Efficient Online *k*-Best Lookup in Weighted Finite-State Cascades

Bryan Jurish

— FINAL DRAFT —

1 Introduction

Weighted finite-state transducers (WFSTs) have proved to be powerful and efficient aids for a variety of natural-language processing tasks, including automatic phonetization and phonological rule systems (Kaplan & Kay, 1994; Laporte, 1997), morphological analysis (Geyken & Hanneforth, 2006), and shallow syntactic parsing (Roche, 1997). In particular, *cascades* arising from the composition of two or more WFSTs can be used to model processing "pipelines", each component of which is itself a (weighted) finite-state transducer. Typically, the input to such a pipeline is a simple string, corresponding to a *lookup* operation for the input string in the processing cascade.

Unfortunately, an exhaustive "offline" compilation of the processing cascade turns out in many cases to be infeasible, due to memory restrictions and the combinatorial properties of the composition operation itself. Even for simple lookup operations in "dense" cascades,[1] the resulting WFST may in fact be several times larger than the processing pipeline itself. In many such cases – particularly in optimization and error-correction problems – the output WFST itself serves only as an intermediate processing datum, however: we are not interested in an exhaustive representation of the lookup output, but rather only in a small finite subset of its language, such as the *k-best paths*.

This paper presents a novel algorithm for efficient k-best search in a subclass of weighted finite-state lookup cascades which avoids the combinatorial explosion associated with "dense" cascade relations by means of *online computation*:[2] dynamic construction of only those states and arcs required for a k-best search of the lookup output. Use of a *greedy termination* clause together with an additional *cutoff parameter* helps to ensure speedy completion and simultaneously prune unwanted results from the output.

[1]Informally, the "density" of a cascade C corresponds to the cardinality of the underlying rational relation $|[\![C]\!]| \leq |\Sigma^* \times \Gamma^*|$; the densest cascades containing at least one valid path for every pair of in- and output-strings $(s,t) \in \Sigma^* \times \Gamma^*$.

[2]Also sometimes referred to as "lazy evaluation" or "on-the-fly computation".

1.1 Example Application

As an example application, consider the task of *orthographic standardization* of historical text, which must precede any adequate treatment of historical corpora by conventional NLP tools, due to the lack of consistent orthographic conventions in such corpora (Jurish, 2008). In this scenario, the processing cascade consists of at least:

- a weighted *edit transducer* M_Δ which robustly models (potential) diachronic change likelihood as a (dense) weighted rational relation, and

- a *target acceptor* A_L representing the synchronically active lexicon of extant word forms.

The processing cascade $C_{\Delta L} = M_\Delta \circ A_L$ thus models all potential diachronic changes resulting in some extant word form. A lookup cascade $C_{\vec{w}\Delta L} = (\mathrm{Id}\,(\vec{w}) \circ C_{\Delta L}) = (\mathrm{Id}\,(\vec{w}) \circ M_\Delta \circ A_L)$ for a historical text form $\vec{w}$ in the cascade $C_{\Delta L}$ represents the set of all extant forms $\vec{v}$ into which $\vec{w}$ may have evolved, weighted by the likelihood of a direct etymological relation $\vec{w} \rightsquigarrow \vec{v}$. The k-best output strings of the lookup cascade are then simply the k extant word forms considered most likely to be directly related to the historical form $\vec{w}$. Restricting the admissable output paths by applying an external cutoff threshold $c_{\max}$ is equivalent to imposing an *a priori* upper bound on the likelihood of acceptable diachronic derivations, which is especially important in the case of a dense editor M_Δ.

1.2 Desiderata

In light of the preceding example, a number of important properties for a candidate solution may be identified:

- **Online computation:** states and transitions of intermediate processing stages should be computed "on-the-fly" and discarded when no longer needed, to avoid the combinatorial explosion associated with dense cascades.

- **Type-wise input:** the algorithm should function efficiently for type-wise input, for maximal flexibility.

- **k-best strings:** output of the algorithm should be an enumeration of the k best strings of the lookup output for a user-specified natural number k, thus allowing the user some control over the maximum degree of ambiguity returned.

- **Arbitrary cascade depth:** the algorithm should not itself impose any upper bound on the depth of the processing cascade. In particular, pair-wise “lazy evaluation” of sub-cascades is to be avoided, since such methods – although elegant and formally correct – tend to introduce nontrivial amounts of runtime and memory overhead.[3]

- **Arbitrary regular weighting function:** the algorithm should function correctly for arbitrary regular weighting functions, *i.e.* for arbitrary cascades of weighted finite-state transducers. In particular, no assumption should be made about the cascade architecture regarding the presence, placement, content, or disposition of an “editor WFST” such as M_Δ.[4]

- **Cutoff threshold:** the algorithm should accept as an additional parameter a cutoff threshold which serves to further restrict the set of acceptable output paths.

- **Greedy termination:** the algorithm should terminate and return as quickly as possible in the average case; *i.e.* as soon as the k best paths have been discovered, or it has been determined that no further paths are to be found below the cutoff threshold.

