Tokenization as Finite-State Transduction

Marco Cognetta* and Naoaki Okazaki

Department of Computer Science, Institute of Science Tokyo
cognetta.marco@gmail.com, okazaki@c.titech.ac.jp

Tokenization is the first step in modern neural language model pipelines where an input text is converted to a sequence of subword tokens. We introduce from first principles a finite-state transduction framework that can encode all possible tokenizations of a regular language. We then constructively show that Byte-Pair Encoding (BPE) and MaxMatch (WordPiece), two popular tokenization schemes, are also efficiently representable by simple finite-state transducers. For BPE, this is particularly surprising given that it does not tokenize strings from left to right and requires a notion of priority.

We also discuss an application of subword-level pattern promotion to guided generation, where the outputs of a language model are constrained to match a specified pattern, and how tokenization-aware promotion offers a theoretical benefit to modeling.

1. Introduction

Modern neural language models typically operate on *subword* tokens, an intermediate granularity between characters and words, which allows for efficiently encoding any input sequence with a finite vocabulary (Mielke et al. 2021). Recently, using finite-state transducers to produce a regular language of *subword* sequences that match a given regular language of *character* sequences (i.e., the concatenation of the subword sequence would match the character-level pattern) has found an application to *guided generation* for large language models (LLMs) (Willard and Louf 2023; Koo, Liu, and He 2024), where model outputs are constrained to match a schema (e.g., a regular expression). However, these subword-level patterns are typically unaware of any underlying tokenization scheme, which means they do not force the model to adhere to canonical tokenizations—thus ignoring a strong inductive bias that the model was trained with. Here, we consider the problem of generating subword-level regular languages that simultaneously match a character-level pattern, and also only admit subword sequences that would have been produced by a tokenization algorithm such as MaxMatch (Devlin et al. 2019) or Byte-Pair Encoding (BPE) (Gage 1994; Sennrich, Haddow, and Birch 2016).

Our contributions are a clear explanation of how one would construct a subword-level automaton from first principles via automata theory. We first introduce a character-to-subword transducer, which allows us to promote character-level patterns

* Corresponding author.

Action Editor: Liang Huang, Giorgio Satta. Submission received: 10 May 2025; revised version received: 12 August 2025; accepted for publication: 15 August 2025.

https://doi.org/10.1162/COLI.a.23

to subword-level patterns, where the surface form of each accepted subword sequence matches the character-level pattern (Section 5). We extend the framework to two popular subword tokenization algorithms, MaxMatch (Section 6.2) and BPE (Section 6.3), to promote character-level patterns to subword patterns while only allowing subword sequences that match the canonical tokenization.

While the primary focus of this article is the theoretical finite-state tokenization framework, in Section 6.4 we implement our framework[1] and show that a simple automata-theoretic trick offers substantial practical optimizations—for example, a $>$200x speedup for BPE pattern promotion over our standard BPE-promotion construction from Section 6.3. Then, in Section 7, we discuss an application of our framework to subword-level guided generation and a potential shortcoming of prior guided generation approaches that our framework resolves. Specifically, that prior approaches constrain models only to the surface form of the specified pattern without regard to the tokenization, and therefore the strong inductive bias of a model towards the canonical tokenization of the input text is ignored.

Overall, the conceptual power of our framework is evidenced by its use of only foundational automata theory and basic constructions, rather than on ad-hoc algorithms and data structures, making it easy to implement and use—for example, by leveraging existing, well-tested finite-state automata frameworks such as OpenFst (Riley, Allauzen, and Jansche 2009).

2. Related Work

Recently, due to the importance of subword tokenization as the interface between raw text and what a language model processes, many research efforts have worked on formalizing various parts of the tokenization process (Mielke et al. 2021). While some have dealt with the actual usage of subword tokens in LLMs (Gastaldi et al. 2025; Zouhar et al. 2023a; Rajaraman, Jiao, and Ramchandran 2024), others have dealt with the tokenization process directly (Zouhar et al. 2023b; Berglund and van der Merwe 2023; Berglund, Martens, and van der Merwe 2024).

Most relevant to our work (particularly Section 6.3) is the work of Berglund, Martens, and van der Merwe (2024), where the authors describe an algorithm to convert BPE tokenizer's merge rules into a deterministic finite automaton (DFA) that only accepts valid sequences of subwords. They describe an ad-hoc, iterative algorithm that modifies an input automaton step-by-step according to BPE merge list. Their algorithm is based on theoretical properties of BPE uncovered in Berglund and van der Merwe (2023). There, the authors show that BPE tokenization can be done in an incremental manner, as only a constant-sized lookahead window is needed to perform BPE in a streaming fashion. This property implies that a finite-state automaton implementation should be possible. The authors also investigate implementation-specific properties of BPE—specifically, subtle differences in how HuggingFace (Moi and Patry 2023) and SentencePiece (Kudo and Richardson 2018) implement BPE. Vieira et al. (2025) describe a local property of subword sequences that is sufficient to prove that they are canonical BPE sequences. This property can be used to build an automaton that encodes all possible canonical BPE subword sequences, which is the goal of the automaton derived in Sections 6.4 and 6.5.

1 The reference implementation can be found at `https://github.com/mcognetta/tokenization-as-finite-state-transduction`.

Representing subword-level patterns with automata has found use in **guided generation**, where language models are constrained to match some surface pattern (Willard and Louf 2023; Koo, Liu, and He 2024), and has been implemented in a number of libraries such as OUTLINES (Outlines), GUIDANCE (GUIDANCE-AI), SGLANG (Zheng et al. 2024), and XGRAMMAR (Dong et al. 2024). Additionally, (Song et al. 2021) found a connection between the MaxMatch subword tokenization algorithm and the Aho-Corasick pattern matching automaton. Cognetta, Zouhar, and Okazaki (2024) and Geh et al. (2024) use subword-level automata to efficiently sample subword sequences that match a specified pattern for use in language modeling.

Each of these areas—tokenization algorithms, pattern-to-subword promotion, guided generation, etc.—make use of different, specialized algorithms. However, no overarching theory has been developed that subsumes all of them, despite their similarities. Our goal in this article is to provide a single, simple framework that captures the full functionality of tokenization and its extensions.

3. Tokenization

Tokenization algorithms convert sequences of characters into sequences of tokens drawn from a subword vocabulary (e.g., `t␣o␣k␣e␣n␣s` → `token␣s`). A number of tokenization algorithms exist, but we focus on two, MaxMatch (Section 3.2) and BPE (Section 3.3), as these are used in many of the most popular modern LLMs.[2,3]

3.1 Subword Vocabularies

The basis of each of the tokenizers that we consider is the **subword vocabulary**—a set of tokens made up of atomic characters that a tokenizer uses to produce its final output. Given a finite set of atomic characters Σ, a subword vocabulary, Γ, is a finite set $\Sigma \subseteq \Gamma \subset \Sigma^+$. The first set inclusion property ensures that the subword is **open-vocabulary**—any sequence built from atomic characters (i.e., a sequence $w \in \Sigma^+$) can be represented by the subword vocabulary.

Tokenizers map from $\Sigma^* \to \Gamma^*$, but it is often useful to recover the original character sequence. We use $\textsc{Char}_\Gamma(t)$ as the function $\Gamma^* \to \Sigma^*$ that produces the atomic character sequence that spells out the subword sequence t.

3.2 MaxMatch Encoding

MaxMatch (or WordPiece) encoding is a tokenization algorithm that, given a subword vocabulary Γ, tokenizes an input sequence *greedily* from left to right. It iteratively selects the longest possible matching token from Γ and adds that to the tokenized output. While a canonical training procedure exists (Schuster and Nakajima 2012), any subword vocabulary can be used with MaxMatch.

Algorithm 1 describes MaxMatch tokenization inference (Devlin et al. 2019). The naïve implementation of Algorithm 1 takes $O(|w|m)$ time, where $|w|$ is the length of the input sequence and $m = \max_{v \in \Gamma} |v|$ is the length of the longest token in Γ. However,

2 MaxMatch: BERT (Devlin et al. 2019).

3 BPE: ROBERTA (Liu et al. 2019), LLAMA (Touvron et al. 2023a,b; Grattafiori et al. 2024), GEMMA (Gemma Team et al. 2024a, b; Gemma Team et al. 2025), GPT (Brown et al. 2020), QWEN (Yang et al. 2025).

Algorithm 1: MaxMatch Tokenization
Input: Vocabulary Γ, String $w \in \Sigma^+$
Output: Tokenized sequence $t \in \Gamma^+$

```
1: i ← 0, t ← ⟨ ⟩                          ▷ The starting index and output tokenization list, respectively.
2: m ← max_{v∈Γ} |v|                       ▷ The length of the longest token in Γ (to bound the iteration length).
3: while i < |w| do                        ▷ Until we reach the end of the string...
4:     for j ∈ 1...m do
5:         if w_{i:i+j} ∈ Γ then           ▷ Find the longest subsequence starting at i that is in Γ.
6:             z ← w_{i:i+j}
7:     APPEND(t, z)                        ▷ Add the token to the output.
8:     i ← i + |z|                         ▷ Skip to the end of that subsequence.
9: return t
```

Step							
0.	b	a	n	a	n	a	s
1.	b a	n	a	n	a	s	
2.	b a	n	a	n a	s		
3.	b a	n	a	n a	s		

Figure 1
An example of MaxMatch tokenization of `bananas` with $\Gamma = \{$`a, b, n, s, ba, na, ban, bana`$\}$. At each step, the longest matching token is added to the tokenized sequence to produce `bana␣na␣s`.

(Song et al. 2021) show that MaxMatch tokenization can be viewed as a variant of the Aho-Corasick algorithm (Aho and Corasick 1975) and describe an $O(|w|)$-time tokenization algorithm. Figure 1 provides an example of MaxMatch tokenization inference.

3.3 Byte-Pair Encoding

Byte-Pair Encoding (BPE) is a two-stage algorithm for producing subword tokenizations (Gage 1994; Sennrich, Haddow, and Birch 2016; Zouhar et al. 2023b) by iteratively *merging* adjacent tokens into a single, larger token.

In this article, we assume that we have already-trained BPE tokenizers and do not rely on the training procedure directly for any proofs, definitions, etc., after this section. However, we include the algorithm pseudocode here to reinforce the point that the order in which merges happen is not arbitrary.

To train a BPE tokenizer, first, a subword vocabulary is formed, given a corpus C and a desired vocabulary size k. Assume $\textsc{Count}(C, a, b)$ counts the number of occurrences of the sequence ab in corpus C and $\textsc{Apply}(T, (a, b))$ applies a "merge" (a, b) in a text T by merging all instances of the subsequence a, b to a single token ab. Then, Algorithm 2 produces the vocabulary and merge list for the BPE tokenizer by iteratively

Algorithm 2: BPE Training
Input: Corpus C over alphabet Σ, Target merge size k
Output: Vocabulary Γ, Merge list μ

$\Gamma = \Sigma,\ \mu = \langle\ \rangle$ ▷ *The initial vocabulary and merge list, respectively.*
for $i \in 1 \dots k$ **do**
  $(a, b) = \arg\max_{(a,b)\in\Gamma^2} \textsc{Count}(C, a␣b)$ ▷ *The most frequent cooccuring pair in C.*
  $\textsc{Append}(\mu, (a, b))$ ▷ *Add merge information to μ (which is ordered by priority).*
  $\textsc{Append}(\Gamma, ab)$ ▷ *Update the token vocabulary.*
  $\textsc{Apply}(C, (a, b))$ ▷ *Merge all instances of $a␣b$ to ab in C.*
return (Γ, μ)

Algorithm 3: BPE Tokenization
Input: BPE Tokenizer $\mathcal{B} = (\Gamma, \mu)$, String $w \in \Sigma^+$
Output: Tokenized sequence $t \in \Gamma^+$

$t \leftarrow w$ ▷ *The initial character sequence, interpreted as tokens.*
$\psi \leftarrow \langle (t_i, t_{i+1}) \mid (t_i, t_{i+1}) \in \mu \rangle$ ▷ *The set of all current possible merges.*
while $\psi \neq \emptyset$ **do**
  $(a, b) \leftarrow \arg\max_{\mu} \psi$ ▷ *The highest priority merge according to the ordering in μ.*
  $t \leftarrow \textsc{Apply}(t, (a, b))$ ▷ *Apply the merge and update t.*
  $\psi \leftarrow \langle (t_i, t_{i+1}) \mid (t_i, t_{i+1}) \in \mu \rangle$ ▷ *The new list of possible merges.*
return t

finding the pair a, b with the highest cooccurence count in the corpus, merging them (via Apply), adding the merge (a, b) to an ordered list of merges μ, and adding the token ab to the subword vocabulary Γ. This repeats until the target vocabulary size is reached. We refer to this as $\mathcal{B} = (\Gamma, \mu)$.

