w_1 \otimes \beta(q_1) \oplus w_{2a} \otimes \beta(q_2)$$
$$\oplus w_{2b} \otimes \beta(q_2) \oplus \beta(q^\phi, c).$$

Alg. 3 is equivalent to copying the entire $\beta(q^\phi, a)$ memo table from q^ϕ, modifying the values for $a \in \Sigma(q)$, and summing. That is, the dictionary $\{c \mapsto w_3 \otimes \beta(q_3), b \mapsto w_4 \otimes \beta(q_4)\}$ would be passed back from q^ϕ to q and updated there to $\{c \mapsto w_3 \otimes \beta(q_3), b \mapsto w_{2a} \otimes \beta(q_2) \oplus w_{2b} \otimes \beta(q_2), a \mapsto w_1 \otimes \beta(q_1)\}$, and $\beta(q, \Sigma)$ would be found by summing the values in this dictionary.

The subtraction-based algorithm (Alg. 4) would compute $\beta(q)$ as

$$\beta(q) \leftarrow w_1 \otimes \beta(q_1) \oplus w_{2a} \otimes \beta(q_2)$$
$$\oplus w_{2b} \otimes \beta(q_2) \oplus \beta(q^\phi) \ominus \beta(q^\phi, \{a, b\})$$
$$= w_1 \otimes \beta(q_1) \oplus w_{2a} \otimes \beta(q_2)$$
$$\oplus w_{2b} \otimes \beta(q_2) \oplus \beta(q^\phi) \ominus \beta(q^\phi, \{b\}).$$

Lastly, Alg. 6 would initialize an aggregator γ at the failure tree root q^ϕ as $\{c \mapsto w_3 \otimes \beta(q_3), b \mapsto w_4 \otimes \beta(q_4)\}$, and pass the aggregator back to q. There, γ would be updated via $\gamma.\mathsf{set}(a, \dots)$ and $\gamma.\mathsf{set}(b, \dots)$ to $\{c \mapsto w_3 \otimes \beta(q_3), b \mapsto w_{2a} \otimes \beta(q_2) \oplus w_{2b} \otimes \beta(q_2), a \mapsto w_1 \otimes \beta(q_1)\}$, causing $\gamma.\mathsf{value}()$ to change. Then, $\beta(q)$ would be computed as $\beta(q) = \gamma.\mathsf{value}()$. Compare this to Alg. 3, which had to explicitly sum up all the values in this dictionary to compute $\beta(q)$, since it did not use an aggregator data structure (App. C) to maintain partial sums over subsets of these values.

B Number of Transitions Added by Failure Expansion

We show §3.1's claim that the number of transitions added by failure expansion (Alg. 2) is $(\overline{s} - s)|\Sigma||Q|$ when the input WFSA-ϕ is deterministic.

In the deterministic case, each out-symbol at a state labels exactly one outgoing transition. Hence the number of added transitions for a given state q equals the number of added out-symbols, $|\overline{\Sigma}(q) \smallsetminus \Sigma(q)| = |\overline{\Sigma}(q)| - |\Sigma(q)|$, where $\overline{\Sigma}(q) \supseteq \Sigma(q)$. Summing over all $q \in Q$, and using definition 9, we get a total number of added transitions of

$$\sum_{q \in Q} \overline{\Sigma}(q) - \sum_{q \in Q} \Sigma(q) = \overline{s}|\Sigma||Q| - s|\Sigma||Q|$$
$$= (\overline{s} - s)|\Sigma||Q|.$$

In the general case where the input WFSA-ϕ may be non-deterministic, each added out-symbol may label anywhere from 1 to $|Q|$ added transitions. Thus the total number of added transitions is between $(\overline{s} - s)|\Sigma||Q|$ and $(\overline{s} - s)|\Sigma||Q|^2$.

C Aggregator Implementation

A Fenwick tree (Fenwick, 1994) is a data structure that stores a sequence $v_1, \dots, v_N$ and can efficiently return any prefix sum of the form $\bigoplus_{n=1}^{N'} v_n$ for $N' \in [0, N]$, as well as allowing the individual elements v_n to be updated. Each prefix-sum query or element update takes $\mathcal{O}(\log N)$ time.