2 Formal Background

Definition 1 (Semiring). *A structure $\mathcal{K} = \langle \mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1} \rangle$ is a semiring if*

1. *$\langle \mathbb{K}, \oplus, \bar{0} \rangle$ is a commutative monoid with $\bar{0}$ as the identity element for $\oplus$,*

2. *$\langle \mathbb{K}, \otimes, \bar{1} \rangle$ is a monoid with $\bar{1}$ as the identity element for $\otimes$,*

3. *$\otimes$ distributes over $\oplus$, and*

4. *$\bar{0}$ is an annihilator for $\otimes$: $\forall a \in \mathbb{K}, a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.*

$\mathcal{K}$ is *commutative* if $\forall a, b \in \mathbb{K}, a \otimes b = b \otimes a$, and $\mathcal{K}$ is *idempotent* if $\forall a \in \mathbb{K}, a \oplus a = a$. For an idempotent semiring $\mathcal{K}$, the *natural order* over $\mathbb{K}$ is the partial order $\leq_{\mathbb{K}}$ defined by $\forall a, b \in \mathbb{K}, (a \leq_{\mathbb{K}} b) :\Leftrightarrow ((a \oplus b) = a)$. The natural order is both *negative* (i.e. $\bar{1} \leq_{\mathbb{K}} \bar{0}$) and *monotonic*, $\forall a, b, c \in \mathbb{K}, (a \leq_{\mathbb{K}} b)$ implies $(a \oplus c) \leq_{\mathbb{K}} (b \oplus c)$, $(a \otimes c) \leq_{\mathbb{K}} (b \otimes c)$, and $(c \otimes a) \leq_{\mathbb{K}} (c \otimes b)$. $\mathcal{K}$ is said to be *bounded* if $\bar{1}$ is an annihilator for $\oplus$: $\forall a \in \mathbb{K}, \bar{1} \oplus a = \bar{1}$. In a bounded semiring, $\bar{1} \leq_{\mathbb{K}} a \leq_{\mathbb{K}} \bar{0}$ for all $a \in \mathbb{K}$. Every bounded semiring is

[3]Preliminary tests with the OpenFst library (Allauzen et al., 2007) supported these intuitions.

[4]This desideratum is considered a critical feature of any candidate solution, eliminating specialized techniques such as those described in Oflazer & Güzey (1994); Oflazer (1996), relying as these do on an implicit *edit distance* weighting function in the style of Levenshtein (1966); Wagner & Fischer (1974), rather than the weighting function arising from an arbitrary WFST cascade.

also idempotent (c.f. Mohri, 2002, Lemma 3). For current purposes, we will restrict our attention to bounded semirings.

Definition 2 (WFST). *A weighted finite-state transducer over a semiring $\mathcal{K}$ is a 6-tuple $M = \langle \Sigma, \Gamma, Q, q_0, F, E\rangle$ with:*[5]

1. *Σ a finite input (or "lower") alphabet,*
2. *Γ a finite output (or "upper") alphabet,*
3. *Q a finite set of states,*
4. *$q_0 \in Q$ the designated* initial state*,*
5. *$F \subseteq Q$ the set of final states, and*
6. *$E \subseteq Q \times Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times \mathbb{K}$, a finite set of transitions.*

For a transition $e = (q_1, q_2, a, b, c) \in E$, we denote by $p[e]$ its source state q_1, by $n[e]$ its destination state q_2, by $i[e]$ its input label a, by $o[e]$ its output label b, and by $c[e]$ its weight (or "cost") c. A weighted finite-state acceptor (WFSA) can be regarded as a WFST with $\Sigma = \Gamma$ and $i[e] = o[e]$ for all $e \in E$.

Definition 3 (String Transducer). *For a string $\vec{w} = w_1 \cdots w_n \in \Sigma^*$ over an alphabet Σ, the string transducer for $\vec{w}$ is the WFSA $\mathrm{Id}(\vec{w}) = \langle \Sigma,\ \Sigma,\ Q_{\vec{w}},\ 0,\ \{n\},\ E_{\vec{w}}\rangle$ with $Q_{\vec{w}} = \{i \in \mathbb{N} : i \leq n\}$ and $E_{\vec{w}} = \bigcup_{i=1}^{n}\{(i-1, i, w_i, w_i, \bar{1})\}$.*

A path π is a finite sequence $e_1 e_2 \ldots e_{|\pi|}$ of $|\pi|$ transitions such that $n[e_i] = p[e_{i+1}]$ for $1 \leq i < |\pi|$. Extending the notation for transitions, we define the source and sink states of a path as $p[\pi] = p[e_1]$ and $n[\pi] = n[e_{|\pi|}]$, respectively. The input label string $i[\pi]$ yielded by a path π is the concatenation of the input labels of its transitions: $i[\pi] = i[e_1]i[e_2]\ldots i[e_{|\pi|}]$; the output label string $o[\pi]$ is defined analogously. The weight $c[\pi]$ of a path π is the $\otimes$-product of its transitions: $w[\pi] = \bigotimes_{i=1}^{|\pi|} c[e_i]$.

If $p[\pi] = q_0$ and $n[\pi] \in F$, π is called *successful*. A *cycle* is a path π with $p[\pi] = n[\pi]$. A string transducer $\mathrm{Id}(\vec{w})$ contains exactly one successful path $\pi_{\vec{w}}$ with $i[\pi_{\vec{w}}] = o[\pi_{\vec{w}}] = \vec{w}$. A state $r \in Q$ is said to be *accessible from* a state $q \in Q$ if there exists a path π with $p[\pi] = q$ and $n[\pi] = r$; r is *accessible* if it is accessible from q_0. For $q \in Q$, $\vec{w} \in \Sigma^*$, $\vec{v} \in \Gamma^*$, and $R \subseteq Q$, $\Pi(q, \vec{w}, \vec{v}, R)$ denotes the set of paths from q to some $r \in R$ with input string $\vec{w}$ and output string $\vec{v}$, and $\Pi(q, R) = \bigcup_{\vec{w}\in\Sigma^*, \vec{v}\in\Gamma^*} \Pi(q, \vec{w}, \vec{v}, R)$ denotes the set of paths originating at q and ending at some $r \in R$.