A trained BPE tokenizer $\mathcal{B} = (\Gamma, \mu)$ can then be used to produce a tokenized sequence from an input character sequence, according to the merge rules. Given a sequence, BPE iteratively finds the highest priority (the lowest index in the merge list) merge that is present in the current (partially) tokenized sequence and merges it. This is done repeatedly until no more merges are available, at which point the tokenized sequence is returned, as shown in Algorithm 3.

Figure 2 gives an example of each merge being iteratively applied to an input string. Notice that successive merges do not necessarily happen in left-to-right order, and that the merging procedure resembles a tree.

3.4 UnigramLM

UnigramLM is another tokenization algorithm that is designed to be stochastic in order to provide *subword regularization*, a technique where language models are exposed to many different tokenizations of the same input text as a form of data augmentation and to add robustness to the model (Kudo and Richardson 2018). The reference implementation of UnigramLM is found in SentencePiece (Kudo and Richardson 2018).

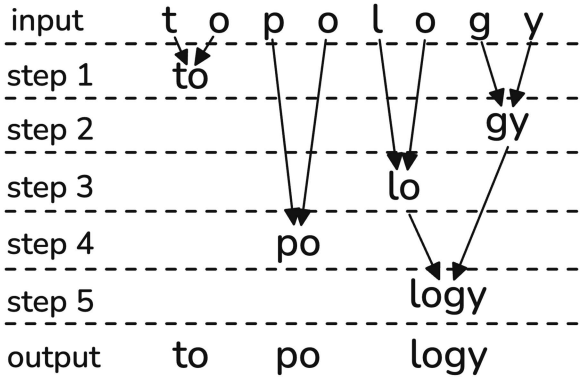


Figure 2
An example of BPE tokenization given $\mu = \langle$(t, o), (g, y), (l, o), (p, o), (lo, gy)$\rangle$. Notice that the merges are not necessarily done left to right or in order of length. The final tokenized sequence is to␣po␣logy.

UnigramLM constructs a subword vocabulary and a corresponding probability distribution over subwords as its initialization.[4] Then the set of all possible tokenizations of an input are scored with the subword probability distribution (i.e., given a tokenization of the input, multiply the probabilities of each subword in the tokenization according to the constructed probability distribution). Tokenizations are then either sampled or the highest probability tokenization is found and used as the final tokenization.

We do not cover UnigramLM in detail in this article for two reasons: First, its tokenization algorithm is a graph search algorithm rather than a deterministic construction, and second, it has a straightforward interpretation under our framework. Specifically, the result of Section 5 provides the exact lattice that would then be scored by the probability distribution and then searched over.

4. Automata and Transducers

We introduce the fundamental concepts from automata theory that will be used in the main results of this article. In general, we provide canonical pseudocode only for the definitions which we will draw on later in proofs, and provide only the definitions for the remaining concepts. In this article, we deal only with *unweighted* automata and transducers (i.e., automata and transducers over the Boolean semiring), so we use logical operators ($\wedge$ and $\vee$) where needed. We additionally omit weights from arcs; if the arc exists, it is assumed to have weight one (TRUE in the Boolean semiring).

Definition 4.1: Finite-State Transducers. These are a generalization of finite automata, which instead of accepting or rejecting strings, accept or reject *pairs* of strings, (x, y), where x is the *input* string and y is the *output* string. In simpler terms, they accept or reject *transformations* of strings, where the input is transformed into the output.

A transducer is defined as $\mathcal{T} = (\Sigma, \Gamma, Q, q_{\text{start}}, F, \delta)$ where Σ and Γ are the finite *input* and *output* vocabularies, Q is a set of states, $q_{\text{start}} \in Q$ is the *initial* state, $F \subseteq Q$ is a set of

4 We do not cover UnigramLM's implementation here as it is both complex and unrelated to the remaining topics in this article.

final states, and δ is a set of *transitions* (or *arcs*) $Q \times \Sigma \times \Gamma \times Q$ (an initial state, an input character, an output character, and a target state). We denote the *empty string* as ε.

A **path** is a sequence of transitions

$$\pi = (q_0, i_0, o_0, q_1),\ (q_1, i_1, o_1, q_2),\ \ldots,\ (q_{n-1}, i_{n-1}, o_{n-1}, q_n),$$

where $(q_i, i_{i+1}, c_{i+1}, q_{i+1}) \in Q \times \Sigma \times \Gamma \times Q$. Let s_i and s_o be functions that concatenate the input and output words (the elements from Σ and Γ) of a path, respectively. Then, a path is denoted $\pi_{x,y}$ if $s_i(\pi) = x$ and $s_o(\pi) = y$. A path $\pi_{x,y}$ is called **accepting** if for all transitions in the path sequence, $(q_i, i_i, o_i, q_{i+1}) \in \delta$, $q_0 = q_{\text{start}}$, and $q_n \in F$, and is called rejecting otherwise. Note that ε can appear in a transition (as either the input or output character or both) but does not "consume" a character from the input or output strings since ε is an empty character (e.g., $ab\varepsilon c = abc$). We use π_x for the analogous path definition for automata.

We write the functional form of a transducer as $\mathcal{T}(x, y)$ for $x \in \Sigma^*$ and $y \in \Gamma^*$. This function returns true if there exists an accepting path $\pi_{x,y}$ and false otherwise.

Definition 4.2: Finite-State Automata. Finite-state automata (or, just *automata*) are a special case of transducers where $\Gamma = \Sigma$ and $\forall (q, i, o, p) \in \delta$, $i = o$. Consequently, automata are transducers which map strings to themselves. As such, we omit the output label from the transition definition where clear; i.e., (q, i, p) rather than (q, i, i, p). An automaton is **deterministic** if there are no ε-transitions and, for all states and for each character in the automaton's alphabet, there is at most one outgoing transition from that state labeled with that character.

An automaton is **trim** if for each state, there is a path from q_0 to that state and a path from that state to some state $q_f \in F$ (accessibility and co-accessibility, respectively). Automata can be converted to an equivalent trim representation in polynomial time (Sipser 1997).

Like transducers, we write $\mathcal{A}(x)$ for $x \in \Sigma^*$ as the functional form of automata, returning true if there is an accepting path π_x and false otherwise.

For convenience, we use a functional-notation for δ when clear for deterministic automata: $\delta(q, c) := p$ if $(q, c, p) \in \delta$. We denote the *language* of a transducer, the set of pairs of strings accepted by it, as $\mathcal{L}(\mathcal{T}) = \{(x, y) \mid (x, y) \in \Sigma^* \times \Gamma^* \wedge \mathcal{T}(x, y)\}$ or $\mathcal{L}(\mathcal{T})$, respectively. Likewise, for automata, we write $\mathcal{L}(\mathcal{A}) = \{x \mid x \in \Sigma^* \wedge \mathcal{A}(x)\}$.

Notational Aside. We make use of several notational shortcuts when convenient. When referring to states, we write $q \in Q$ or $q \in \mathcal{A}$ or $\mathcal{T}$ interchangeably (for automata and transducers, respectively). Likewise, $|\mathcal{A}|$ and $|\mathcal{T}|$ are defined as the number of states in the automaton or transducer.

For transitions, we often treat δ as a (partial) function:

- $\delta(q, i, o, p)$ is a transition, used interchangeably with $\delta(q, i, o) = p$ since all transducers in this article are deterministic
- $\delta(q)$ is the set of transitions $\{(q, i, o, p) \mid (q, i, o, p) \in \delta\}$

Definition 4.3: Label Projection. Suppose we have a transducer $\mathcal{T} = (\Sigma, \Gamma, Q, q_{\text{start}}, F, \delta)$. The unary output projection function PROJ transforms the transducer into a new transducer $\mathcal{A} = (\Gamma, \Gamma, Q, q_{\text{start}}, F, \delta')$, where Γ, Q, q_{start}, and F remain the same, but

$\delta' = \{(q, o, p) \mid (q, i, o, p) \in \delta\}$. This has the effect of transforming a transducer into an automaton that computes:

$$\textsc{Proj}(\mathcal{T}) = \{y \in \Gamma^* \mid \exists x \in \Sigma^* \text{ s.t. } (x, y) \in \mathcal{T}\}. \tag{1}$$

That is, the set of all strings $y \in \Gamma^*$ which have a valid transduction from some string $x \in \Sigma^*$.

Definition 4.4: Transducer Composition. A fundamental binary operation on transducers is composition. Let $\mathcal{T}_1 = (\Sigma, \Xi, Q_1, q_{\text{start}}, F_1, \delta_1)$ and $\mathcal{T}_2 = (\Xi, \Gamma, Q_2, q'_{\text{start}}, F_2, \delta_2)$ be transducers with a shared alphabet Ξ on the output side of $\mathcal{T}_1$ and the input of $\mathcal{T}_2$. Then, composition $\circ$ produces a new transducer that recognizes the relation:

$$[[\mathcal{T}_1 \circ \mathcal{T}_2]](x, z) = \{(x, z) \in \Sigma^* \times \Gamma^* \mid \exists y \in \Xi^* \text{ s.t. } (x, y) \in \mathcal{T}_1 \wedge (y, z) \in \mathcal{T}_2\}.$$

That is, the set of pairs of strings $(x, z) \in \Sigma^* \times \Gamma^*$ where there is an intermediate string $y \in \Xi^*$ such that $\mathcal{T}_1$ accepts (x, y) and $\mathcal{T}_2$ accepts (y, z).

The composition algorithm proceeds like automata intersection (indeed, intersection is just a special case of composition, where both inputs are automata). For each state $q_i, q'_j \in Q_1 \times Q_2$, the composed transducer has a state q_{ij}. For each $(q_i, c_i, c_o, p_j) \in \delta_1$ and $(q'_k, c'_i, c'_o, p'_l) \in \delta_2$ such that $c_o = c'_i$, there is a transition $(q_{ij}, c_i, c'_o, p_{jl})$. The initial state is $(q_{\text{start}}, q'_{\text{start}})$ and the final states are all q_{ij} such that $(q_i, q'_j) \in F_1 \times F_2$.

Algorithm 5 in Appendix A gives pseudocode for generic transducer composition (Allauzen and Mohri 2009).[5]

5. Tokenization-Agnostic Pattern Promotion

5.1 Character-to-Subword Transducer

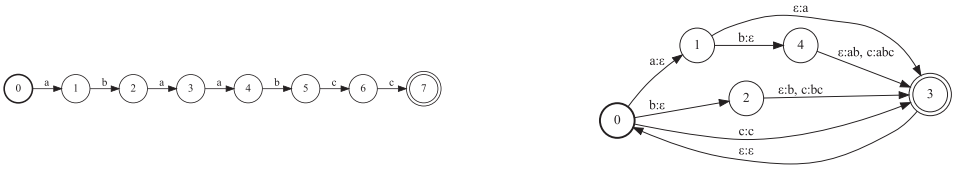
We now introduce the key ingredient to our framework: the character-to-subword transducer. This is a transducer that maps character sequences to subword units, e.g., `t␣o␣k␣e␣n` → `tok␣en`. The useful aspect of this transducer is that it can produce a succinct representation of *all* possible subword tokenizations of a given input, according to a subword vocabulary, Γ. That is, if $\{\texttt{tok}, \texttt{to}, \texttt{en}, \texttt{k}\} \subseteq \Gamma$, then it will encode the mappings to both `tok␣en` and `to␣k␣en`.