Our aggregator interface in §5 is simpler. It only queries the full sum $\bigoplus_{n=1}^{N} v_n$ (the case $N' = N$). Thus, the order of the elements is not considered by this interface. §6.4 noted that in the special
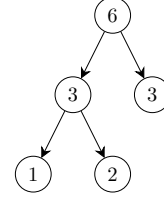

Figure 4: A Fenwick tree computing $1 + 2 + 3 = 3 + 3 = 6$.

case where subtraction is available (and numerically stable), an aggregator can be implemented even more efficiently *without* a Fenwick tree, since then it is easy to update the sum in constant time when updating any element. However, subtraction is not guaranteed to be available for arbitrary $\oplus$ operations (e.g., $\oplus = \max$).

A Fenwick tree stores the elements v_n at the leaves of a balanced binary tree. Each internal (non-leaf) node stores the $\oplus$-sum of the values stored at its children. As a result, thanks to the associativity of $\oplus$, the root of the tree contains the full sum $\bigoplus_{n=1}^{N} v_n$, which can be looked up in $\mathcal{O}(1)$ time. An example of a Fenwick tree (in the real semiring) is presented in Fig. 4. Note that we draw the root of a Fenwick tree at the top and consider it to be the ancestor of all other nodes, whereas failure trees had the root as the descendant of all other states.

Initial creation of the Fenwick tree takes only $\mathcal{O}(N)$ total time by visiting all nodes in bottom-up order and setting each non-leaf node to the $\oplus$-sum of its children. When a leaf v_n is updated, just its ancestors are recomputed, again in bottom-up order. As there are about $\log N$ ancestors, this update takes $\mathcal{O}(\log N)$ total time.

Our aggregator is a Fenwick tree that stores $N = |\Sigma|$ elements, where v_n is the value associated with the nth element of Σ. (That is, we identify the possible keys $a \in \Sigma$ with the integers $[1, N]$.) Initially, $v_n = \mathbf{0}$, but may be changed by set. Each call to set takes $\mathcal{O}(\log |\Sigma|)$ time; this factor appears in our runtime analysis. To achieve our runtime bounds for sparse WFSAs, we must take care not to spend $\mathcal{O}(|\Sigma|)$ time initializing all of the leaves and internal nodes to $\mathbf{0}$ every time we create an aggregator. Array initialization overhead can always be avoided, using a method from computer science folklore (Aho et al., 1974, exercise 2.12).

Alternatively, we can store values in the Fenwick tree only for those keys for which values have been set. Under this design, the operation $\mathsf{set}(a : \Sigma, v : \mathbb{K})$ must update $v_n \leftarrow v$ where n is the integer index associated with key a. To find n, the aggregator maintains a hash table that maps

keys to consecutive integers. We assume $\mathcal{O}(1)$-time hash operations. The first key that is set is mapped to 1, the second is mapped to 2, etc. When a key a is set for the first time—that is, when it is not found in the hash table—N is incremented, the mapping $a \mapsto N$ is added to the hash table, and $v_N = v$ is appended to the Fenwick sequence. The hash table is also consulted by the get operation.

In our application, for an aggregator that represents state q, the keys that have been set are $\overline{\Sigma}(q)$. The design in the previous paragraph therefore reduces N from $|\Sigma|$ to $N = |\overline{\Sigma}(q)|$. As a result, the factor $\mathcal{O}(\log|\Sigma|)$ in our analysis could actually be reduced to $\mathcal{O}(\log \max_{q \in Q} |\overline{\Sigma}(q)|)$.

Note that to obtain this runtime reduction, the undo method must properly undo the changes not only to the Fenwick tree but to the integerizing hash table (see footnote 7). If a call to set in Visit incremented N and added $a \mapsto N$, then the call to undo in Leave must remove $a \mapsto N$ and decrement N, thereby keeping N small as desired.

D Weighted ϕ-Transitions

Throughout the main paper, we assumed that all ϕ-transitions have a weight of $\mathbf{1}$. This simplifying assumption is typically violated by backoff models (e.g., Allauzen et al., 2003). Fortunately, it can be removed with relatively small changes to our equations, algorithms and data structures.