[5]WFSTs are sometimes defined with an additional *final weight* function $\rho : Q \to \mathbb{K}$, and/or a non-deterministic *initial weight function* $\alpha : Q \to \mathbb{K}$ in place of q_0. I ignore these extensions here in the interest of clarity.

Definition 4 (Transducer Weight)**.** *The weight assigned by a WFST M to a pair of strings $(\vec{w}, \vec{v}) \in \Sigma^* \times \Gamma^*$ is defined as*

$$[\![M]\!](\vec{w}, \vec{v}) = \bigoplus_{\pi \in \Pi(q_0, \vec{w}, \vec{v}, F)} c[\pi]$$

Definition 5 (Composition of WFSTs)**.** *Given WFSTs $M_1 = \langle \Sigma, \Gamma, Q_1, q_{0_1}, F_1, E_1 \rangle$ and $M_2 = \langle \Gamma, \Delta, Q_2, q_{0_2}, F_2, E_2 \rangle$ over a commutative and complete*[6] *semiring $\mathcal{K}$, the composition of M_1 and M_2 is written $M_1 \circ M_2$, and is itself a WFST such that for all $\vec{w} \in \Sigma^*$, $\vec{v} \in \Delta^*$:*

$$[\![M_1 \circ M_2]\!](\vec{w}, \vec{v}) = \bigoplus_{\vec{u} \in \Gamma^*} [\![M_1]\!](\vec{w}, \vec{u}) \otimes [\![M_2]\!](\vec{u}, \vec{v})$$

Further, $M_3 = \langle \Sigma, \Delta, (Q_1 \times Q_2), E_3, (q_{0_1}, q_{0_2}), (F_1 \times F_2) \rangle$ is such a WFST, $[\![M_3]\!] = [\![M_1 \circ M_2]\!]$, where:[7]

$$\begin{aligned} \widetilde{Q} &= \{(q, q, \varepsilon, \varepsilon, \bar{1}) : q \in Q\} \\ E_3 &= \bigcup_{\substack{(q_1, r_1, a_1, a_2, c_1) \in E_1 \cup \widetilde{Q_1} \\ (q_2, r_2, a_2, a_3, c_2) \in E_2 \cup \widetilde{Q_2}}} \{((q_1, q_2), (r_1, r_2), a_1, a_3, c_1 \otimes c_2)\} \end{aligned}$$

3 Algorithms

This section develops an algorithm for discovering the k-best label paths in a dense cascade of weighted finite state transducers over a bounded semiring, attempting to fulfill the desiderata from section 1.2. We begin with a brief review of the well-known algorithm which serves as the basis for the current approach, and consider its generalization to abstract semiring weights in section 3.1. Section 3.2 extends the algorithm to online lookup operations in weighted finite-state cascades. Section 3.3 explores the implications of a greedy k-best termination strategy, and section 3.4 extends the discussion to include a user-specified cutoff threshold. Finally, section 3.5 addresses the problem of extending the algorithm to return output label strings.

The current approach is best understood as a variant of the well-known *Dijkstra Algorithm* (Dijkstra, 1959; Cormen et al., 2001), presented here as Algorithm 1. Using a Fibonacci heap (Fredman & Tarjan, 1987) to implement the processing queue S, and assuming constant time access to outgoing edges for a vertex, Algorithm 1 has running time $\mathcal{O}(\textsc{Dijkstra}) = \mathcal{O}(|E| + |V| \log |V|)$.

[6] c.f. Ésik & Kuich (2004)

[7] The construction given here for E_3 is only valid for idempotent semirings; c.f. Mohri et al. (1996) for a generalization to non-idempotent semirings.

Algorithm 1 Dijkstra (1959)

```
1: function DIJKSTRA($V, E, v_0$)
2:     $d[\cdot] := \{v \mapsto \infty : v \in V\}$                /* Initialize */
3:     $d[v_0] := 0$
4:     $S := V$
5:     while $S \neq \emptyset$                                  /* Main loop */
6:         $u := \arg\min_{u \in S} d[u]$                    /* Best-first search */
7:         $S := S \backslash \{u\}$
8:         foreach $e \in E : p[e] = u$                          /* Expand */
9:             $d' := d[u] + c[e]$                           /* Accumulate */
10:            if $d' < d[v]$ then                              /* Relax */
11:                $d[v] := d'$
12:    return $d[\cdot]$
```

3.1 Semiring Weights

In its original form, Dijkstra's algorithm assumes a graph $G = \langle V, E\rangle$ with non-negative real-valued weighted edges $E \subseteq (V \times V \times \mathbb{R}_+)$, ordered by the natural linear order $<$, thus implicitly equating "best" with "$<$-minimal". The first adaptation to be undertaken is a straightforward generalization of the Dijkstra algorithm to an abstract semiring $\mathcal{K} = \langle \mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1}\rangle$, using minimality with respect to a partial order $\leq_{\mathbb{K}}$ to define "best" weights.