Building a Character-to-Subword Transducer. A character-to-subword transducer recognizes the relation:

$$\mathcal{L}(\mathcal{T}) = \bigcup_{w \in \Gamma} (w_1 \text{␣} w_2 \text{␣} \ldots \text{␣} w_n, w) \tag{2}$$

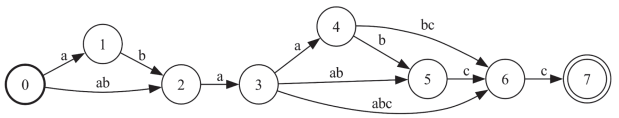
which is the set of transductions from the character spelling of a word (w_1␣w_2␣…␣w_n, where $w_i \in \Sigma$) to the single-token representation of the word $w \in \Gamma$.

5 In general, generic composition is restricted to ε-free transducers, but it also applies to other classes of transducers (e.g., over idempotent semirings and without ε-cycles, which is true for the class of transducers considered in this article).



(a) An input pattern automaton $\mathcal{A}$. (b) A character-to-subword transducer $\mathcal{T}$.

(c) $\mathcal{A} \circ \mathcal{T}$ (d) $\textsc{Proj}(\mathcal{A} \circ \mathcal{T})$



(e) $\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T}))$

Figure 3
An example of projecting a character-level input pattern $\mathcal{A} =$ `abaabcc` to the subword level, given a subword vocabulary {`a, b, c, ab, abc, bc`} represented by the character-to-subword transducer $\mathcal{T}$. The intermediate transducers formed during this process are shown in panels (c) and (d), and the final, minimized subword automaton is given in panel (e). Observe that for every accepting path in $\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T}))$, the concatenation of the subwords on that path satisfy the pattern in $\mathcal{A}$ when spelled out character-by-character. For example, `ab␣a␣a␣bc␣c` and `a␣b␣a␣abc␣c`, which are accepted by $\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T}))$, both correspond to `abaabcc`, which is accepted by $\mathcal{A}$.

A simple way to build a character-to-subword transducer is to construct a trie transducer that maps characters to subwords and then (optionally) minimize it. The final result is an acyclic, deterministic transducer for which each valid path encodes a unique word in the subword vocabulary, and ε-closure allows it to transduce arbitrarily long sequences (Cognetta, Allauzen, and Riley 2019).[6]

Character-Level To Subword-Level Patterns. We now have all the pieces to construct a subword-level automaton that matches a given character-level pattern. Let $\mathcal{T}$ be a character-to-subword transducer over a subword vocabulary Γ (produced by Algorithm 6) and let $\mathcal{A}$ be a trim, deterministic automaton over Σ^* recognizing some given pattern. The composition $\mathcal{A} \circ \mathcal{T}$ produces a character-sequence-to-subword-sequence transducer, for which for any path $\pi_{x,y}$, $s_i(\pi) = \textsc{Char}_\Gamma(s_o(\pi))$. This is demonstrated in Figure 3c. Thus, the concatenation of the output labels on any accepting path spells out a character sequence which would have been accepted by $\mathcal{A}$. In fact, this transducer encodes *all* subword sequences with this property. We then compute:

$$\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T})) \tag{3}$$

6 Algorithm 6 in Appendix B gives a pseudocode implementation that is used in the proofs found there.

which results in a minimal, deterministic automaton recognizing subword sequences which spell out character sequences that satisfy $\mathcal{A}$. Lemma 1 shows that a determinization step, which can take exponential time in the size of the automaton, is not necessary between the PROJ and MIN steps.

Lemma 1
Let $\mathcal{A}$ be minimal, deterministic automaton over Σ and $\mathcal{T}$ be a character-to-subword transducer over $\Sigma \subseteq \Gamma \subset \Sigma^+$ (Equation (2)). Then ε-REMOVAL(PROJ($\mathcal{A} \circ \mathcal{T}$)) is deterministic.

Lemma 1 implies MIN(PROJ($\mathcal{A} \circ \mathcal{T}$)) can be computed in polynomial time in the size of $\mathcal{A}$ and $\mathcal{T}$. First, $\mathcal{A} \circ \mathcal{T}$ can be computed in polynomial time by the standard transducer composition algorithm (Allauzen and Mohri 2009). Likewise, PROJ can be done in $O(|\delta|) \in O(|\mathcal{A}||\Gamma|)$ time. Since PROJ($\mathcal{A} \circ \mathcal{T}$) produces an *automaton* over Γ^*, the standard ε-Removal and MIN algorithms can be used, which both have polynomial runtimes the size of the automaton (Sipser 1997).

Figure 3 gives an example of each of these stages for a given pattern and character-to-subword transducer.

6. Tokenization-Preserving Pattern Promotion

The subword-level pattern promotion described in Section 5 allows one to form an automaton that encodes all subword sequences which, when concatenated, would match the character-level input pattern. These subword sequences are simply drawn from the subword vocabulary, and do not reflect the underlying tokenization scheme.

For example, given a pattern `a␣b␣a␣a␣b` and a vocabulary $\Gamma = \{\texttt{a}, \texttt{b}, \texttt{ab}, \texttt{aba}\}$, a tokenization-agnostic subword-level automaton would accept the sequences `a␣b␣a␣a␣b`, `ab␣a␣ab`, etc. However, a MaxMatch tokenizer would always output the tokenization `aba␣ab` when given this input. This motivates us to determine if we can simultaneously promote character-level patterns to subword-level patterns *and* preserve the underlying tokenization scheme.

Formally, let $t \in \Gamma^+$ be a sequence of subwords. Then, rather than the tokenization agnostic:

$$\{t \mid \textsc{Char}_\Gamma(t) \in \mathcal{L}(\mathcal{A})\},$$

we want to produce the language:

$$\{t \mid \textsc{Char}_\Gamma(t) \in \mathcal{L}(\mathcal{A}) \wedge T(\textsc{Char}_\Gamma(t)) = t\},$$

where T is a BPE or MaxMatch tokenizer. In other words, we want to only accept the set of subword sequences that correspond to the tokenizations of character sequences which match $\mathcal{A}$ as opposed to the set of all possible subword sequences which match the character-level pattern.

On the surface, this seems difficult, as the pattern could describe an infinite set of strings each with a unique canonical tokenization, so we cannot possibly enumerate them all. However, here we show that both MaxMatch and BPE tokenization can be expressed as finite state transducers. This allows us to use generic finite-state transduction operations such as composition and projection to convert character-level automata to tokenization-preserving subword-level automata efficiently.

Representing BPE as a finite-state transduction is a particularly surprising result. As shown in Figure 2, the merge ordering of BPE inference resembles a tree, which suggests that a string finite-state transducer not be powerful enough to encode the inference algorithm. Additionally, BPE requires a notion of priority and an unbounded, stack-like data structure and BPE merges are not performed left to right, while transducers have constant memory, have no notion of priority, and operate in strictly left-to-right order.

6.1 φ-Transitions

We first introduce a key component used in the constructions for both MaxMatch and BPE transducers. Automata and transducers can be augmented with a special φ symbol used to denote **failure** transitions, in the sense of the Knuth-Morris-Pratt algorithm (KMP) (Knuth, Jr., and Pratt 1977) *failure function* or the Aho-Corasick algorithm *failure transition* (Aho and Corasick 1975). These algorithms use failure functions to efficiently move to another state in an automaton (or trie) while doing string pattern matching in order to not have to perform redundant calculations. However, more generally, in automata theory φ-transitions (transitions marked with the label φ) are transitions that can only be traversed when no other matching transition is available. The presence of φ-transitions does not change the computational power of automata or transducers, but can allow for more succinct and clear representations of them (Allauzen and Riley 2018).

A φ-transition from a state q is a transition (q, φ, o, p) that acts like an ε transition in that it does not consume an input symbol, but can only be traversed if the current symbol that we are looking for is not present in any other transition originating from q. In other words, they are a "fallback" transition—if no other matching transition exists, then a φ-arc can be traversed (Svete et al. 2022).

6.2 MaxMatch-Preserving Pattern Promotion

We begin with MaxMatch tokenization, which is rooted in automata theory and is thus somewhat more straightforward to understand intuitively. We follow Song et al. (2021), and build a character-to-subword transducer in the style of the Aho-Corasick automaton (Aho and Corasick 1975).

Roughly, given a vocabulary Γ, the standard Aho-Corasick automaton allows one to find all substrings where an input string matches an item from the vocabulary in time linear to the input length. This is done by including a number of **failure arcs** (φ-transitions) in the automaton—arcs that map a state (which corresponds to a prefix of a token) to the state that corresponds to the longest matching *suffix* of that token which is also a prefix of another word in the vocabulary, and which can only be traversed if no suitable character transition exists at the current state. The failure arcs allow for full-token prefixes of the current token to be marked as matched, while moving to a prefix of another token that is currently valid, without recomputation.

Song et al. (2021) show that this construction can be slightly modified to produce only matches which are greedily as long as possible, which corresponds exactly to the MaxMatch (WordPiece) tokenization inference algorithm. They achieve this by augmenting the failure arcs (referred to as **failure links**) with a "popping" mechanism, which pops the longest prefix token(s) that have already been matched before moving to the suffix state via the failure arc. Like the original Aho-Corasick algorithm, failure arcs and links can be precomputed and stored for fast inference. The authors note that,

despite using the algorithmic form of the Aho-Corasick algorithm, their approach can also be viewed as an application of finite-state transducers.

Algorithm 7 (Appendix C) is a simplified version of the failure-trie precomputation algorithm (Song et al. 2021) and Algorithm 8 (Appendix C) shows how to convert the failure-trie into a character-to-subword transducer, $\mathcal{T}_{Aho}$.[7] The structure of the resulting transducer is the same as the Aho-Corasick automaton, where non-failure arcs (corresponding to characters) have input and output labels $c \in \Sigma$ and failure arcs (corresponding to words) have input label φ and output label $w \in \Gamma$.

$\mathcal{T}_{Aho}$ is a valid character-to-subword transducer in that each accepting path corresponds to a sequence of characters from Σ on the input and a sequence of subwords from Γ on the output such that the concatenation of the subwords is the same as the character sequence. However, differently from the character-to-subword transducers from Section 5.1, given an input automaton $\mathcal{A}$ to compose with $\mathcal{T}_{Aho}$, there is a one-to-one correspondence between sequences accepted by $\mathcal{A}$ and sequences accepted by $\mathcal{T}_{Aho}$ which corresponds to the greedy parse.

As such, $\mathcal{T}_{\text{Aho}}$ can be composed with an input pattern $\mathcal{A}$ using the φ-transition semantics and projected to an automaton over Γ via:

$$\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T}_{\text{Aho}}))$$

as in Equation (3). The result is a subword automaton which only matches subword sequences that would match the character pattern in $\mathcal{A}$ *and* are a greedy tokenization.

Figure 4 gives an example of an input automaton composed with a standard character-to-subword transducer (as described in Section 5.1) and with a MaxMatch transducer.

6.3 BPE-Preserving Pattern Promotion

We now turn to BPE, and attempt to construct a transducer similar to $\mathcal{T}_{Aho}$, but which produces a subword-level automaton that matches the tokenizations induced by a BPE tokenizer $\mathcal{B} = (\Gamma, \mu)$.

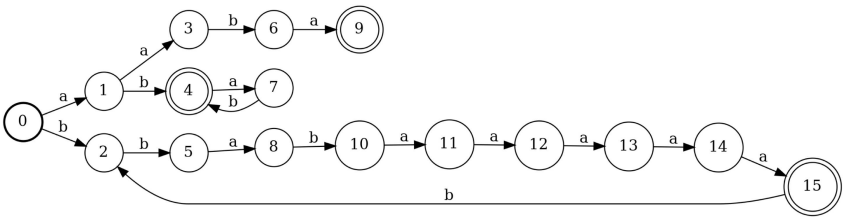
Recall Algorithm 3, the canonical algorithm for BPE tokenization. This algorithm processes the input string in arbitrary order (in that it does not necessarily form the final tokenized sequence from the left to the right, but can skip back and forth) and resembles a context-free grammar parse tree, and has a notion of priority. Each of these suggest that the relation described by BPE is not regular and cannot be represented by a finite-state transducer.

However, consider the following alternative implementation of BPE:

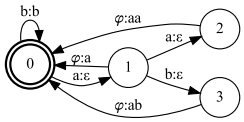
Rather than selecting the highest priority merge available, Algorithm 4 starts from the character-level string and applies *every* merge in order of priority and produces an identical result to Algorithm 3 (Zouhar et al. 2023b). Intuitively, $(\texttt{ab}, \texttt{c}) \rightarrow \texttt{abc}$ can only be meaningfully applied if the merge $(\texttt{a}, \texttt{b}) \rightarrow \texttt{ab}$ was already applied.