Most simply, a weighted failure transition $q \xrightarrow{\phi/w^\phi} q^\phi$ could be simulated by a path $q \xrightarrow{\phi/\mathbf{1}} q^\varepsilon \xrightarrow{\varepsilon/w^\phi} q^\phi$ where q^ε is a newly introduced intermediate state with only an ε-transition. We would then have to eliminate the ε-transition as mentioned in §2. In this case, this simply means replacing $q^\varepsilon \xrightarrow{\varepsilon/w^\phi} q^\phi$ in E with transitions $\{q^\varepsilon \xrightarrow{a/w^\phi \otimes w} q' : q^\phi \xrightarrow{a/w} q' \in E\}$. However, this may be expensive when the original fallback state q^ϕ has many outgoing transitions, which is typical in a backoff setting. Copying all of those transitions to a parent as in Alg. 2 (failure expansion) is exactly what the new methods in this paper are designed to avoid. We therefore give direct modifications to our constructions.

Suppose the failure transition for state q has weight w^ϕ—that is, E contains $q \xrightarrow{\phi/w^\phi} q^\phi$—where perhaps $w^\phi \neq \mathbf{1}$. Then the second case of Eq. (9) should be modified to set

$$\beta(q, a) = w^\phi \otimes \beta(q^\phi, a)$$

for any $a \notin \Sigma(q)$. Similarly, w^ϕ should be incorporated into Eq. (13), which becomes

$$\beta(q, \Sigma \smallsetminus \Sigma(q)) = w^\phi \otimes \bigoplus_{b \in \Sigma \smallsetminus \Sigma(q)} \beta(q^\phi, b)$$

Finally, the subtraction expression in the right-hand side of Eq. (14) must be left-multiplied by w^ϕ.

In Alg. 2, which constructs the failure-expanded edge set, the update at state q becomes

$$\overline{E} \leftarrow \overline{E} \cup \{q \xrightarrow{a/w^\phi \otimes w} q' \mid q^\phi \xrightarrow{a/w} q' \in \overline{E}, a \notin \Sigma(q)\}$$

Algs. 3 and 4 undergo straightforward modifications based on the modified Eqs. (9), (13) and (14). When $\beta(q^\phi, b)$ is copied backwards over a ϕ-transition $q \xrightarrow{\phi/w^\phi} q^\phi$, it must be left-multiplied by w^ϕ to yield $\beta(q, b)$. This affects Alg. 3 lines 8–9 and Alg. 4 line 12, as well as the purple terms in Alg. 4 line 9. These modifications do not affect the asymptotic runtime complexity.

Alg. 6 requires more modification. We must extend our aggregator class (§5) with a new method that left-multiplies all elements by a constant:[17]

1: **class** Aggregator():   ▷ *We use γ to refer to an aggregator instance*
        ⋮
6:      **def** mult(m: $\mathbb{K}$)   ▷ *$\forall a \in \Sigma$, updates $\gamma(a) \leftarrow m \otimes \gamma(a)$*

In Alg. 5, Visit(γ, q) should begin by calling mult(w^ϕ) where w^ϕ is the weight of the failure arc from q. Consequently, Leave(γ, q) should be modified to undo one more update than before.

How to implement the mult method efficiently?

With both subtraction and division The subtraction-based aggregator (§6.4) can be modified to still support all operations in $\mathcal{O}(1)$ time, provided that the ring $\mathbb{K}$ is actually a divsion ring (non-commutative field), i.e., it supports division by non-$\mathbf{0}$ multipliers. The aggregator maintains an overall multiplier M, initially $\mathbf{1}$, and the call mult(m) replaces M with $m \otimes M$; thus, M is a product of the I multipliers applied far, $m_I \otimes \cdots \otimes m_1$. As in App. C, we identify each key with an integer index n. If a has index n, then set(a, v) stores $M^{-1} \otimes v$ into v_n.[18] Later get(a) can return $M \otimes v_n$; since M

---

[17]This extension can more generally support the case where the multipliers m fall in a monoid of scalars $(\mathbb{M}, \otimes, \mathbf{1})$ that acts on a monoid $(\mathbb{K}, \oplus, \mathbf{0})$ of values v (perhaps vectors). However, for our WFSA-ϕ application, both the scalars and the values are weights from the same semiring: m will always be a failure weight $w^\phi \in \mathbb{K}$.