First, the initialization of the best-distance vector $d[\cdot]$ must be adapted to use the relevant semiring constants $\bar{0}$ and $\bar{1}$:

```
2: $d[\cdot] := \{v \mapsto \bar{0} : v \in V\}$        /* Initialize $d[\cdot] : V \to \mathbb{K}$ */
3: $d[v_0] := \bar{1}$
```

Next, the best-first order of extraction from the vertex-queue S must be adapted to use the partial order $\leq_{\mathbb{K}}$:

```
6: $u := \arg\min_{u \in S, <_{\mathbb{K}}} d[u]$      /* Best-first search using $<_{\mathbb{K}}$ */
```

Finally, the RELAX step is adapted to use the semiring multiplication operation $\otimes$ for accumulating the characteristic weight of a path, as well as the semiring order for the "better-path" check of line 10:

```
9: $d' := d[u] \otimes c[e]$                     /* Accumulate using $\otimes$ */
10: if $d' <_{\mathbb{K}} d[v]$ then                         /* Relax */
11:     $d[v] := d'$
```

Dijkstra's original algorithm emerges as an instance of the generalized algorithm using the non-negative tropical semiring $\langle \mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0\rangle$ (Simon, 1987). Important to note is that the generalization to abstract semirings has implications for the correctness of the algorithm. In particular, graph cycles with a net weight $c <_{\mathbb{K}} \bar{1}$ will cause the algorithm never to terminate at all, inducing an infinite series of $<_{\mathbb{K}}$-decreasing weights $d[u]$ for the cycle root vertex u, leading to an infinite loop of RELAX steps. Further, the relaxability check of line 10 is not meaningful for all partial orders $\leq_{\mathbb{K}}$: a non-monotonic order may cause a partial path to be disregarded here which would lead to a better path for some subsequent vertex. We therefore restrict our attention for current purposes to *bounded* (idempotent) semirings using the monotonic *natural semiring order* (c.f. section 2), $\forall a, b \in \mathbb{K}$:

$$\begin{array}{rclr} (\bar{1} \oplus a) & = & \bar{1} & \text{(Boundedness)} \\ (a \oplus a) & = & a & \text{(Idempotence)} \\ (a \leq_{\mathbb{K}} b) & \Leftrightarrow & ((a \oplus b) = a) & \text{(Natural Order)} \end{array}$$

3.2 Online Cascade Lookup

The next task is to extend the algorithm to operate on lookup cascades $C = (\mathrm{Id}(\vec{w}) \circ M_2 \circ \cdots \circ M_{|C|})$ for $\vec{w} \in \Sigma_2^*$. Suppose the standard construction for composition of WFSTs given in Definition 5 yields for C the WFST $M = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$. Clearly, $\langle Q, E \rangle$ can be treated as an edge-labelled weighted graph. By assumption however, M is too large to be computed offline, so that in particular the composition of transitions E must be performed at runtime. The resulting algorithm is presented here together with some auxiliary subroutines as Algorithm 2.

The online expansion of outgoing transitions from a state $q = \langle q_{\vec{w}}, q_2, \ldots, q_{|C|} \rangle \in Q$ is performed by the auxiliary function ARCS given in Algorithm 2.[8] ARCS is implemented as a pair of calls to the function EXPAND-ARCS, which recursively descends the cascade, linking together transitions from adjacent components with matching out- rsp. input labels in accordance with Definition 5.

The only other change made to the core algorithm DIJKSTRA-CASCADE is a move to sparse administrative structures: rather than initialize the queue S with the set of all cascade states Q, which would entail explicitly representing such states and thus pre-compiling them, Algorithm 2 instead uses a dynamic queue S which at any given point in the computation holds only those states which need to be (re-)investigated. Similarly, the map $d[\cdot]$ of best weights is implemented as a sparse partial map, and the default case $d[q] = \bar{0}$ is handled by the the auxiliary function COST. For Algorithm 2, the use of sparse structures has few consequences – states unreachable from q_0 will no longer be processed, but the algorithm otherwise proceeds exactly as in Algorithm 1, with running time growing by a factor of the cascade depth $|C|$ to allow for online expansion of transitions. Since cascade depth is expected to be a small constant, we ignore it in the sequel.

[8]The function ARCS takes advantage of the facts that in a lookup cascade, the initial component $\mathrm{Id}(\vec{w})$ has at most one outgoing (non-ε) arc e, and that $i[e] = o[e] = \vec{w}[q[1]]$. A generalization to arbitrary WFST cascades would involve iterating over all outgoing arcs of the initial cascade component here.

Algorithm 2 Dijkstra's algorithm for online lookup cascades

```
1:  function DIJKSTRA-CASCADE($w, C$)
2:      $S := \{q_0\}$                                                /* Initialize */
3:      $d[\cdot] := \{q_0 \mapsto \bar{1}\}$
4:      while $S \neq \emptyset$                                      /* Main loop */
5:          $q := \arg\min_{q \in S, <_{\mathbb{K}}} d[q]$             /* Best-first search */
6:          $S := S \backslash \{q\}$
7:          foreach $e \in$ ARCS($w, C, q$)                 /* Expand outgoing arcs */
8:              $d' := d[q] \otimes c[e]$                             /* Accumulate */
9:              if $d' <_{\mathbb{K}}$ COST($d[\cdot], n[e]$) then                /* Relax */
10:                 $d[n[e]] := d'$
11:                 $S := S \cup \{n[e]\}$                            /* Enqueue */
12:     return $d[\cdot]$
13: function ARCS($\vec{w}, C, q$)
14:     return EXPAND-ARCS $(C, q, 1, \vec{w}[q[1]])$ $\cup$ EXPAND-ARCS($C, q, 1, \varepsilon$)
15: function EXPAND-ARCS($C, q, i, a$)
16:     $A := \emptyset$
17:     foreach $e \in E_i \cup \{(q[i], q[i], \varepsilon, \varepsilon, \bar{1})\} : p[e] = q[i]$ & $i[e] = a$
18:         if $i = |C|$ then
19:             $A := A \cup \{e\}$
20:         else
21:             foreach $e' \in$ EXPAND-ARCS($C, q, i+1, o[e]$)
22:                 $A := A \cup \{(\langle p[e], p[e']\rangle, \langle n[e], n[e']\rangle, i[e], o[e'], c[e] \otimes c[e'])\}$
23:     return $A$
24: function COST($d[\cdot], q$)
25:     if $d[q]$ defined then return $d[q]$
26:     return $\bar{0}$
```