This alternative algorithm suggests a finite-state transducer representation is possible—we can simulate BPE by forming, for each $(\texttt{a}, \texttt{b}) \in \mu$, a transducer that maps $\texttt{a␣b} \rightarrow \texttt{ab}$. We call such a transducer a merge **gadget**, $G_{(\texttt{a},\texttt{b})}$. The structure of these gadgets is simple, as shown in Figure 5.

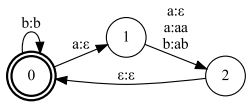
7 Algorithms 7 and 8 are given in Appendix C for space reasons, as they are not crucial to the results of this article.



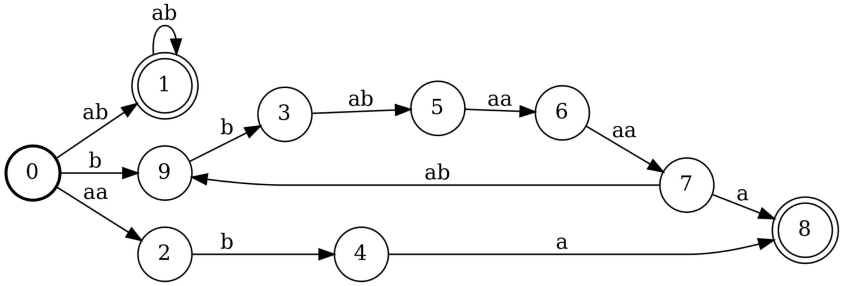
(a) A character pattern automaton $\mathcal{A}$.



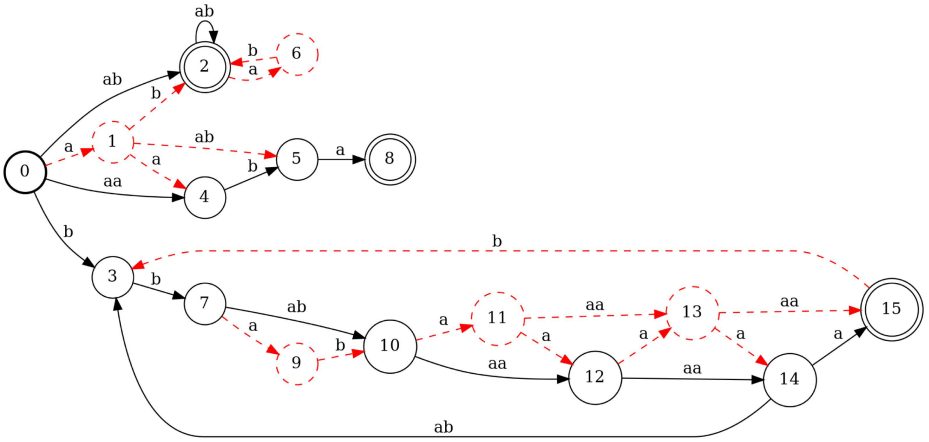
(b) The MaxMatch-Preserving Transducer $\mathcal{T}_{\text{Aho}}$



(c) The Tokenization-Agnostic Transducer $\mathcal{T}$



(d) MIN(PROJ($\mathcal{A} \circ \mathcal{T}_{Aho}$))



(e) MIN(PROJ($\mathcal{A} \circ \mathcal{T}$))

Figure 4
A character-level automaton $\mathcal{A}$ is composed with subword lexicons over {`a`, `b`, `aa`, `ab`} represented by the MaxMatch-preserving transducer $\mathcal{T}_{Aho}$, and the tokenization-agnostic transducer $\mathcal{T}$, shown in Figures (b) and (c), respectively. The results of the composition are shown in (d) and (e). In (e) specifically, arcs and states that appear in the unconstrained automaton but would not appear in the constrained automaton (since they do not encode greedy maximal matches) are shown in dashed-red.

Algorithm 4: Iterative BPE
Input: Word $w \in \Sigma^+$, BPE Tokenizer $\mathcal{B} = (\Gamma, \mu)$
Output: Tokenized sequence $t \in \Gamma^+$

1: **for** $(a, b) \in \mu$ **do**
2: $\quad w \leftarrow \textsc{Apply}(w, (a, b))$
3: **return** w

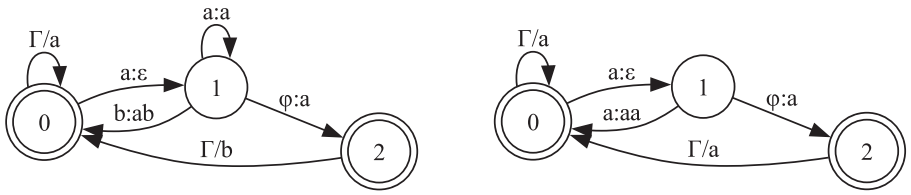
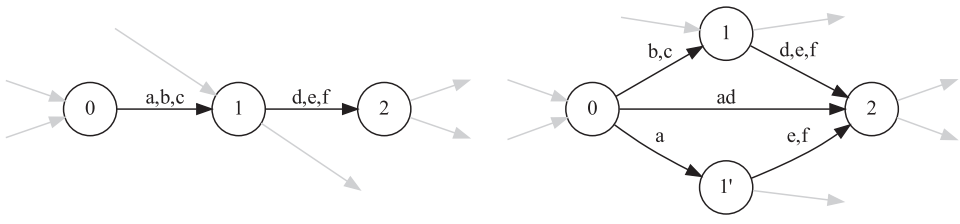


Figure 5
Merge gadgets $G_{(\mathtt{a},\mathtt{b})}$ and $G_{(\mathtt{a},\mathtt{a})}$ for the merges $(\mathtt{a}, \mathtt{b}) \rightarrow \mathtt{ab}$ and $(\mathtt{a}, \mathtt{a}) \rightarrow \mathtt{aa}$, respectively. All arcs that don't have an output symbol are assumed to be of the form (q, c, c, p).

For simplicity, we focus on merges of the form $(\mathtt{a}, \mathtt{b})$ (i.e., the component tokens of the merge are different), but merges of the form $(\mathtt{a}, \mathtt{a})$ have a nearly identical structure. A gadget contains three states: the *initial* state (q_0), the *active* state (q_1), and the *abort* state (q_2). For a merge gadget $G_{(\mathtt{a},\mathtt{b})}$, the initial state contains a self loop (q_0, c, c, q_0) for each $c \in \Gamma$ such that $c \neq \mathtt{a}$. It also contains an arc to the active state $(q_0, \mathtt{a}, \varepsilon, q_1)$. The active state contains three arcs, one which resolves the merge $(q_1, \mathtt{b}, \mathtt{ab}, q_0)$, one which postpones the merge $(q_1, \mathtt{a}, \mathtt{a}, q_1)$, and one which cleans up the merge if we read a character other than b or have nothing else to consume, $(q_1, \varphi, \mathtt{a}, q_2)$. The reason for φ in the last arc is that, when reading the initial $\mathtt{a}$, we do not emit an a in case we will form a merge. The self-loop on state q_1 consumes and produces $\mathtt{a}$'s, but if we fail to make a match, we need to produce one additional $\mathtt{a}$ to make up for the one that was not generated on the arc from q_0 to q_1. The abort state contains an arc $(q_2, c, c, q_0), \forall c \neq \mathtt{b}$, which allows us to go back to the start state after failing to make a match at the active state.

Figure 6 gives an example of an application of a merge gadget to an automaton.

To build intuition, suppose we have an automaton $\mathcal{A}$ representing the input string $\mathtt{b␣c␣a␣b␣a␣b␣c␣c}$, a BPE tokenizer $\mathcal{B} = (\Gamma, \mu)$, where $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{ab}, \mathtt{bc}, \mathtt{cc}, \mathtt{abc}\}$ and $\mu = [(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{c}), (\mathtt{ab}, \mathtt{c})]$, and a list of gadgets $\mathcal{G} = [G_{(\mathtt{a},\mathtt{b})}, G_{(\mathtt{b},\mathtt{c})}, G_{(\mathtt{c},\mathtt{c})}, G_{(\mathtt{ab},\mathtt{c})}]$.



(a) A subset of an automaton with a path $\mathtt{a␣d}$. (b) After composition with $G_{(\mathtt{a},\mathtt{d})}$.

Figure 6
An example of how composing with a merge gadget modifies the input automaton.

After applying $\mathcal{A} \circ G_{(\texttt{a,b})}$, we have a new transducer where the input is the sequence `b␣c␣a␣b␣a␣b␣c␣c` but the output is the sequence `b␣c␣ab␣ab␣c␣c`, and all instances of `a␣b` have been merged to `ab`. Repeating this for each of the merge gadgets results in

$$\mathcal{A} \circ G_{(a,b)} \circ G_{(b,c)} \circ G_{(c,c)} \circ G_{(ab,c)}$$

which transduces `b␣c␣a␣b␣a␣b␣c␣c` to `bc␣ab␣ab␣cc`, the exact sequence produced by $\mathcal{B}$(`bcababcc`). Written out another way, we can implement BPE by computing the composition:

$$\textsc{Min}\left(\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right) \tag{4}$$

Figure 7 gives an example of BPE-preserving pattern promotion given a BPE tokenizer.

6.3.1 Runtime. Via gadgets, we have shown that regular languages are closed under BPE transduction. The naïve application of the full merge composition in Equation (4) takes $O(|\mathcal{A}| \prod_{m\in\mu} |G_m|) = O(3^{|\mu|}|\mathcal{A}||\Gamma|)$ time (Allauzen and Mohri 2009). For a BPE tokenizer with just 30 merges, this would be prohibitively expensive, let alone a tokenizer with $64k$ merges.

However, we find that the actual runtime is $\textsc{Poly}(|\mathcal{A}|, |\mu|, |\Gamma|)$. In particular, we bound the size of the resulting automaton and show that each consecutive merge can be performed in polynomial time in the size of the input pattern automaton. The proof is moved to Appendix D due to the length.

Theorem 1
Given an automaton $\mathcal{A}$ and a series of merges $m_1, m_2, \ldots, m_k \in \mu$. Let $\mathcal{A}' = \textsc{Min}(\textsc{Proj}(\mathcal{A} \circ G_{m_1} \circ G_{m_2} \circ \cdots \circ G_{m_{k-1}}))$. Then, $\textsc{Min}(\textsc{Proj}(\mathcal{A}' \circ G_{\mu_k}))$ can be computed in $\textsc{Poly}(|\mathcal{A}|, k, |\Gamma|)$ time.

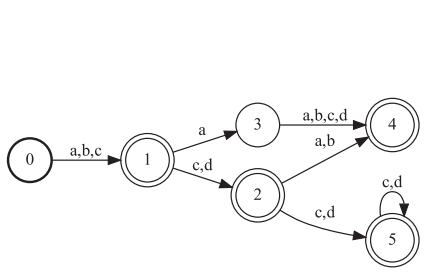
Similarly to Theorem 6.3.1, the BPE DFA constuction algorithm by Berglund, Martens, and van der Merwe (2024) also runs in polynomial time—the authors prove a bound of $O(|\mathcal{A}||\Gamma|\mu|^2)$.

Importantly, Theorem 6.3.1 (and also the result from Berglund, Martens, and van der Merwe [2024]) place polynomial bounds on the size of the resulting automaton.