[18]In the special case where $M = \mathbf{0}$, so that M^{-1} does not exist, an arbitrary value may be stored, since it will just be multiplied by $\mathbf{0}$ later.

has been updated in the meantime, this yields the originally set value v left-multiplied by all *subsequent* multipliers, since M has been updated. The aggregator also maintains the total $\bigoplus_{n=1}^{N} v_n$ as v_n values are set or replaced (using subtraction), and the value method returns $M \otimes \bigoplus_{n=1}^{N} v_n$.

With subtraction only If $\mathbb{K}$ does not support division, then the subtraction-based aggregator can be rescued as follows. The aggregator maintains the number I of multipliers applied so far, as well as their product M as before. The function $\mathsf{set}(a, v)$ now stores v into v_n and I into i_n, and later $\mathsf{get}(a)$ returns $M_{i_n} \otimes v_n$, where in general M_i is defined to be the product of multipliers *subsequent* to m_i, that is, $M_i = M_I \otimes \cdots \otimes m_{i+1}$. The aggregator maintains the current total S that should be returned by lookup; the $\mathsf{mult}(m)$ method left-multiplies this total by m, while the method $\mathsf{set}(a, v)$ modifies this total by adding $v \ominus \mathsf{get}(a)$ before it updates (v_n, i_n).[19] The difficulty is now in obtaining the partial products M_i without division. This can be done by maintaining $m_1, \ldots, m_I$ in a Fenwick tree.[20] This means that mult and get now take time $\mathcal{O}(\log I)$ rather than $\mathcal{O}(1)$. The effect on §6.4's ring-based version of Alg. 6 is to add $\sum_{q' \in Q} \operatorname{ancs}(q') \log |\boldsymbol{\pi}_{\max}| \leq |Q| |\mathcal{T}_{\max}| \log |\boldsymbol{\pi}_{\max}|$ to the asymptotic runtime expression. This is the same cost as if every state had $\log |\boldsymbol{\pi}_{\max}|$ additional outgoing symbols. $|\boldsymbol{\pi}_{\max}|$ is usually very small.

Without subtraction For this case, we stored the summands in a Fenwick tree (App. C). Fortunately, it is possible to extend that data structure to support mult in time $\mathcal{O}(N)$, where N is the number of elements, without affecting the asymptotic runtime of set, value, or undo. The asymptotic runtimes of Algs. 5–6 will remain unchanged.

In our modified Fenwick tree, the N leaves store *unscaled* values $u_1, \ldots, u_N \in \mathbb{K}$. Each node j (leaf or internal node) stores a multiplier m_j that will be lazily applied to all of the leaves that are descendants of j. Thus, the scaled value v_n is found as the product $m_r \otimes m_{j_1} \otimes m_{j_2} \otimes \cdots \otimes m_n \otimes u_n$, where $r, j_1, \ldots n$ is the path from the root r to the leaf n. Thus, the leaves store the elements v_n directly (as they would in an ordinary Fenwick tree) only

in the special case where all the multipliers are $\mathbf{1}$. In general v_n must be computed on demand. The runtime of get is now $\mathcal{O}(\log N)$ rather than $\mathcal{O}(1)$, but Algs. 5–6 never actually use the get method.

The new call $\mathsf{mult}(m)$ simply replaces $m_r \leftarrow m \otimes m_r$, which affects all v_n in $\mathcal{O}(1)$ total time.

To support fast computation of the total value $\bigoplus_{n=1}^{N} v_N$, we also store partial sums at the nodes, as before. Thus, each node j stores a pair (m_j, u_j). The *scaled value* of node j is defined to be $m_j \otimes u_j$. When j is an internal node, we ensure as an invariant that u_j is the sum of the scaled values of j's children, updating it whenever j's children change. The value method simply returns the scaled value of the root in $\mathcal{O}(1)$ time.

The interesting modification is to the set method. To set v_n to v, leaf n is modified to set $(m_n, u_n) \leftarrow (\mathbf{1}, v)$—but also, all of n's ancestors j must be modified to have multipliers $m_j = \mathbf{1}$, so that $v_n = \mathbf{1} \otimes \cdots \otimes \mathbf{1} \otimes v = v$ as desired. Before being set to $\mathbf{1}$, each old multiplier m_j is "pushed down" to its children so that it still affects all leaves of j. The method descends from the root r to leaf n: it pushes the m_j values out of the way on the way down, updates the leaf at the bottom, and restores the invariant by recomputing the u_j values on the way back up as it returns.