3.3 k-Best Final States

Dijkstra's algorithm solves the *single source shortest distances* problem, returning a map $d : Q \to \mathbb{K}$ which associates each state with the best net weight of any path to that state from the designated initial state q_0. In the current problem context, we are not interested in an exhaustive enumeration $d[\cdot]$ of net weights for all cascade states, but rather only for the *final states* of the lookup output: $d_F : F \to \mathbb{K}$. Even more specifically, we are interested only in the k-best mappings for some final state, a partial function $d_{F,k} : F \xrightarrow{\text{partial}} \mathbb{K}$ such that the following hold:

$$\begin{aligned} d_{F,k} &\subseteq d_F \subseteq d \\ |d_{F,k}| &\leq k \\ \forall q, r \in Q \,.\, d[q] <_{\mathbb{K}} d[r] \;\&\; r \in \mathrm{dom}(d_{F,k}) &\Rightarrow q \in \mathrm{dom}(d_{F,k}) \end{aligned}$$

Clearly, $d_F = (d \restriction F)$ is simply the restriction of the map $d[\cdot]$ to the subset F of final states, and $d_{F,k}$ can be generated from d_F by extraction of the $j \leq k <_{\mathbb{K}}$-minimal elements. Not only does such an extraction add additional runtime complexity,[9] it requires that Algorithm 2 first run in its entirety, which as noted above is unacceptable for dense cascades. Instead, we can optimize the algorithm for the task of discovering $d_{F,k}[\cdot]$ as given in Algorithm 3.[10]

Algorithm 3 Dijkstra's algorithm adapted for k-best net weights to final states

```
 1: function DIJKSTRA-KBEST($w, C, k$)
 2:     $S := \{q_{0_C}\}$                                  /* Initialize */
 3:     $d[\cdot] := \{q_{0_C} \mapsto \bar{1}\}$
 4:     $d_{F,k}[\cdot] := \emptyset$
 5:     while $S \neq \emptyset$                            /* Main loop */
 6:         $q := \arg\min_{q \in S, <_{\mathbb{K}}} d[q]$          /* Best-first search */
 7:         $S := S \backslash \{q\}$
 8:         if $q \in F_C$ then                              /* Finality check */
 9:             $d_{F,k}[q] := d[q]$
10:             if $|d_{F,k}| = k$ then break              /* Greedy termination */
11:         foreach $e \in \text{ARCS}(w, C, q)$          /* Expand outgoing arcs */
12:             $d' := d[q] \otimes c[e]$                      /* Accumulate */
13:             if $d' <_{\mathbb{K}} \text{COST}(d[\cdot], n[e])$ then          /* Relax */
14:                 $d[n[e]] := d'$
15:                 $S := S \cup \{n[e]\}$                        /* Enqueue */
16:     return $d_{F,k}[\cdot]$
```

The best-first queue management and relaxation strategy of Algorithms 1 and 2 remains unchanged in the function DIJKSTRA-KBEST of Algorithm 3, thus DIJKSTRA-KBEST terminates whenever DIJKSTRA does, and the correctness conditions are unaffected – termination is guaranteed for bounded semirings. The additional statements in lines 8-10 can all be implemented as (amortized) constant-time operations, so the worst-case running time also remains unchanged: $\mathcal{O}(\text{DIJKSTRA-KBEST}) = \mathcal{O}(\text{DIJKSTRA}) = \mathcal{O}(|E| + |Q| \log |Q|)$. Memory use grows in the worst case by at most $\mathcal{O}(|F|)$ for storage of the partial output map $d_{F,k}[\cdot]$.

More important for the current problem context are the average case time and space complexity for DIJKSTRA-KBEST *vs.* the original DIJKSTRA-CASCADE function. Whereas DIJKSTRA-CASCADE must always compute the

[9] $\mathcal{O}(|F|k)$, using a standard implementation for small k.

[10] As originally presented by Dijkstra (1959), the algorithm includes a single designated sink vertex parameter as well as a greedy termination clause, which is extended here to a set of designated final states.

net weight of at least one complete path to each state, Dijkstra-kBest need only compute weights for at most k paths ending in final states. That the first such weights computed are indeed the k best weights sought follows from the correctness of the best-first search order, which in turn follows from the boundedness of the semiring $\mathcal{K}$. Since immediately upon discovery of the k^{th} best weight to a final state at line 10, Algorithm 3 breaks out of the queue-processing loop and returns the partial map $d_{F,k}$, its time complexity can be more precisely specified by (1),

$$\mathcal{O}\left(\textsc{Dijkstra-kBest}\right) = \mathcal{O}\left(|E_{F,k}| + |Q_{F,k}| \log |Q_{F,k}|\right) \tag{1}$$

where:

$$\begin{aligned} Q_{F,k} &= \{q \in Q : d[q] \leq_{\mathbb{K}} \max\left(\operatorname{rng}\left(d_{F,k}\right)\right)\} \\ E_{F,k} &= \{e \in E : p[e] \in Q_{F,k}\} \end{aligned}$$

Assuming that k best final weights were indeed found (which will always be the case if $k \leq |F|$), $Q_{F,k}$ is the set of states to which at least one path exists with a net weight less than or equal to some k-best final weight in $d_{F,k}$, and $E_{F,k}$ is the set of all transitions leaving any state in $Q_{F,k}$. By the correctness of the best-first search order for bounded semirings, $Q_{F,k}$ contains all and only those states q which may be extracted from the queue at line 6 before discovery of the k^{th} best net weight to a final state at line 8 and consequent termination at line 10. It follows that $E_{F,k}$ is the set of transitions which must be expanded (and possibly relaxed) by the loop of lines 11-15.