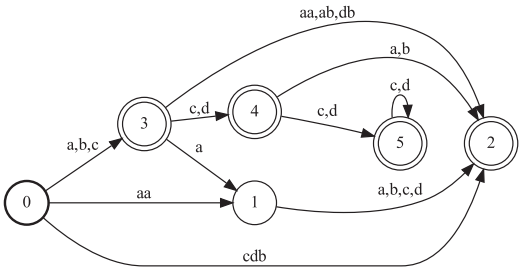
Corollary 1
The size of

$$\textsc{Min}\left(\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right)$$

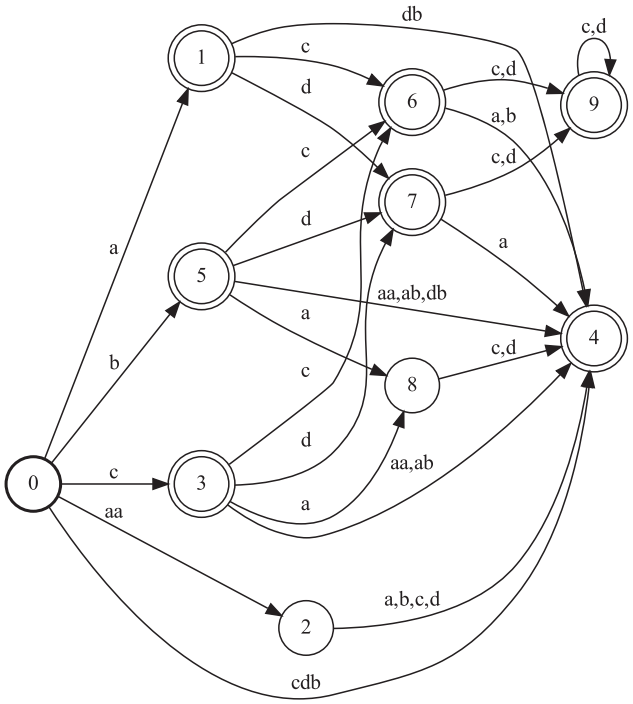
is $\in \textsc{Poly}(|\mathcal{A}|, |\mu|, |\Gamma|)$.



(a) A pattern automaton $\mathcal{A}$.



(b) Tokenization Agnostic MIN(PROJ($\mathcal{A} \circ \mathcal{T}$))



(c) MIN(PROJ($\mathcal{A} \circ G_{(a,a)} \circ G_{(a,b)} \circ G_{(d,b)} \circ G_{(c,db)}$))

Figure 7
An example of BPE-preserving pattern promotion of an automaton $\mathcal{A}$ for a BPE tokenizer $\mathcal{B}$ with $\Gamma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{aa}, \texttt{ab}, \texttt{db}, \texttt{cdb}\}$ and $\mu = \langle(\texttt{a}, \texttt{a}), (\texttt{a}, \texttt{b}), (\texttt{d}, \texttt{a}), (\texttt{c}, \texttt{db})\rangle$. Any path in the BPE-preserving pattern automaton (c) matches the pattern from $\mathcal{A}$ and corresponds to the tokenization produced by $\mathcal{B}$, while the tokenization-agnostic promoted pattern (b) accepts non-canonically tokenized sequences.

6.4 A Faster Way To Promote Patterns to BPE

One issue with the formulation in Equation (4) is that enumerating all of the merges and applying them is computationally expensive and many of the merge compositions are unnecessary if their components do not appear in the input pattern. We propose an alternative formulation that leverages pre-computation by saving a large "BPE" automaton

and then intersecting it with a subword-agnostic automaton (i.e., the automaton formed by Equation (3)).

Formally, let $\mathcal{B} = (\Gamma, \mu)$ be a BPE tokenizer, with $\Sigma \subseteq \Gamma$ as the base character alphabet. Consider the language Σ^*, represented by the automaton $\mathcal{A}_{\Sigma^*}$. Promoting this to a BPE automaton via Equation (4) gives:

$$\mathcal{A}_{\text{BPE}} = \textsc{Min}\left(\textsc{Proj}\left(\mathcal{A}_{\Sigma^*} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right)$$

an automaton over Γ that encodes all valid BPE subword sequences. That is, $\forall_{t\in\Gamma^*}\ t \in \mathcal{L}(\mathcal{A}_{\text{BPE}}) \iff \mathcal{B}(\textsc{Char}_\Gamma(t)) = t$.

Given an input character-level pattern $\mathcal{A}$ over Σ and $\mathcal{T}$, the tokenization-agnostic character-to-subword transducer for Γ (i.e., the transducer from Section 5, Equation (3)), then we have that

$$\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ \mathcal{T})) \circ \mathcal{A}_{\text{BPE}} \tag{5}$$

and

$$\textsc{Min}\left(\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right)$$

recognize the same language. This can be seen by noting that $\textsc{Proj}(\mathcal{A} \circ \mathcal{T})$ recognizes *all* subword sequences that correspond to $\mathcal{A}$. Then, composition with $\mathcal{A}_{\text{BPE}}$ filters it so that only subword sequences that are valid BPE sequences are accepted. Likewise, by construction $\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)$ converts each sequence in $\mathcal{A}$ into its BPE tokenization, yielding the same language.

Computing $\mathcal{A}_{\text{BPE}}$ is slow, but once it is computed we can reuse it freely to have much faster DFA intersection than computing the full form of Equation (4). We demonstrate this in Table 1, where we compare Equation (4) and Equation (5) for promoting an automaton with different sized BPE vocabularies. The inherent sequential nature of Equation (4) coupled with the intermediate increases in the number of states (which then requires minimization) results in a long runtime. On the other hand, each of the steps in the precomputed BPE version (Equation (5)) are relatively quick to compute and so the heavy cost of precomputing $\mathcal{A}_{\text{BPE}}$ is mitigated if we use it to promote many character-level patterns.

To test this empirically, we use tokenizers trained from scratch with HUGGINGFACE TOKENIZERS (Moi and Patry 2023) on the WIKITEXT-2 corpus (Merity et al. 2016) of various sizes. Each tokenizer uses the same training configuration, and only varies the target vocabulary size. The tokenizers are then converted into a list of merge gadgets for Equation (4) and compiled to a single $\mathcal{A}_{\text{BPE}}$ representation for Equation (5).

Indeed, in Table 1, we see that utilizing the precomputed $\mathcal{A}_{\text{BPE}}$ leads to multiple-orders-of-magnitude speedups over the sequential composition approach by avoiding unnecessary computation (i.e., the composition of irrelevant gadgets) and by taking advantage of efficient DFA intersection algorithms (i.e., lazy composition, where only the necessary states that are visited during composition are constructed, rather than the

Table 1
Wall time for BPE promotion of an automaton using different-sized vocabularies. The input automaton accepts strings that are within edit-distance-one of any of 100 words drawn from the WIKITEXT-2 training corpus and has 2.4k states and 46k transitions. As the BPE vocabulary grows, the time required for full composition (Equation (4)) increases dramatically while the precomposition method (Equation (5)) remains relatively fast. See `speedtest.py` in the reference implementation to reproduce this table.

Vocabulary Size	Full Composition (Equation (4))	Preconstructed $\mathcal{A}_{\text{BPE}}$ (Equation (5))	Speedup
4k	16.09s	0.19s	85x
8k	43.39s	0.24s	181x
16k	103.37s	0.38s	272x
32k	228.57s	0.45s	514x

full product automaton).[8] In the most extreme case with a 32*k* vocabulary, we observe a $>$500x speedup when using the precomputed automaton rather than the sequential application of each merge gadget.

6.4.1 Speed Comparison to Existing Tokenizers. It is difficult to directly compare our BPE implementation to existing tokenizers since the space of problems that they can solve is different. Specifically, existing off-the-shelf tokenizer implementations such as SENTENCEPIECE (Kudo and Richardson 2018) or HUGGINGFACE TOKENIZERS (Moi and Patry 2023) can only operate on individual string inputs, whereas our finite-state transduction framework is capable of tokenizing arbitrary regular languages, which can potentially have an infinite number of strings. Additionally, while production-ready tokenizers have been highly tuned, we use the standard OPENFST library and its Python wrapper PYNINI (Gorman 2016), which, despite being well optimized, are still generic libraries not specialized to tokenization.

Nevertheless, we prepared a benchmark paragraph for comparing the HUGGINGFACE BPE implementation to canonical subword promotion via Equation (5). We used the same set of tokenizers as in Tables 1 and 2. The benchmark paragraph has 1.2*k* characters and is taken from the WIKITEXT-2 corpus. The timings were measured by averaging the time required to produce the canonical tokenization of the text (either in a list of tokens or a linear automaton encoding only the canonical subword sequence) over 100 trials.[9]

At $|\Gamma| = 4k$, we observed 5.5x slowdown (0.0006*s* vs. 0.0033*s*) from the HUGGINGFACE tokenizer to our implementation. As the size of the vocabulary grows, the slowdown becomes more pronounced—at $|\Gamma| = 32k$, we observed a 70x slowdown (0.0002*s* vs. 0.014*s*). We attribute this to there being more arcs in the unconstrained subword lattice corresponding to partial paths that will be pruned out after composition with the canonical BPE automaton as well as the memory requirements of holding the canonical BPE tokenizer (we do not use the implicit form introduced in Section 6.5 since we did not implement it in a form compatible with PYNINI) and the runtime penalties that come with it.

8 This experiment was run on an `AMD Ryzen 9 5950X` 16-Core Processor with 32Gb of RAM.
9 The code to reproduce this experiment can also be found in `speedtest.py` in the reference implementation.

Table 2
Automaton size information for $\mathcal{A}_{\text{BPE}}$ for different vocabulary sizes when trained on the WIKITEXT-2 training corpus using the HUGGINGFACE tokenizer's BPE implementation (Moi and Patry 2023). As the number of merges increases, the naïve encoding's size explodes due to the large number of arcs present. The implicit encoding is able to avoid explicitly storing most arcs, resulting in a large space reduction. See `compression.py` in the reference implementation to reproduce this table.

Vocabulary Size	States	Representation	Explicitly Stored Arcs	Size (bytes)	Reduction
4k	827	Default	3.1M	61.8MB	96.10%
		Implicit	222k	2.4MB	
8k	2.0k	Default	14.4M	317.7MB	94.99%
		Implicit	1.3M	15.9MB	
16k	3.8k	Default	55.6M	1.3GB	94.73%
		Implicit	5.4M	70.3MB	
32k	6.4k	Default	173.0M	4.5GB	90.00%
		Implicit	32.2M	449.9MB	

Note that we can easily design natural cases where the transducer implementation outperforms any standard tokenizer. Take, for example, the edit distance automaton used in Table 1, which we found to encode nearly 90k unique sequences. Suppose the HUGGINGFACE tokenizer requires 0.0001 seconds to tokenize the shortest sequence in this automaton, then we would expect enumerating every sequence and tokenizing it serially would require at least 9 seconds, which is 20 times slower than using the 32k vocabulary precomputed BPE automaton approach from Table 1 on the same task.

6.5 Reducing the Encoding Size of $\mathcal{A}_{\text{BPE}}$

Naïvely storing $\mathcal{A}_{\text{BPE}}$ results in a large space cost. For example, $\mathcal{A}_{\text{BPE}}$ for a 16k vocabulary BPE tokenizer trained on WIKITEXT-2 data has 3.8k states and 56M transitions and takes up 4.5GB on disk when using the OPENFST finite-state transducer library (Riley, Allauzen, and Jansche 2009). To make matters worse, modern LLMs often use significantly larger vocabularies, which limits the use of this method.

Here, we discuss a practical optimization that can substantially reduce the encoding size of $\mathcal{A}_{\text{BPE}}$ while remaining performant.[10] The optimization is based on the following statement proven[11] by Berglund and van der Merwe (2023) (slightly restated):

Statement 1.1 (*Berglund and van der Merwe 2023; Corollary 2*) *Let $\mathcal{B} = (\Gamma, \mu)$ be a BPE tokenizer with atomic character alphabet $\Sigma \subseteq \Gamma$. Let w and $w' \in \Sigma^*$ be strings with*

10 The specific optimization we discuss is only for BPE tokenizers which use *trailing* or *intermediate* token-delimiter information (i.e., `[to] [po] [logy_]` or `[to] [_po] [_logy]`, respectively), or for tokenizers that do not use token-delimitation at all. A similar optimization exists for another common token-delimiter method, *leading* delimitation (i.e., `[_to] [po] [logy]`), but it is slightly more complicated.

11 A similar result was proven in Vieira et al. (2025; Appendix B), which provides a local property of subword sequences that ensures they are canonical BPE sequences. This result can also be used to produce a compressed representation $\mathcal{A}_{\text{BPE}}$ by computing arc and state information on-the-fly via the local property.

tokenizations $\mathcal{B}(w) = t_1 t_2 \dots t_m$ *and* $\mathcal{B}(w') = t'_1 t'_2 \dots t'_n \in \Gamma^*$*, respectively. Then,* $t_m = t'_1$ *implies* $\mathcal{B}(\textsc{Char}_\Gamma(t_1 t_2 \dots t_m)\textsc{Char}_\Gamma(t'_2 t'_3 \dots t'_n)) = t_1 t_2 \dots t_m t'_2 t'_3 \dots t_n$.

