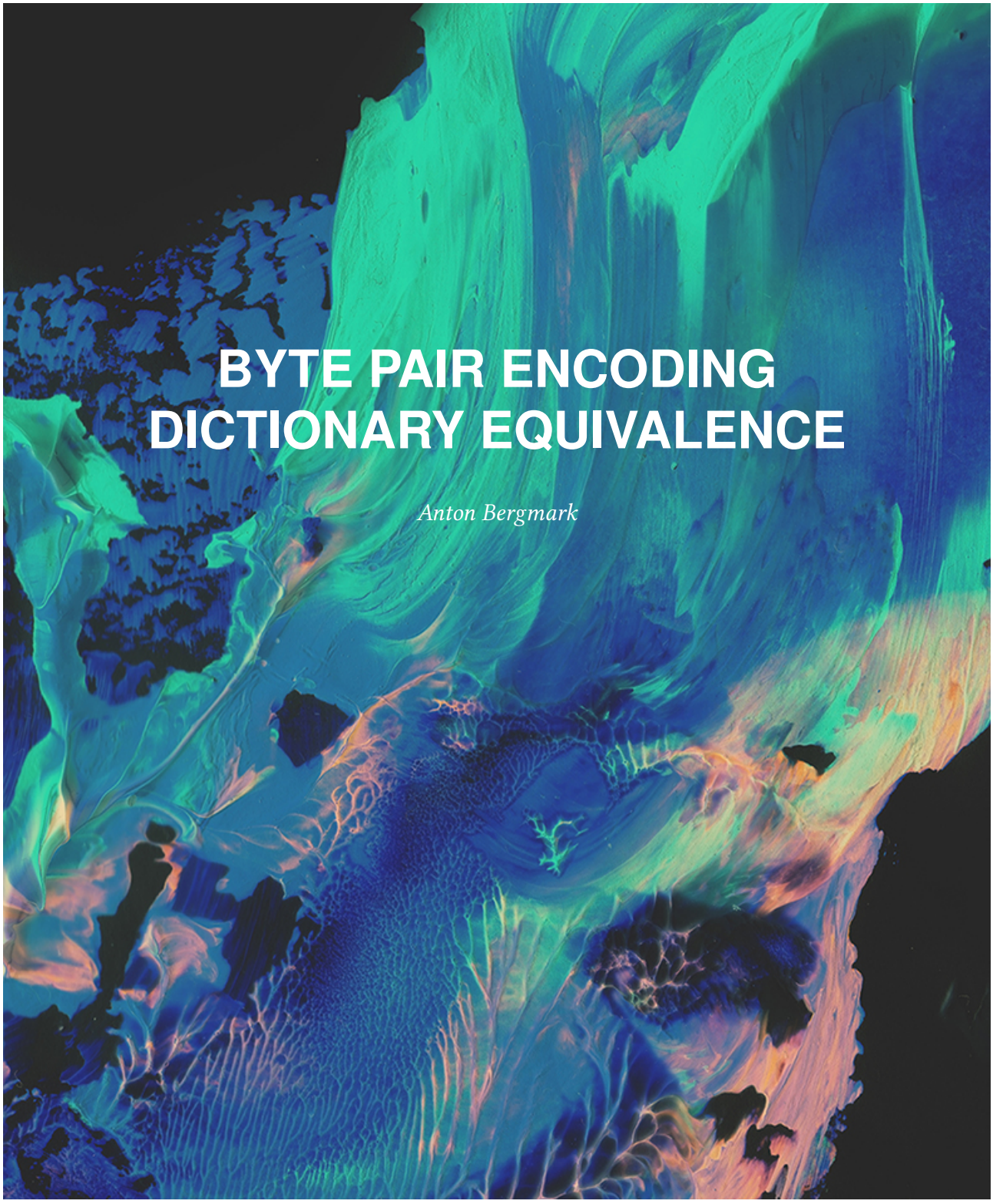

BYTE PAIR ENCODING DICTIONARY EQUIVALENCE

Anton Bergmark

Bachelor Thesis, 15 hp/credits
Education program

2025

Abstract

This thesis proposes a rewrite procedure for determining the equivalence of a subset of byte-pair encoding (BPE) dictionaries, a data structure essential to the tokenization process in modern large language models. We present a normalization procedure that constructs a *normal form* for proper BPE dictionaries, where a dictionary is *proper* if every token in a merge rule of length greater than one has been created by a higher-priority rule. We prove that two proper dictionaries containing the same rules are equivalent if and only if their normal forms are identical. The existing approach determines dictionary equivalence by transforming dictionaries into deterministic finite automata (DFAs) and checking for DFA equivalence. This approach is computationally expensive for large vocabularies, where the performance depends on the size of the token alphabet. In contrast, the new normalization procedure is more efficient in practice, since the time complexity of the normalization procedure depends on the length of the tokens instead, which are typically considerably smaller than the size of the token alphabet. This results in a more scalable solution for equivalence checking in real-world scenarios.

Acknowledgements

I would like to thank my supervisor, Willeke Martens, for their guidance and support throughout this thesis project. Since the beginning of the project, you have provided invaluable insights and feedback that have made this thesis possible and have taught me so much about this interesting field of research. Your dedication, expertise, and encouragement have made this journey both fun and rewarding. I could not have asked for a better supervisor and I sincerely hope that we can work together again in the future. I also want to thank my partner, Wilma, for your constant support and encouragement throughout this project. When my motivation waned or I felt overwhelmed, you were always there to encourage me and remind me of what is important. I am so grateful for everything you have done for me. Thank you.