1: **def** $\mathsf{set}(a: \Sigma, v: \mathbb{K})$:
2:   $n \leftarrow$ leaf that stores value of key a
3:   $\mathsf{set\_desc}(n, v, r)$ $\quad \triangleright$ *r is the root of the Fenwick tree*
4: **def** $\mathsf{set\_desc}(n: \text{leaf}, v: \mathbb{K}, j: \text{node})$:
5:   $\triangleright$ *j is an ancestor of n; j's own proper ancestors have multiplier 1; so will j upon return*
6:   **if** j is a leaf : $(m_j, u_j) \leftarrow (\mathbf{1}, v)$ $\quad \triangleright$ *Since j = n*
7:   **else**
8:     **for** $k \in \mathsf{children}(j) : m_k \leftarrow m_j \otimes m_k$
9:     $m_j \leftarrow \mathbf{1}$ $\quad \triangleright$ m_j *has been pushed down*
10:    $\mathsf{set\_desc}(n, v, \text{child of } j \text{ that is anc. of } n)$
11:    $u_j \leftarrow \bigoplus_{k \in \mathsf{children}(j)} m_k \otimes u_k$ $\triangleright$ *Restore invariant at j*

The **else** clause in $\mathsf{set\_desc}$ can be rephrased (less readably) to avoid looping twice over $\mathsf{children}(j)$:

8:    $k \leftarrow$ the child of j that is an ancestor of n
9:    $m_k \leftarrow m_j \otimes m_k$ $\quad \triangleright$ *push m_j down to k*
10:   $\mathsf{set\_desc}(n, v, k)$
11:   $u_j \leftarrow u_k$ $\quad \triangleright = m_k \otimes u_k$, *since now $m_k = 1$*
12:   **for** $k' \in \mathsf{siblings}(k) :$ $\triangleright$ *in a binary tree, there will be ≤ 1*
13:     $m_{k'} \leftarrow m_j \otimes m_{k'}$ $\quad \triangleright$ *push m_j down to k'*
14:     $u_j \oplus= m_{k'} \otimes u_{k'}$
15:   $m_j \leftarrow \mathbf{1}$ $\triangleright$ m_j *has been pushed down and invariant restored at j*

---

[19]Although get is used internally within this implementation of set, that does not contradict the comment later in this section that get is never called directly by Algs. 5–6.

[20]A Fenwick tree supports suffix sums (or indeed, sums over any contiguous subsequence of elements) as efficiently as prefix sums. In our case, the elements in the Fenwick tree are multipliers, so the "sum" operation to be used is $\otimes$, not $\oplus$.

E Tree Splitting Details

Alg. 6 is applicable to any acyclic semiring-weighted WFSA-ϕ. However, updating an Aggregator as it travels within a failure tree incurs an additional worst-case multiplicative runtime factor of $|\mathcal{T}_{\max}|$, the size of the biggest failure tree. This section outlines an improvement by lessening this impact. We do so by *splitting* large failure trees into multiple smaller ones.

Alg. 6 destructively updates an aggregator γ when Visiting a state q from q^ϕ. This takes time $\mathcal{O}(|\Sigma(q)| \log |\Sigma|)$. In contrast, Alg. 3 can be thought of as non-destructively *copying* γ to q from q^ϕ, which means the work can be saved and does not have to be redone if q is re-Visited later.

This inspires us to hybridize Alg. 6 as follows: Visit(γ, q) in Alg. 5 may optionally copy-and-update γ rather than just updating it. Copying effectively cuts the transition $q \xrightarrow{\phi/w} q^\phi$, making the sub-tree rooted at q a new independent failure tree with its own Aggregator instance. Copy-and-update does incur a one-time cost of $\mathcal{O}(|\Sigma|)$,[21] but now Alg. 6 line 3 will select a smaller failure tree.

However, at what states (if any) should we split each failure tree? The optimal set of splits depends on the topological order (§6.2) used by Alg. 6.

Dynamic splitting heuristics A simple greedy heuristic would be to split at q upon any call Visit(q) where copy-and-update is estimated to be cheaper than destructive updating, based on the current size of the aggregator and the number of required updates $|\Sigma(q)|$. However, this does not consider the future benefit of having smaller failure trees, and it does not adapt to the topological order.