The implication of this corollary is that, if one has generated a token sequence $t_1 t_2 \dots t_n$, then only tokens $t \in \Gamma$ such that $\mathcal{B}(\textsc{Char}_\Gamma(t_n t)) = t_n t$ are valid continuations. In other words, the set of valid continuing subwords at each step depends only on the most-previously-generated token. This property induces a strict structure in the resulting automaton. Let $q_v = \delta(q_0, v)$ be the state reached by reading token $v \in \Gamma$ from the start state q_0 of $\mathcal{A}_{\text{BPE}}$.[12] Then, for any state $q \neq q_0$, either $\delta(q, v) = \delta(q_0, v) = q_v$ or $v \notin \delta(q)$ (i.e., the transition is disallowed).

This property also implies that *all* states other than q_0 are of the form q_v—so simply knowing the full set of q_v for $v \in \Gamma$ as well as the set of invalid continuations for each $v \in \Gamma$ is enough to reconstruct the entire $\mathcal{A}_{\text{BPE}}$ automaton.

We can now use a simple implicit encoding of $\mathcal{A}_{\text{BPE}}$: store each of $\delta(q_0, v)$ for all $v \in \Gamma$ and, for each $u \in \Gamma$, store the set $v \in \Gamma$ such that $\mathcal{B}(\textsc{Char}_\Gamma(uv)) \neq uv$, which we call $\textsc{Banned}(u)$.

Then, the transition function can be defined as:

$$\delta(q_u, v) = \begin{cases} \varnothing & v \in \textsc{Banned}(u) \\ \delta(q_0, v) & \text{otherwise,} \end{cases}$$

where each sub-function can be computed in $O(1)$ time via hashing.

We implement this idea and in Table 2 show the size reductions in a family of BPE tokenizers[13] trained on the same corpus but with different vocabulary sizes. For every vocabulary size, we observe that only a small percentage of arcs need to be stored explicitly, and the rest can be inferred implicitly. This leads to more than a 90% reduction in space for our full set of tokenizers.

7. An Application to Guided Generation

Guided generation is a technique used to constrain the outputs of language models to adhere to a specified pattern (Willard and Louf 2023; Koo, Liu, and He 2024). This is necessary since typically neural language models assign non-zero probability to all sequences due to the use of the softmax function. If a sequence of a specified pattern is desired, it would be beneficial to be able to assign non-zero probability to sequences which match that pattern, and zero probability to everything else. Guided generation frameworks typically take in a pattern (e.g., a regular expression) and constrain the model to match that pattern by masking out logits in the decoder softmax layer that correspond to tokens which would cause the sequence to fail to match the pattern. Formally, rather than sampling from:

$$\mathcal{P}(w \mid w_1 w_2 \dots w_k)$$

one would sample from:

$$\mathcal{P}(w \mid w_1 w_2 \dots w_k, w_1 w_2 \dots w_k w \Sigma^* \subseteq \mathcal{L}(\mathcal{A}))$$

12 It is possible that several subwords in Γ map to the same state after minimization. For clarity, we nevertheless refer to them with different subscripts (q_u, q_v, etc.).

13 The tokenizers in this experiment use trailing whitespace subword markers to delimit spaces.

where the second condition specifies that appending that subword would still result in a prefix of a string that matches the pattern specified by $\mathcal{A}$, ensuring that sequences which do not match the pattern are given zero probability.

Often, $\mathcal{A}$ is more easily specified at the character level, while the language model operates at some subword level. To fix this granularity mismatch, subword-level patterns are used (Willard and Louf 2023; Koo, Liu, and He 2024). However, prior work constrains models only to match the original pattern $\mathcal{A}$ without respect to the underlying tokenization scheme (i.e., the tokenization agnostic promotion described in Section 5). Here we discuss why that might cause issues with modeling.[14]

First, we note that language models are not conditioned on the *surface form* of the text, but rather the exact tokenization of the text. Thus, the same text tokenized in two different ways will map to different representations within the model.

Imagine we wish to constrain a language model trained with a MaxMatch-style tokenizer to output just the text `racecar` (encoded by $\mathcal{A}$). At the start of decoding, the model samples from the distribution:

$$\mathcal{P}(w \mid \texttt{<sos>}, w\Sigma^* \subseteq \mathcal{A})$$

Because there is no context and the model is not explicitly aware that its outputs are being constrained, the model may choose a higher probability token that still prefix-matches the regular expression—for example, `r`. Then, in the next inference step, the next token is drawn from:

$$\mathcal{P}(w \mid \texttt{<sos>␣r}, \texttt{r}w\Sigma^* \subseteq \mathcal{A})$$

After several decoding steps, imagine that the model has generated the token sequence `r␣a␣ce␣car`, which does indeed satisfy the prescribed regular expression. However, the canonical tokenization on which the model was trained is `race␣car` (i.e., this is the token sequence that the MaxMatch tokenizer would have produced given the character sequence `r␣a␣c␣e␣c␣a␣r`). The model was only able to generate a sequence with surface form that matched the input pattern due to the increasingly strict constraints on its output at each step. As typical neural language models are conditioned not on the surface form of the text, but rather on the explicit choice of tokenization, this can have large downstream effects. That is, the distribution:

$$\mathcal{P}(w \mid \texttt{<sos>␣r␣a␣ce␣car})$$

can be much different than the distribution:

$$\mathcal{P}(w \mid \texttt{<sos>␣race␣car})$$

despite the context having the same surface form.

One way to counteract this is to periodically stop decoding, detokenize and re-tokenize the sequence, and then resume decoding. This will put the text back into the canonical tokenized form that the model was trained on. However, this is a slow

14 A comprehensive investigation of the effect of tokenization-agnostic vs tokenization-aware constrained generation on LLMs is beyond the scope of this article, and we leave it for future work. Our goal here is simply to introduce the problem, which has not been discussed so far in the literature.

and expensive step (since the model must re-embed the entire context) and does not fully resolve the issue since the current text was still generated using non-canonical tokenizations. Thus, rather than using tokenization-agnostic pattern promotion, the practitioner can use the tokenization-aware pattern-promotion constructions described in Sections 6.2 and 6.3 to avoid the issues introduced in this section by not only forcing the model to adhere to some surface-text pattern, but also to the tokenization scheme that it was trained on.

Two recent studies have considered constraining to canonical BPE tokenizations during generation, and showed that adding this constraint provably reduces the KL-divergence of the distribution of subword sequences modeled by the constrained LLM and the true distribution of subword sequences (i.e., what the model was trained on) (Vieira et al. 2025; Chatzi et al. 2025), but neither study the effect of this constraining on downstream tasks. Further, it is not immediately clear if constraining to canonical sequences would benefit downstream tasks, as some works have found that non-canonical tokenizations carry meaningful signal in downstream tasks (Geh et al. 2024; Zheng et al. 2025).

However, canonicalization can confer a benefit to decoding speed during downstream tasks. Consider a constrained generation task like generating an output in a JSON schema, where the generation alternates between free-form text (i.e., the values in the JSON object) and predefined, structured text (i.e., the JSON syntax and the predefined field names). Since the predefined structured text has only one canonical tokenization, the canonical subword automaton corresponding to those parts has a special structure in that each state has only one outgoing arc. For example, consider the JSON schema

$$\{\texttt{``Name'':``}\Sigma^*\texttt{'',``Address'':``}\Sigma^*\texttt{''}\}$$

While the Σ^* sections can have free-form text and therefore an infinite number of possible (canonical) tokenizations, there is only one tokenization for `''Name'':` and `''Address'':`. When this is promoted to a canonical subword automaton, the sections of the automaton encoding those parts will be "linear"—there is only one outgoing arc per state, since there is only one valid path to decode the word. During decoding, since there is only one outgoing arc, the probability mass assigned to that arc must be 1. Therefore, no LLM decoding is necessary, and we can simply append the label on that arc to the context and continue to the next token for free. OUTLINES refers to this as *coalescence*[15] and several constrained generation systems have implemented this optimization and observed speedups (Zheng et al. 2024; Cognetta et al. 2025).

8. Conclusion

We formalized character-to-subword-level regular expression pattern promotion through a simple finite-state transduction framework. This framework was then extended to enforce dual, tokenizer-aware constraints: the promoted subword pattern simultaneously accepts only sequences which match the original character-level pattern, but do so while also only allowing sequences that would match an underlying BPE or MaxMatch tokenizer's output. While the MaxMatch-preserving construction is rooted in classical automata theory via the Aho-Corasick algorithm, the BPE construction is

15 `https://blog.dottxt.ai/coalescence.html`.

a novel and surprising result, given that at first glance, BPE is seemingly incapable of being captured by finite-state transducers due to its use of priority and that it does not process strings in any defined order. Further, we show that, contrary to the standard complexity analysis of the BPE construction, BPE-preserving pattern promotion can be done in polynomial time in the size of the input automaton and BPE tokenizer. Finally, we briefly discussed our finite-state transduction framework's application to guided generation. While subword-level patterns can constrain the outputs of LLMs to match a specific pattern, our tokenization-aware pattern promotions can do this while also adhering to the tokenization scheme that the model was trained on (and thus has an inductive bias towards), which has the potential to improve language modeling performance. We leave this for future work.

While the end results of our framework (i.e., subword-level automata that specify some character and possibly tokenizer-level constraints) are present in prior work, our framework is still novel in that it neatly captures a wide class of tokenization algorithms using only basic automata theory and finite-state transducer composition. This is in contrast to ad-hoc algorithms for promoting character-level automata into subword-level automata (Willard and Louf 2023; Koo, Liu, and He 2024) or converting BPE tokenizers into automata (Berglund, Martens, and van der Merwe 2024).

A number of practical limitations of this work exist. First, despite the general BPE and MaxMatch algorithms being captured by our framework, there are any number of minor variations of each that would require special care. Second, modern tokenizers have a complicated preprocessing pipeline, which may not be amenable to finite-state transducers. Third, our results here can work for context-free languages, since context-free languages are closed under intersection with regular languages, but we do not provide any insight on what the time or space complexity would be, especially for BPE promotion, and we consider this to be important future work. And fourth, even in the case of regular languages, the size of the automaton after canonical BPE-promotion can be quite large. As seen in Section 6.5, the BPE-promotion of Σ^* can be enormous, but we were able to develop a space-efficient encoding of it. However, this blow up can happen in other cases, such as if a regular expression even contains Σ^* as a term, where there aren't easy compression schemes.