In many interesting cases, $Q_{F,k}$ and $E_{F,k}$ will be much smaller than Q and E respectively, so that the reduced time complexity of Equation (1) represents a major improvement over a brute force approach using Algorithm 2 directly. Consider for example a simple error-correction cascade similar to that described in section 1.1, and let p_c be the average probability over all states $q \in Q_{\vec{w}\Delta L}$ that a path exists from the initial state $q_{0_{\vec{w}\Delta L}}$ to q with net weight $c' \leq_{\mathbb{K}} c$. If $c \in \mathbb{K}$ is the maximum weight to a k-best final state, then the expected size of $Q_{F,k}$ is $\mathrm{E}(|Q_{F,k}|) = \mathrm{E}_{p_c}(\mathbb{1}_{Q_{\vec{w}\Delta L}}) = \sum_{q \in Q_{\vec{w}\Delta L}} p_c = p_c |Q_{\vec{w}\Delta L}|$. The number of states which must be expanded for a k-best search with maximum net path weight c thus depends crucially on p_c, which can be understood as the probability of the existence of a "neighbor" path with edit cost $c' \leq_{\mathbb{K}} c$. It is therefore of paramount importance for purposes of runtime efficiency both (a) to ensure that M_Δ models the phenomena it is intended to represent as accurately as possible, effectively minimizing p_c globally for all $c \in \mathbb{K}$, and (b) to minimize p_c locally by preventing $c = \max(\operatorname{rng}(d_{F,k}))$ from growing too large, since $c \leq c'$ implies $p_c \leq p_{c'}$.

3.4 Cutoff Threshold

An unsubtle but effective method for local minimization of the maximum path weight returned by Algorithm 3 is the explicit specification of a user-specified cutoff threshold $c_{\max} \in \mathbb{K}$ on path weights as an input parameter. Intuitively, such a parameter represents an *a priori* upper bound on the cost of "acceptable" paths. For the example application from section 1.1, a parameter $c_{\max}$ can limit the algorithm's running time even when the input word $\vec{w}$ represents an extinct lexeme not explicitly accounted for by a dense M_Δ, in which case its k nearest neighbors according to $[\![M_\Delta]\!]$ would be randomly distributed in L, and their inclusion as "best" paths for $\vec{w}$ would only introduce noise (both precision and recall errors) into the host application. Implementing the parameter $c_{\max}$ for Algorithm 3 requires only the insertion of a simple check after line 7:

```
if $d[q] >_{\mathbb{K}} c_{\max}$ then break /* Cost upper-bound exceeded */
```

Whenever $c_{\max}$ is exceeded for the minimum-cost state in the queue, it must also be exceeded for every other queued state as well. Since $\leq_{\mathbb{K}}$ is monotonic, queue processing can cease as soon as any any state with a minimum net path weight exceeding $c_{\max}$ is extracted from the queue. Note that while it is possible in the case of the example cascade architecture from section 1.1 to incorporate $c_{\max}$ into the edit transducer by modifying M_Δ such that for all $\vec{w} \in \Sigma_\Delta^*, \vec{v} \in \Gamma_\Delta^*$, $\Pi(q_{0_\Delta}, \vec{w}, \vec{v}, F_\Delta) \neq \emptyset$ implies $[\![M_\Delta]\!](\vec{w}, \vec{v}) \leq_{\mathbb{K}} c_{\max}$, such a construction not only introduces additional storage requirements by introducing new states into M_Δ, but is not in general possible if the processing cascade (which by assumption is too large to be computed and stored offline in its entirety) contains multiple independent weighted components. Implementation of the upper bound as a parameter does not increase run time complexity or storage requirements for the algorithm, and allows additional flexibility: the user may for example choose to instantiate $c_{\max}$ as a function of input word length, representing the upper bound in terms of average cost per character rather than a global cost for all words.

3.5 Label Strings

Extending the algorithm to return the k-best (output) label strings rather than the k-best net path weights is not as trivial a task as it may at first appear. The traditional method (Cormen et al., 2001) of maintaining a backtrace vector $p[\cdot] : Q \to Q$ mapping states to their best predecessors causes the number of returned paths $|d_{F,k}|$ be bounded above by the number of final states $|F|$, and does not correctly compute the k-best paths if these are defined to include labels in addition to states. Extending the semiring $\mathcal{K}$ to a k-best semiring $\mathcal{K}_k$ as described by Mohri (2002) not only yields a non-idempotent semiring, but also entails additional modifications for direct storage of path backtraces in the semiring itself.

The current approach instead extends the processing queue S to store state-string pairs $\langle q, s\rangle \in Q \times \Gamma^*$ such that s is the output label string of some path from q_0 to q. The best-weight vector is then re-defined as a (sparse) map $d[\cdot] : Q \times \Gamma^* \to \mathbb{K}$ such that $d[q, s]$ represents the net weight associated with the best path from q_0 to q with output label string s, and the output buffer $d_{F,k}$ is similarly extended to a buffer $d_{\Pi,k}$. An additional kludge[11] parameter $x_{\max} \in \mathbb{N}$ limits the number of allowable queue extractions. The resulting algorithm is presented here as Algorithm 4.