A more sophisticated dynamic heuristic is for Visit(q) to split at q if not doing so would cause the *total* time spent so far on all Visit(q) calls[22] to exceed the time that it would take to copy-and-update at q. (Put another way, it does so if it now realizes in retrospect that it would have been better for the very first Visit(q) call to have invested in copy-and-update.) This ensures that our enhanced Alg. 6 will take at most twice as long as Alg. 3,

which always does copy-and-update. It eventually splits any state that is Visited often enough by the chosen topological order, especially if that state is expensive to Visit. On the other hand, if the chosen topological order is compatible so that every state is Visited only once, it will still achieve or outperform the best-case behavior of the original Alg. 6.

Static splitting algorithms We may also consider static methods, which do not adapt to the topological order that is actually used, but optimize to mitigate the worst case. In the runtime analysis of §6.1, the failure tree $\mathcal{T}$ contributes $\mathcal{O}(f(\mathcal{T}) \log |\Sigma|)$ to the failure term in the worst-case runtime of Alg. 6,[23] where $f(\mathcal{T}) \stackrel{\text{def}}{=} \sum_{q \in \mathcal{T}} |\Sigma(q)| \operatorname{ancs}(q)$.[24] We may seek a split that is optimal with respect to this runtime bound. Let q_1 be the root of $\mathcal{T}$. Suppose we choose to copy-and-update the aggregator when we first visit each of $q_2, \ldots, q_K \in \mathcal{T}$, essentially cutting off each state q_k from its fallback state q_k^ϕ. (Here all of the q_k are to be distinct.) This splits $\mathcal{T}$ into trees $\mathcal{T}_1, \ldots, \mathcal{T}_K$, where each $\mathcal{T}_k$ is rooted at q_k. Then the contribution of these K trees to the asymptotic runtime upper bound is proportional to $(K-1)|\Sigma| + \sum_{k=1}^K f(\mathcal{T}_k) \log |\Sigma|$, where the first term covers the cost of the $K - 1$ copy-and-update operations, and where the factor $\operatorname{ancs}(q)$ in the definition of $f(\mathcal{T}_k)$ considers only the ancestors of q within $\mathcal{T}_k$. Our goal is to choose $K \geq 1$ and $q_2, \ldots, q_K$ to minimize this expression.

We first remark that requiring $K \leq 2$ makes it easy to solve the problem in time $\mathcal{O}(|\mathcal{T}|)$ time, assuming that we already know $|\Sigma(q)|$ for each $q \in \mathcal{T}$. Define $D_q = \sum_{q' > q} |\Sigma(q')|$, the total number of out-symbols at proper descendants of q. The improvement $f(T) - (f(T_1) + f(T_2))$ from splitting $\mathcal{T}$ at q_2 is simply $D_{q_2} \operatorname{ancs}(q_2)$. Intuitively, there are D_{q_2} out-symbols that can no longer be encountered when Visit$^+$ is called on any of the $\operatorname{ancs}(q_2)$ states in $\mathcal{T}_2$.[25] This yields an improvement of $-|\Sigma| + D_{q_2} \operatorname{ancs}(q_2) \log |\Sigma|$ in the runtime bound. A simple recursion from the root q_1 is enough to find D_q and $\operatorname{ancs}(q)$ at every state q, and thus find the state $q \neq q_1$ that achieves the best improvement in the runtime bound when chosen as q_2. If no choice achieves a positive improvement,

---

[21]When we copy-and-update, each modification takes only $\mathcal{O}(1)$ time, not $\mathcal{O}(\log |\Sigma|)$. The strategy is to copy all $\mathcal{O}(|\Sigma|)$ elements from γ of the Fenwick tree, update some of them, and only then build a new Fenwick tree from the updated elements, which takes only $\mathcal{O}(|\Sigma|)$ time: see App. C.

[22]Remark: For a given q, all such calls do exactly the same work and should take the same amount of time, regardless of other splits.

[23]Versus $\mathcal{O}(\sum_{q \in \mathcal{T}} |\overline{\Sigma}(q)| - |\Sigma(q)|)$ in the case of Alg. 3.

[24]As footnote 10 notes, this bound may be improved by defining f to sum over only the *non-root* states $q \in \mathcal{T}$.

[25]When f is defined using the tighter bound of footnotes 10 and 24, redefine $D_q = \sum_{q_1 > q' \geq q} |\Sigma(q')|$, so that D_{q_2} will now include q_2 but exclude q_1.

then we do not split the tree and leave $K = 1$.