Appendix A. Transducer Composition

Algorithm 5: Generic Transducer Composition
Input: $\mathcal{T}_1 = (\Sigma, \Xi, Q_1, q_{\text{start}}, F_1, \delta_1)$, $\mathcal{T}_2 = (\Xi, \Gamma, Q_2, q'_{\text{start}}, F_2, \delta_2)$
Output: Composition $\mathcal{T}_1 \circ \mathcal{T}_2$

$Q_c \leftarrow \{(q_{\text{start}}, q'_{\text{start}})\}$
$F_c \leftarrow \{\}$
$\delta_c \leftarrow$ empty transition function
queue $\leftarrow \{(q_{\text{start}}, q'_{\text{start}})\}$
while queue is not empty **do**
    $(q, q') \leftarrow$ queue.POP()
    if $(q, q') \in F_1 \times F_2$ **then**
        $F_c \leftarrow F_c \cup \{(q, q')\}$
    for $(q, i, \xi_1, p) \in \delta_1$ and $(q', \xi_2, o, p') \in \delta_2(y)$ **do**
        if $\xi_1 = \xi_2$ or $\xi_1 = \varepsilon$ or $\xi_2 = \varepsilon$ **then**
            $Q_c \leftarrow Q_c \cup \{(p, p')\}$
            queue.ENQUEUE$((p, p'))$
            $\delta_c \leftarrow \delta_c \cup \{((q, q'), i, o, (p, p'))\}$
return $\mathcal{T}_c = (\Sigma, \Gamma, Q_c, (q_{\text{start}}, q'_{\text{start}}), F_c, \delta_c)$

Appendix B. Tokenization-Agnostic Character-to-Subword Transducer Construction

Algorithm 6: Character-to-Subword Transducer Construction
Inputs: Subword Vocabulary $\Sigma \subseteq \Gamma \subset \Sigma^+$

$q_{\text{start}} \leftarrow$ new state, $Q \leftarrow \{q_{\text{start}}\}$, $F \leftarrow \{\}$
$\delta \leftarrow$ empty transition function
for $w \in \Gamma$ **do**
  cur $\leftarrow q_{\text{start}}$
  for $c \in w$ **do**
    $q' \leftarrow$ new state
    $Q \leftarrow Q \cup \{q'\}$
    if c is last character of w **then**
      $\delta \leftarrow \delta \cup \{(\text{cur}, c, w, q')\}$ ▷ *Arc emitting* w
    else
      $\delta \leftarrow \delta \cup \{(\text{cur}, c, \varepsilon, q')\}$
    cur $\leftarrow q'$
  $q' \leftarrow$ new state
  $Q \leftarrow Q \cup \{q'\}$
  $\delta \leftarrow \delta \cup \{(\text{cur}, \varepsilon, w, q')\}$
  $F \leftarrow F \cup \{q'\}$ ▷ q' *is end of word*
$\mathcal{T} \leftarrow \varepsilon$-CLOSE(DETERMINIZE($(\Sigma, \Gamma, Q, q_{\text{start}}, F, \delta)$))
return $\mathcal{T}$

Appendix C. MaxMatch Transducer Construction

Algorithm 7: Failure Function Computation (Song et al. 2021)
Input: Vocabulary Γ
Outputs: Trie root, Failure Functions f/F

root $\leftarrow$ NODE() ▷ *Build a trie representing* Γ
for $w \in \Gamma$ **do**
  cur $\leftarrow$ root
  for $c \in w$ **do**
    if $c \notin$ cur.children **then**
      cur.children[c] $\leftarrow$ NODE()
      cur.children[c].str $\leftarrow$ cur.str $+ c$
    cur $\leftarrow$ cur.children[c]
  cur.final = `True` ▷ *Corresponds to* $w \in \Gamma$
queue $\leftarrow$ [root] ▷ *Begin computing failure arcs*
while queue is not empty **do**
  cur $\leftarrow$ queue.DEQUEUE()
  for $c \in$ cur.children **do**
    $v \leftarrow$ cur.children[c]
    if $v.final$ **then**
      $f(v) \leftarrow r$
      $F(v) \leftarrow$ [cur.str]
    else
      $z \leftarrow f(\text{cur})$
      $Z \leftarrow []$
      while $z \neq$ `null` **do**
        EXTEND($Z, F(z)$)
        $z \leftarrow f(z)$
      if $z \neq$ `null` **then**
        $f(v) \leftarrow z$.children[c]
        $F(v) \leftarrow$ EXTEND(F(cur), Z)
    queue.ENQUEUE(v)
return root, f, F

We briefly reintroduce the terminology from Song et al. (2021). Let Γ be a subword vocabulary, and let T be a trie with state space V such that each state in the trie corresponds to a prefix of a token in Γ. Let $v \in V$ be a state in the trie corresponding to prefix $w_1 w_2 \dots w_k$. Define two functions $f(v) : V \to V \cup \{\texttt{null}\}$, the failure link function, and $F(v) : V \to \Gamma^*$, the failure pop function. $f(v)$ computes the state v which corresponds to the suffix $w_k w_{k+1} \dots w_n$ that remains after chunking $w_1 w_2 \dots w_{k-1}$ into tokens $t_1, t_2, \dots t_l$ where $t_i \in \Gamma$ and each corresponds to the longest possible matching token at each step. Then, $F(v) = [t_1, t_2, \dots t_k]$. Algorithm 7 does not differentiate between marked and unmarked subwords (e.g., `token` vs `##token`), but can easily be extended to handle this case.

Algorithm 8: Trie-to-Transducer Conversion
Inputs: Trie root, Failure Functions f/F, Vocabulary Γ over Σ
Output: Aho-Corasick Character-to-Subword Transducer $\mathcal{T}_{Aho}$

$Q \leftarrow \{q_{\text{root}}\}$ ▷ *State set*
queue $\leftarrow$ [root]
while queue is not empty **do**
  $cur \leftarrow$ queue.DEQUEUE()
  for $c \in$ cur.children **do**
    $v \leftarrow$ cur.children[c]
    APPEND(Q, q_v)
    $\delta \leftarrow \delta \cup (q_{\text{cur}}, c, \varepsilon, q_v)$
    queue.ENQUEUE(v)
  if $f(\text{cur}) \neq \texttt{null}$ **then** ▷ *Add φ-arcs*
    $q_{\text{tmp}} \leftarrow q_{\text{cur}}$
    for $w \in F(\text{cur})$ **do**
      if w is the last element in $F(\text{cur})$ **then**
        $\delta \leftarrow \delta \cup (q_{\text{tmp}}, \varphi, w, q_{f(\text{cur})})$
      else
        $q_{\text{next}} \leftarrow$ new state
        APPEND(Q, q_{next})
        $\delta \leftarrow \delta \cup (q_{\text{tmp}}, \varphi, w, q_{\text{next}})$
        $q_{\text{tmp}} \leftarrow q_{\text{next}}$
return $(Q, \Sigma, \Gamma, \delta, q_{\text{root}}, \{q_{\text{root}}\})$ ▷ $\mathcal{T}_{Aho}$

Appendix D. Proofs

Lemma 1
Let $\mathcal{A}$ be minimal, deterministic automaton over Σ and $\mathcal{T}$ be a character-to-subword transducer over $\Sigma \subseteq \Gamma \subset \Sigma^+$ (Equation (2)). Then ε-REMOVAL(PROJ($\mathcal{A} \circ \mathcal{T}$)) is deterministic.

Proof. Let $\mathcal{T}$ be a transducer which recognizes Equation (2) and was built by Algorithm 6. By construction, each path from q_0 to some $q \in F$ that does not contain an $(\varepsilon, \varepsilon)$ transition (that is, it does not return to the start state) contains only a single non-ε output label.

Thus, during composition, suppose there is some created state (q, q_{start}), where $q \in \mathcal{A}$ and q_{start} is the start state of the character-to-subword transducer. Let (p, q_{start}) be a state that is reached after reading some path with exactly one non-ε output label. Then, after projection, this path is a sequence of ε transitions followed by a single non-ε transition. This can be contracted to a single transition from (q, q_{start}) to (p, q_{start}) labeled with the non-ε label from Γ. Since each path from the start state to a final state in $\mathcal{T}$ contains a

single, unique output label, there cannot be more than one path leading out of (q, q_{start}) with that label, thus that state has a deterministic transition function. □

Proposition 1
Consider an automaton $\mathcal{A}$ and a single merge $G_{(a,b)}$, and a triplet of states $x, y, z \in Q$ such that $\delta(x, a) = y$, $\delta(y, b) = z$. During the composition of $\mathcal{A} \circ G_{(a,b)}$ over x, y, z at most one new distinguishable state is created.

Proof. First, we note that if there is no such triplet $x, y, z \in Q$ where $\delta(x, a) = y$, $\delta(y, b) = z$, then the composition $\mathcal{A} \circ G_{(a,b)}$ will describe exactly the same language as $\mathcal{A}$ by definition, so their minimal forms are identical and no new state is created. So, we assume that such a triplet exists. The composition of an automaton with a merge gadget must eliminate the possibility of reading a␣b.

Thus, the state y must be split to eliminate this sequence. Let y' and y'' be two new states. Then, for each $c \in \delta(x)$ s.t. $c \neq a$, set $\delta(x, c) = y'$. And, for all transitions $\delta(q, c) = y$ set $\delta(q, c) = y'$. Finally, set $\delta(q, a) = y''$. Then, set $\delta(y') = \delta(y)$ and $\delta(y'', c) = \delta(y, c), \forall c \neq b$.

By construction, all paths that passed through y are preserved, except for the path from $x \to y \to z$ reading a␣b and the original state y can be removed (alternatively, y' can be constructed from y in place). □

Remark 1
Figure 6 gives an example of Proposition 1.

Remark 2
We call y' and y'' **derived** states from the triplet x, y, z as they are the states that are derived from y after a merge gadget is applied and splits the state. We note that a derived state is specific to the exact triplet (i.e., the derived states of y from the triplet x, y, z are different from the derived states of y from the the triplet q, y, z, where $q \neq x$).

Lemma 2
Let $\mathcal{A}$ be a trim, deterministic automaton and $m_1, m_2, \ldots, m_k$ be a series of merges. Assume there is a triplet of states $x, y, z \in \mathcal{A}$ and let $Y = \{y, y^1, y^2, \ldots, y^j\}$ be the set of derived states of y some number of times during the compositions with $\{G_{m_i}\}_i^k$. Then, Y contains at most $k + 1$ distinguishable states.

Proof. This follows directly from the determinism of $\mathcal{A}$. Since $\mathcal{A}$ is deterministic, for any merge gadget, there is at most one suitable triplet from $\{x\} \times Y \times \{z\}$ which the merge gadget can be applied to (as otherwise either x or some $y' \in Y$ would contain two arcs with the same label, violating determinism). Thus, for each merge, the singular applicable triplet can be split at most once per merge according to Proposition 1. Since there are at most k merges, then a derived state can be split at most k times, so Y has a maximum cardinality of $k + 1$. □

So far we have only considered how successive merges act on a set triplet x, y, z or a set of triplets $\{x\} \times Y \times \{z\}$ where Y is the set of derived states from the triplet x, y, z. Now, we are concerned with the general case $X \times Y \times Z$. We consider this by case analysis.

Lemma 3
Let $\mathcal{A}$ be a trim, deterministic automaton (with states Q), $x, y, z \in Q$ be a triplet, and X, Y, Z be some set of derived states where $x \in X$, $y \in Y$, and $z \in Z$. Let $x', x'' \in X$,

$y', y'' \in Y$, and $z', z'' \in Z$ be arbitrary states in the derived state sets. Consider a merge gadget $G_{(a,b)}$ and let x', y', z' and $x'', y'', z'' \in X, Y, Z$ be two triplets of states. Then the application of $G_{(a,b)}$ to x', y', z' and x'', y'', z'' produces at most one new indistinguishable state.

Proof. We proceed by case analysis and show that for each choice of x^1, y^1, z^1 and x^2, y^2, z^2 then G_m is either not applicable over the triplet or produces exactly one new indistinguishable state. There are 8 possible cases, depending on if $x^1 \stackrel{?}{=} x^2$, etc. We assume that, if the case is possible, then the appropriate transitions for the merge (a, b) exist, as otherwise it is guaranteed that no new states are added.

Case 1): $x^1 = x^2, y^1 = y^2, z^1 = z^2$
This case is covered by Proposition 1.

Case 2): $x^1 = x^2, y^1 = y^2, z^1 \neq z^2$
This case is not possible, due to determinism (y^1 must have two arcs labeled b but which go to different states, which violates the determinism of $\mathcal{A}$).

Case 3): $x^1 = x^2, y^1 \neq y^2, z^1 = z^2$
This case is not possible, due to determinism (x^1).

Case 4): $x^1 = x^2, y^1 \neq y^2, z^1 \neq z^2$
This case is not possible, due to determinism (x^1).

Case 5): $x^1 \neq x^2, y^1 = y^2, z^1 = z^2$
Assume that the gadget operating on x^1, y^1, z^1 creates two splits y^1 into q' and q'' while x^2, y^1, z^1 splits y^1 into p', p''. Like Proposition 1, the transition functions are remapped like:

- $\delta(x^1, c) = p', \forall c \in \{c \mid c \in \delta(x^1) \wedge c \neq a \wedge \delta(x^1, c) = y^1\}$
- $\delta(x^1, a) = p''$
- $\delta(p') = \delta(y^1)$
- $\delta(p'', c) = \delta(y^1, c), \forall c \neq b \in \delta(y)^1$
- $\delta(x^2, c) = q', \forall c \in \{c \mid c \in \delta(x^2) \wedge c \neq a \wedge \delta(x^2, c) = y^1\}$
- $\delta(x^2, a) = q''$
- $\delta(q') = \delta(y^1)$
- $\delta(q'', c) = \delta(y^1, c), \forall c \neq b \in \delta(y)^1$

Then, by construction q' and p' are indistinguishable and can be merged, and likewise for q'' and p''. So since state y^1 is removed and replaced with q' and q'', only one new state is added.