Algorithm 4 Adaptation of Dijkstra's algorithm for k-best output label strings

```
 1: function DIJKSTRA-STRINGS($\vec{w}, C, k, c_{\max}, x_{\max}$)
 2:     $S := \{\langle q_0, \varepsilon\rangle\}$                                  /* Initialize */
 3:     $d[\cdot] := \{\langle q_0, \varepsilon\rangle \mapsto \bar{1}\}$
 4:     $d_{\Pi,k}[\cdot] := \emptyset$
 5:     while $S \neq \emptyset$                                       /* Main loop */
 6:         $\langle q, s\rangle := \arg\min_{\langle q,s\rangle \in S, <_{\mathbb{K}}} d[q, s]$          /* Best-first search */
 7:         $S := S \backslash \{\langle q, s\rangle\}$
 8:         if $x_{\max} = 0$ then break                  /* Too many extractions */
 9:         $x_{\max} := x_{\max} - 1$
10:         if $d[q, s] >_{\mathbb{K}} c_{\max}$ then break /* Cost upper-bound exceeded */
11:         if $q \in F$ then                                   /* Finality check */
12:             $d_{\Pi,k}[q, s] := d[q, s]$
13:             if $|d_{\Pi,k}| = k$ then break               /* Greedy termination */
14:         foreach $e \in$ ARCS($w, C, q$)              /* Expand outgoing arcs */
15:             $d' := d[q, s] \otimes c[e]$                           /* Accumulate */
16:             $s' := s^\frown o[e]$                                    /* Append */
17:             if $d' <_{\mathbb{K}}$ COST($d[\cdot], \langle n[e], s'\rangle$) then                /* Relax */
18:                 $d[n[e], s'] := d'$
19:                 $S := S \cup \{\langle n[e], s'\rangle\}$
20:     return $d_{\Pi,k}[\cdot]$
```

Assume for the moment that the kludge parameter is vacuous, e.g. $x_{\max} = -1$. If the cascade contains an instance of a certain type of "degenerate" cycle, then Algorithm 4 may never terminate at all. A degenerate cycle in this sense can be operationally defined as one which may lead to an infinite series of RELAX steps for increasingly long label strings at lines 17 through 19. Formally, we call a cycle π degenerate if (2)-(5) hold for some $\pi' \in E^*$

[11] c.f. Raymond (2010)

and for all $n \in \mathbb{N}$.

$$\begin{array}{rcll} \pi'\pi & \in & \Pi(q_0, Q) & (2) \\ i[\pi] & = & \varepsilon & (3) \\ o[\pi] & \neq & \varepsilon & (4) \\ c\,[\pi^n] & \leq_{\mathbb{K}} & c_{\max} & (5) \end{array}$$

Condition (2) requires that degenerate cycles are accessible. A non-accessible cycle can never induce an infinite loop, since only accessible states are ever inserted into the queue at line 19. Condition (3) states that only paths with an empty input string can be degenerate. This follows from the fact that C is a lookup cascade with initial component $\mathrm{Id}\,(\vec{w})$, so that the maximum number of iterations for a cycle with $i[\pi] = s \neq \varepsilon$ is $\frac{|\vec{w}|}{|s|}$. Condition (4) states that degenerate cycles must have non-empty output strings, since a cycle with $o[\pi] = \varepsilon$ generates at most one index configuration $\langle q, s\rangle$ for its source state $q = p[\pi]$, and this configuration will fail the relaxability check at line 17 after the first iteration. Finally, condition (5) captures the intuition that a degenerate cycle may be iterated arbitrarily many times without its weight exceeding the bound parameter[12] $c_{\max}$, $c[\pi^n] = c[\pi]^n = \bigotimes_{i=1}^n c[\pi] \leq_{\mathbb{K}} c_{\max}$, and thus will never be pruned by the check at line 10. In particular, this condition attains for $c[\pi] = \bar{1} <_{\mathbb{K}} c_{\max}$, since $\bar{1}^n = \bar{1}$ for all $n \in \mathbb{N}$. We denote by $\widetilde{\Pi}(C)$ the set of paths for which the weight-independent criteria (2)-(4) hold, $\widetilde{\Pi}(C) = \{\pi \in E^* : \exists \pi' \in E^* : \pi'\pi \in \Pi(q_0, Q)\ \&\ p[\pi] = n[\pi]\ \&\ i[\pi] = \varepsilon \neq o[\pi]\}$

That Algorithm 4 finds the k best label strings in the absence of degenerate cycles whenever at least k distinctly labelled successful paths exist follows from the correctness of Algorithm 3. Note in particular that paths with distinct label strings ending in the same state are treated as distinct objects, as are paths with identically labelled strings ending in distinct states, analogous to the trellis construction used in the well-known *Viterbi algorithm* (Viterbi, 1967).

Rather than rely on an expensive cycle check to detect degenerate cycles, we introduce the kludge parameter $x_{\max}$ which places an upper bound on the number of queue extractions performed and limits the running time of Algorithm 4 to $\mathcal{O}(x_{\max}(\max(\deg(Q)) + \log x_{\max}))$, where $\max(\deg(Q))$ is the maximum output degree of any state in Q, $\deg(q \in Q) = |\{e \in E : p[e] = q\}|$. Despite its inelegance, this technique can be useful in both development and production environments – in the former to detect and report potential errors in the cascade architecture, and in the latter to place a hard limit on the computational resources consumed.

[12]For purposes of defining path degeneracy without a greedy termination clause, the upper bound variable $c_{\max}$ may be defined in terms of the maximum target weight, $c_{\max} = \max(\mathrm{rng}(d_{\Pi,k}))$.