We now present an exact algorithm for the full problem, with no bound on K. Roughly speaking, after we split at a state q' (making it the root of its own failure tree), we will also consider splitting again at its ancestors q, but we do not make these decisions greedily—we use dynamic programming. The main observation is that if q is currently in a failure tree with root $q' > q$ (where either $q' = q_1$ or we previously split at q'), then splitting at q will give a further improvement of $-|\Sigma| + (D_q - D_{q'}) \operatorname{ancs}(q) \log |\Sigma|$. Denote this quantity by $\Delta_{q|q'}$. We now wish to find the set of states $S = \{q_2, \ldots, q_K\} \subseteq \mathcal{T} \smallsetminus \{q_1\}$ that maximizes $\sum_{k=2}^{K} \Delta_{q_k | q'_k}$, where q'_k is the highest state in $\{q_1, \ldots, q_K\}$ that is a proper descendant of q_k (that is, $q'_k > q_k$). This sum is the total improvement obtained by splitting at all of $\{q_2, \ldots, q_K\}$, since it is the total that would be obtained by splitting them successively in any reverse topological order.

For each state $q \in \mathcal{T}$ and each $q' > q$, define

$$\bar{\Delta}_{q|q'} = \max(\check{\Delta}_{q|q'}, \hat{\Delta}_{q|q'}) \tag{15}$$

$$\check{\Delta}_{q|q'} = \left(\sum_p \bar{\Delta}_{p|q}\right) + \Delta_{q|q'} \tag{16}$$

$$\hat{\Delta}_{q|q'} = \left(\sum_p \bar{\Delta}_{p|q'}\right) + 0 \tag{17}$$

where p in the summations ranges over the parents of q (if any) in the failure tree. Here $\bar{\Delta}_{q|q'} \geq 0$ is the maximum total improvement that can be obtained by splitting a failure tree rooted at $q' > q$ at any set of states $\leq q$; $\check{\Delta}_{q|q'}$ is the maximum if this set includes q, and $\hat{\Delta}_{q|q'}$ is the maximum if this set does not include q.[26] The optimal split of $\mathcal{T}$ then has total improvement $\sum_p \bar{\Delta}_{p|q_1}$ where q_1 is the root of $\mathcal{T}$ and p ranges over its parents.

Tracing back through the derivation of this optimal improvement, one may determine which states were split to obtain it. This is similar to following backpointers in the Viterbi algorithm. Concretely, define

$$\bar{S}_{q|q'} = \begin{cases} \hat{S}_{q|q'} & \text{if } \bar{\Delta}_{q|q'} = \hat{\Delta}_{q|q'} \\ \check{S}_{q|q'} & \text{otherwise} \end{cases} \tag{18}$$

$$\check{S}_{q|q'} = \left(\bigcup_p \bar{S}_{p|q}\right) \cup \{q\} \tag{19}$$

$$\hat{S}_{q|q'} = \left(\bigcup_p \bar{S}_{p|q'}\right) \cup \varnothing \tag{20}$$

For example, $\bar{S}_{q|q'} \geq 0$ is the optimal set of states

---

[26] A similar split into two cases—include q or exclude q—is used in the well-known linear-time dynamic programming algorithm for finding the max-weighted independent set of vertices in a tree. In effect, these algorithms label each tree node with a bit saying whether or not to include it, subject to some constraints. They resemble methods for refining the nonterminal labels of a parse tree (Petrov and Klein, 2007).

$\leq q$ to split in a failure tree rooted at $q' > q$. The optimal set of split points in $\mathcal{T}$, not counting the original root q_1, is $S = \bigcup_p \bar{S}_{p|q_1}$ where, again, p ranges over the parents of q_1. Any of the unions written here can be enumerated by (recursively) enumerating the disjoint sets that are unioned together, without any copying to materialize the sets.

Concretely, we first work from the leaves down to the root: at each q, we compute and memoize all of the Δ quantities (quantities (15)–(17) for all $q' > q$), after first having done so at the parents of q. We then enumerate S using the definitions (18)–(20), which recurse from the root back up to the leaves. Thanks to the choice at Eq. (18) based on the Δ quantities, this recursion enumerates only sets that are actually subsets of S. In particular, it enumerates $\bar{S}_{q|q'}$ for just those $q' > q$ pairs such that q' is the highest proper descendant of q in the optimal set $\{q_1\} \cup S$.