Case 6): $x^1 \neq x^2, y^1 = y^2, z^1 \neq z^2$
This case is not possible, due to determinism (y^1).

Case 7): $x^1 \neq x^2, y^1 \neq y^2, z^1 = z^2$

This case has two subcases. First, consider some merge $G_{(a',b')}$ which operated on some set of states q, x, p to create x^1 and x^2. Then, $\delta(x^1, b') = \delta(x^2, b')$ for all $b' \neq d$. So if $b \neq d$, this case is impossible.

Next, assume $b = d$, then without loss of generality, there cannot be an arc $\delta(x^1, b)$, as it was removed to prevent reading $\mathtt{a}'\mathtt{b}'$ and so this triplet is not compatible with the merge and no new states are added. Thus, this case is not possible.

Case 8): $x^1 \neq x^2, y^1 \neq y^2, z^1 \neq z^2$
This case is not possible, due to a combination of the reasoning of Case 7 and determinism. In short, $y^1 = y^2$ or else this case is impossible, and if $y^1 = y^2$ then $z^1 \neq z^2$ implies $\mathcal{A}$ is not deterministic.

For all 8 cases, we have shown that either a merge is not able to operate on the specified pairs of triplets, or that, regardless of the choice of triplet, only one new indistinguishable state is added. □

Proposition 2

$$\textsc{Min}(\textsc{Proj}(\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n)) = \textsc{Min}(\textsc{Proj}(\ldots \textsc{Min}(\textsc{Proj}(\textsc{Min}(\textsc{Proj}(\mathcal{T}_1 \circ \mathcal{T}_2)) \circ \mathcal{T}_3)) \cdots \circ \mathcal{T}_n))$$

Proof. This follows directly from the definition of composition and projection. Minimization is not relevant as it does not change the language of the transducer. □

Proposition 3
Let $\mathcal{A}$ be a trim, deterministic automaton and $G_{(a,b)}$ be a BPE merge. Then ε-$\textsc{Removal}(\textsc{Proj}(\mathcal{A} \circ G_{(a,b)}))$ is deterministic.

Proof. Only the arc from state 0 to 1 has an ε-output. States 0 and 2 are output-deterministic, but state 1 has an a output transition on two arcs. However, by φ-semantics, only one of those arcs can be taken from any state by definition. Thus, at each state, there is only one valid transition per output symbol. Since $\mathcal{A}$ is also deterministic, each state has a deterministic transition function, up to ε-closure. There are no ε-cycles, and the only ε-transition is between state 0 and 1. Since state 0 does not have a as an output label (but state 1 does) the ε-closure does not violate determinism. □

Theorem 1
Given an automaton $\mathcal{A}$ and a series of merges $m_1, m_2, \ldots, m_k \in \mu$. Let $\mathcal{A}' = \textsc{Min}(\textsc{Proj}(\mathcal{A} \circ G_{m_1} \circ G_{m_2} \circ \cdots \circ G_{m_{k-1}}))$. Then, $\textsc{Min}(\textsc{Proj}(\mathcal{A}' \circ G_{\mu_k}))$ can be computed in $\textsc{Poly}(|\mathcal{A}|, k, |\Gamma|)$ time.

Proof. Consider any triplet $x, y, z \in Q$, of which there are $|Q|^3$. By Proposition 1, Lemma 2, and Lemma 3, each triplet can only form k new states as a result of composition with $\bigcirc_i^k G_{m_i}$. As a result, the size of $|\mathcal{A}'| \in O(k|\mathcal{A}|^3)$. Proposition 3 shows that any $\textsc{Min}(\textsc{Proj}(A \circ G_m))$ can be done in polynomial time. Thus, via induction, following Proposition 2, $\textsc{Min}(\textsc{Proj}(\mathcal{A}' \circ G_{\mu_k}))$ can be computed in $O(k|\mathcal{A}|^3)$ time by iteratively computing $\textsc{Min}(\textsc{Proj}(\ldots \textsc{Min}(\textsc{Proj}(\textsc{Min}(\textsc{Proj}(\mathcal{A} \circ G_{m_1})) \circ G_{m_2})) \cdots \circ G_{m_k}))$. The

maximum size of any intermediate automaton is $O(k|\mathcal{A}^3|)$ and so composition, projection, and minimization can be done in polynomial time. □

Corollary 1
The size of

$$\textsc{Min}\left(\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right)$$

is $\in \textsc{Poly}(|\mathcal{A}|, |\mu|, |\Gamma|)$.

Proof. Let $k = |\mu|$. Note $|\mu| \in O(|\Gamma|)$ by definition. Theorem 6.3.1 shows that

$$\textsc{Min}\left(\textsc{Proj}\left(\mathcal{A} \circ \left(\bigcirc_{(a,b)\in\mu} G_{(a,b)}\right)\right)\right)$$

can be computed in $\textsc{Poly}(|\mathcal{A}|, |\mu|, |\Gamma|)$ time. This implies the size of the output must also be polynomial in $|\mathcal{A}|$, $|\mu|$, and $|\Gamma|$. □

Acknowledgments
The authors would like to thank Vilém Zouhar and Tim Vieira for their helpful comments on earlier versions of this article. These research results were obtained from commissioned research (no. 22501) by National Institute of Information and Communications Technology (NICT), Japan.

References

Aho, Alfred V. and Margaret J. Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Communications of ACM*, 18(6):333–340. https://doi.org/10.1145/360825.360855

Allauzen, Cyril and Mehryar Mohri. 2009. N-way composition of weighted finite-state transducers. *International Journal of Foundations of Computer Science*, 20(4):613–627. https://doi.org/10.1142/S0129054109006772

Allauzen, Cyril and Michael D. Riley. 2018. Algorithms for weighted finite automata with failure transitions. In *Implementation and Application of Automata - 23rd International Conference, CIAA 2018, Proceedings*, volume 10977 of *Lecture Notes in Computer Science*, pages 46–58. https://doi.org/10.1007/978-3-319-94812-6_5

Berglund, Martin, Willeke Martens, and Brink van der Merwe. 2024. Constructing a BPE tokenization DFA. In *Implementation and Application of Automata - 28th International Conference, CIAA 2024, Proceedings*, volume 15015 of *Lecture Notes in Computer Science*, pages 66–78. https://doi.org/10.1007/978-3-031-71112-1_5

Berglund, Martin and Brink van der Merwe. 2023. Formalizing BPE tokenization. In *Proceedings of the 13th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2023*, volume 388 of *EPTCS*, pages 16–27. https://doi.org/10.4204/EPTCS.388.4

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. https://dl.acm.org/doi/abs/10.5555/3495724.3495883

Chatzi, Ivi, Nina Corvelo Benz, Stratis Tsirtsis, and Manuel Gomez-Rodriguez. 2025. Canonical autoregressive generation. *arXiv:2506.06446*.

Cognetta, Marco, Cyril Allauzen, and Michael Riley. 2019. On the compression of lexicon transducers. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing, FSMNLP 2019*, pages 18–26. https://doi.org/10.18653/v1/W19-3105

Cognetta, Marco, David Pohl, Junyoung Lee, and Naoaki Okazaki. 2025. Pitfalls,

subtleties, and techniques in automata-based subword-level constrained generation. In *Tokenization Workshop*. https://icml.cc/virtual/2025/47782

Cognetta, Marco, Vilém Zouhar, and Naoaki Okazaki. 2024. Distributional properties of subword regularization. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10753–10763. https://doi.org/10.18653/v1/2024.emnlp-main.600

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. https://doi.org/10.18653/v1/N19-1423

Dong, Yixin, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. XGrammar: Flexible and efficient structured generation engine for large language models. *arXiv:2411.15100*.

Gage, Philip. 1994. A new algorithm for data compression. *The C Users Journal Archive*, 12:23–38.

Gastaldi, Juan Luis, John Terilla, Luca Malagutti, Brian DuSell, Tim Vieira, and Ryan Cotterell. 2025. The foundations of tokenization: Statistical and computational concerns. *arXiv:2407.11606*.

Geh, Renato, Honghua Zhang, Kareem Ahmed, Benjie Wang, and Guy Van Den Broeck. 2024. Where is the signal in tokenization space? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3966–3979. https://doi.org/10.18653/v1/2024.emnlp-main.230

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, et al. 2025. Gemma 3 technical report.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, et al. 2024a. Gemma: Open models based on Gemini research and technology.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, et al. 2024b. Gemma 2: Improving open language models at a practical size.

Gorman, Kyle. 2016. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, pages 75–80. https://doi.org/10.18653/v1/W16-2409

Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 herd of models.

GUIDANCE-AI. GUIDANCE. https://github.com/guidance-ai/guidance.

Knuth, Donald E., James H. Morris Jr., and Vaughan R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350. https://doi.org/10.1137/0206024

Koo, Terry, Frederick Liu, and Luheng He. 2024. Automata-based constraints for language model decoding. In *First Conference on Language Modeling*. *arXiv:2407.08103*.

Kudo, Taku and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71. https://doi.org/10.18653/v1/D18-2012

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv:1907.11692*.

Merity, Stephen, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv:1609.07843*.

Mielke, Sabrina J., Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. 2021. Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP. *arXiv:2112.10508*.

Moi, Anthony and Nicolas Patry. 2023. Huggingface's tokenizers.

Outlines. OUTLINES. https://github.com/outlines-dev/outlines

Rajaraman, Nived, Jiantao Jiao, and Kannan Ramchandran. 2024. Toward a theory of tokenization in LLMs. *arXiv:2404.08335*.

Riley, Michael, Cyril Allauzen, and Martin Jansche. 2009. OpenFST: An open-source, weighted finite-state transducer library and its applications to speech and language. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, Tutorial Abstracts*, pages 9–10. https://doi.org/10.3115/1620950.1620955

Schuster, Mike and Kaisuke Nakajima. 2012. Japanese and Korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2012*, pages 5149–5152. https://doi.org/10.1109/ICASSP.2012.6289079

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725. https://doi.org/10.18653/v1/P16-1162

Sipser, Michael. 1997. *Introduction to the Theory of Computation*. PWS Publishing Company.

Song, Xinying, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. 2021. Fast WordPiece tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2089–2103. https://doi.org/10.18653/v1/2021.emnlp-main.160

Svete, Anej, Benjamin Dayan, Ryan Cotterell, Tim Vieira, and Jason Eisner. 2022. Algorithms for acyclic weighted finite-state automata with failure arcs. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8289–8305. https://doi.org/10.18653/v1/2022.emnlp-main.567

Touvron, Hugo, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. LlaMa: Open and efficient foundation language models.

Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. LlaMa 2: Open foundation and fine-tuned chat models.

Vieira, Tim, Tianyu Liu, Clemente Pasti, Yahya Emara, Brian DuSell, Benjamin LeBrun, Mario Giulianelli, Juan Luis Gastaldi, John Terilla, Timothy J. O'Donnell, and Ryan Cotterell. 2025. Language models over canonical byte-pair encodings. In *Forty-second International Conference on Machine Learning*.

Willard, Brandon T. and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv:2307.09702*.

Yang, An, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report.

Zheng, Brian Siyuan, Alisa Liu, Orevaoghene Ahia, Jonathan Hayase, Yejin Choi, and Noah A. Smith. 2025. Broken tokens? Your language model can secretly handle non-canonical tokenizations. *arXiv:2506.19004*.

Zheng, Lianmin, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. *arXiv:2312.07104*.

Zouhar, Vilém, Clara Meister, Juan Gastaldi, Li Du, Mrinmaya Sachan, and Ryan Cotterell. 2023a. Tokenization and the noiseless channel. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5184–5207. https://doi.org/10.18653/v1/2023.acl-long.284

Zouhar, Vilém, Clara Meister, Juan Gastaldi, Li Du, Tim Vieira, Mrinmaya Sachan, and Ryan Cotterell. 2023b. A formal perspective on byte-pair encoding. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 598–614. https://doi.org/10.18653/v1/2023.findings-acl.38