When $x_{\max}$ is finite but the break at line 8 is not responsible for its termination, Algorithm 4 has the running time specified in (6),

$$\mathcal{O}\left(\textsc{Dijkstra-Strings}\right) = \mathcal{O}\left(|E_{\Pi,k}| + |V_{\Pi,k}| \log |V_{\Pi,k}|\right) \tag{6}$$

where:

$$\begin{aligned} V_{\Pi,k} &= \{\langle q, s\rangle \in Q \times \Gamma^* : d[q,s] \leq_{\mathbb{K}} \max\left(\mathrm{rng}\left(d_{\Pi,k}\right)\right)\} \\ E_{\Pi,k} &= \{\langle\langle q, s\rangle, e\rangle \in V_{\Pi,k} \times E : p[e] = q\} \end{aligned}$$

Since $x_{\max}$ is finite, $V_{\Pi,k}$ and $E_{\Pi,k}$ are as well, since at most finitely many extractions have been performed and each extraction relaxes only finitely many transitions. $V_{\Pi,k}$ and $E_{\Pi,k}$ are further limited by the cutoff threshold $c_{\max} \in \mathbb{K}$ as described in section 3.4. In particular, whenever C contains no degenerate cycles, $x_{\max}$ may be set to $kn|Q| \leq kn|\vec{w}| \prod_{i=1}^{|C|} |Q_i|$, where $n = \min\{n \in \mathbb{N} : \forall \pi \in \widetilde{\Pi}(C) : c[\pi^n] >_{\mathbb{K}} c_{\max}\}$ to guarantee both termination and discovery of the $j \leq k$ best paths, although this quantity is considered too large to be of practical use in the dense cascades for which the algorithm was developed, for which the explicit enumeration and storage of Q itself would incur unacceptable computational overhead.

4 Summary

We have presented an algorithm for discovery of the k best output label strings for weighted finite state transducer lookup cascades of arbitrary depth which computes cascade structure online and is therefore suitable for use with "dense" cascades which cannot be pre-compiled. The correctness conditions and running time of the algorithm were discussed for both worst- and average-case scenarios, as were the implications of a greedy termination strategy and an external cutoff threshold. Some properties of degenerate cascades were identified, and the subclass of lookup cascades for which the algorithm is expected to terminate was restricted accordingly. The algorithm as presented here has been implemented in the `gfsmxl` C library (Jurish, 2009), and is being successfully used to implement a robust orthographic standardization cascade for historical German text.

References

Allauzen, C. & M. Mohri (2009). Linear-space computation of the edit-distance between a string and a finite automaton. In: *London Algorithmics 2008: Theory and Practice*. College Publications.

Allauzen, C., M. Riley, J. Schalkwyk, W. Skut & M. Mohri (2007). OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In: J. Holub & J. Zdárek (eds.) *Implementation and Application of Automata,* *12th International Conference, CIAA 2007, Prague, Czech Republic, July 16-18, 2007, Revised Selected Papers*. Springer. 11–23.

Cormen, T. H., C. E. Leiserson, R. L. Rivest & C. Stein (2001). *Introduction to Algorithms (2nd ed.)*. Cambridge, MA: MIT Press and McGraw-Hill.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik* 1. 269–271.

Ésik, Z. & W. Kuich (2004). Equational Axioms for a Theory of Automata. In: C. M. Vide, V. Mitrana & G. Păun (eds.) *Formal Languages and Applications*. (Studies in Fuzziness and Soft Computing, vol. 148). Berlin, Heidelberg: Springer. 183–196.

Fredman, M. L. & R. E. Tarjan (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34. 596–615.

Geyken, A. & T. Hanneforth (2006). TAGH: A Complete Morphology for German based on Weighted Finite State Automata. In: *Finite State Methods and Natural Language Processing, 5th International Workshop, FSMNLP 2005, Revised Papers*. (Lecture Notes in Computer Science, vol. 4002). Berlin: Springer. 55–66.

Jurish, B. (2008). Finding canonical forms for historical German text. In: A. Storrer, A. Geyken, A. Siebert & K.-M. Würzner (eds.) *Text Resources and Lexical Knowledge: Selected Papers from the 9th Conference on Natural Language Processing KONVENS 2008*. Berlin: Mouton de Gruyter. 27–37.

Jurish, B. (2009). *libgfsmxl C library, version 0.0.9*. URL `http://www.ling.uni-potsdam.de/~jurish/projects/gfsm/#gfsmxl`.

Kaplan, R. M. & M. Kay (1994). Regular Models of Phonological Rule Systems. *Computational Linguistics* 20. 331–378.

Laporte, É. (1997). Rational Transductions for Phonetic Conversion and Phonology. In: Roche & Schabes (1997).

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10. 707–710.

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

Mohri, M., F. C. N. Pereira & M. Riley (1996). Weighted Automata in Text and Speech Processing. In: *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended Finite State Models of Language*. Chichester: John Wiley and Sons.

Oflazer, K. (1996). Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics* 22. 73–89.

Oflazer, K. & C. Güzey (1994). Spelling Correction in Agglutinative Languages. In: *ANLP*. 194–195.

Raymond, E. S. (ed.) (2010). *Jargon File version 4.4.7*. URL `http://catb.org/jargon/html/`.

Roche, E. (1997). Parsing with Finite-State Transducers. In: Roche &

Schabes (1997).

Roche, E. & Y. Schabes (eds.) (1997). *Finite-State Language Processing.* Cambridge, MA: MIT Press.

Simon, I. (1987). The Nondeterministic Complexity of Finite Automata. Tech. Rep. RT-MAP-8073, Instituto de Matemática e Estatística da Universidade de São Paulo.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory* . 260–269.

Wagner, R. A. & M. J. Fischer (1974). The String-to-String Correction Problem. *Journal of the ACM* 21. 168–173.