The total runtime is dominated by (15)–(17) and is proportional to the number of $q' > q$ pairs in $\mathcal{T}$. Summed over all trees $\mathcal{T}$, this is just the total height of all states in all failure trees, or equivalently $\sum_{q' \in Q} (\operatorname{ancs}(q') - 1)$. This resembles the failure term in the worst-case runtime of Alg. 6, but is much faster since it eliminates all factors that depend on $|\Sigma|$. Thus, when a compatible order is not known (§6.2), taking the time to optimally split the failure trees may be worth the investment.

Runtime analysis after static splitting To get a sense of how this improves the worst-case runtime, consider an idealized WFSA-ϕ where every state q has the same number of out-symbols, $\Sigma(q) = s|\Sigma|$. Furthermore, relax the runtime bound by replacing $\operatorname{ancs}(q)$ in the definition of $f(\mathcal{T})$ by the larger value $|\mathcal{T}|$, so $f(\mathcal{T}) \stackrel{\text{def}}{=} s|\Sigma||\mathcal{T}|^2$.

This means when we split $\mathcal{T}$ into $\mathcal{T}_1, \ldots, \mathcal{T}_K$, our earlier runtime expression $(K - 1)|\Sigma| + \sum_{k=1}^{K} f(\mathcal{T}_k) \log |\Sigma|$ becomes $(K - 1)|\Sigma| + \sum_{k=1}^{K} s|\Sigma||\mathcal{T}_k|^2 \log |\Sigma|$, or more simply, $|\Sigma|(K - 1 + (s \log |\Sigma|) \sum_{k=1}^{K} |\mathcal{T}_k|^2)$. For a given K, this is minimized when all K trees have equal size $\frac{|\mathcal{T}|}{K}$, yielding a minimum of

$$|\Sigma| \left(K - 1 + \frac{s \log |\Sigma|}{K} |\mathcal{T}|^2 \right) \tag{21}$$

Setting the derivative with respect to K to zero, we find that the optimal $K = |\mathcal{T}| \sqrt{s \log |\Sigma|}$.

However, for a WFSA with sufficiently dense out-symbols, namely one with $s > \frac{1}{\log |\Sigma|}$, this asks to take $K > |\mathcal{T}|$, which is impossible. There the

method will have to settle for $K = |\mathcal{T}|$, splitting each state into its own failure tree. This makes Alg. 6 reduce to Alg. 3.

Conversely, for a WFSA with sufficiently sparse out-symbols, namely one with $s < \frac{1}{|\mathcal{T}_{\max}|^2 \log |\Sigma|}$, the above formula asks to take $K < 1$ for all failure trees. That is also impossible: the method will have to settle for $K = 1$, not splitting $\mathcal{T}$ at all. This is the original version of Alg. 6.

In between these two extremes, we can take $K \approx |\mathcal{T}|\sqrt{s \log |\Sigma|}$ as proposed above. This makes the bound (21) on the contribution of failure tree $\mathcal{T}$ to the runtime become $\mathcal{O}(|\Sigma||\mathcal{T}|\sqrt{s \log |\Sigma|})$. Note that the $\sqrt{\ }$ term is < 1 because we are not too dense, so this may beat Alg. 3. It also beats the original Alg. 6: if we did not split the tree but kept $K = 1$, the expression would give $\mathcal{O}(|\Sigma||\mathcal{T}|^2 s \log |\Sigma|)$. In short, splitting the tree avoids the quadratic worst-case cost of Alg. 6. To put it another way, by eliminating the worst-case interaction among the K trees, we have reduced from $\mathcal{O}(|\Sigma|K^2)$ to $\mathcal{O}(|\Sigma|K)$. Recall that $K \geq 1$ since we are not too sparse, so this is again an improvement.
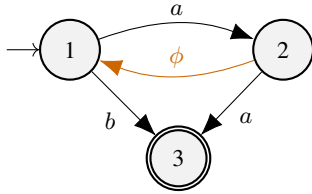
F Example of a Non-Suitable WFSA-ϕ

Figure 5: Example of a WFSA-ϕ where $E \cup E^\phi$ is not acyclic, yet its failure expanded transition set $\overline{E}$ is.