Contents

1 Introduction

Large language models (LLMs) must be able to handle any character sequence, including those never seen during training [5]. Subword tokenization addresses this challenge by splitting rare or unfamiliar sequences into smaller, more frequent components called subwords or *tokens*, drawn from a fixed vocabulary. For instance, the word 'supercalifragilisticexpialidocious' does not appear in the vocabulary of GPT-4, yet it can still be processed as the sequence of known tokens super cal if rag ilistic exp ial id ocious as indicated by the boxes.

Byte-pair encoding (BPE) tokenization is the dominant subword tokenization algorithm used by modern LLMs. BPE tokenization transforms a sequence of characters into a sequence of tokens using a *dictionary* of *merge rules*. The dictionary specifies how and in what order to combine smaller units into larger tokens [6], see Chapter 2. While BPE tokenization has several sematic variants, such as those used by HuggingFace[1] and SentencePiece [4], these approaches yield identical tokenizations if the dictionary is *proper* – that is, when every rule's constituents are derived from higher-priority rules [2, 1]. In Figure 1 we see the process of tokenizing a string using the merge rules from a dictionary.

$$D = \begin{bmatrix} u \wr v \\ v \wr y \\ y \wr w \end{bmatrix}$$

u v y w
⇓ $u \wr v$
uv y w
⇓ $v \wr y$
uv y w
⇓ $y \wr w$
uv yw

Figure 1: The process of tokenizing a string using a dictionary. The dictionary with the merge rules is seen on the left. Feeding the promt 'uvyw' divides the string into single character tokens. If a rule can be applied, we merge the two tokens in the tokenization and do this until all rules in priority order has applied, producing a final tokenization. Here $u \wr v$ and $y \wr w$ can apply to the given input, but not $v \wr y$.

Given two BPE dictionaries, it is not obvious just by looking at the rules whether they produce the same tokenization for all possible input strings, even if they contain the same merge rules in a different order. Swapping the order of as little as two rules can lead to completely different tokenizations of a given string. As dictionaries in practice can contain tens of thousands of rules, determining whether two dictionaries are equivalent and produce the same tokenizations for all input strings is a non-trivial task.

Berglund et al. [1] introduced an algorithm to construct deterministic finite automata (DFA) from a proper dictionary that recognizes all valid tokenizations according to the dictionary. We take a closer look at this and the time complexity of this procedure in Chapter 2. This introduces the possibility of applying algorithms for analyzing DFAs. One such algorithm is the DFA equivalence algorithm presented by Hopcroft and Karp [3], which can determine whether two DFAs are equivalent and, in turn, whether two dictionaries produce the same

[1] https://huggingface.co/learn/llm-course/chapter6/5

tokenizations. This indirect approach of constructing and comparing DFAs is an inefficient and expensive procedure, especially for large dictionaries. In Chapter 3 we prove that the time complexity of this procedure is $O(|\Gamma||D|^2)$, where $|\Gamma|$ is the size of the token alphabet and $|D|$ is the number of merge rules in the dictionary.

Instead of constructing BPE token DFAs and testing dictionary equivalence via DFA equivalence, we explore whether equivalence can be determined directly from the structure of the dictionaries themselves. In particular, we propose a normal form for proper dictionaries such that two dictionaries are equivalent if and only if their normal forms are identical in Chapter 4. We construct a normalization algorithm for proper BPE dictionaries and show that the time complexity of this procedure is $O(t|D|^2)$, where t is the length of the longest token in the token alphabet and typically small in practice. Finally, in Chapter 5 we discuss the implications of our findings and the reason why the normalization procedure is more efficient than the indirect DFA approach in practice, even when the time complexity of the procedures may look similar at first glance.

2 BPE Tokenization

Byte-Pair Encoding tokenization is a widely used method in natural language processing and large language models to break down words into subwords or tokens. BPE tokenization allows LLMs to represent any character sequence in terms of a sequence of tokens from a fixed vocabulary, making them more flexible and robust. In this chapter, we introduce the semantics of BPE tokenization and dictionaries, how they work, and how dictionaries can create DFAs that can be used to recognize valid tokenizations.

2.1 Notation

We mainly follow Berglund et al. [1] for notation. An *alphabet* Σ is a finite set of symbols. The set Σ^* denotes all finite strings over Σ, including the empty string ϵ, and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.

Individual symbols are denoted by $\alpha, \beta, \gamma, a, b, c$, strings or tokens by u, v, w, and tokenizations by τ, ϕ. For a tokenization τ, $|\tau|$ denotes the number of tokens in τ. When writing $\phi = u \wr \tau \wr v$, we mean that ϕ consists of a token u, followed by an arbitrary long tokenization τ, and then a token v. Thus, $|\phi| = |\tau| + 2$.

A token alphabet Γ over Σ is a set of tokens $w \in \Sigma^+$, such that either $w \in \Sigma$ or there exists some $u, v \in \Gamma$ such that $w = uv$. When the underlying alphabet Σ is clear or unimportant, we simply refer to the token alphabet as Γ.

A token sequence is denoted as $u_1 \wr u_2 \wr \ldots \wr u_n$ where $u_i \in \Gamma$ for $i = 1, 2, \ldots, n$. We say that a token sequence τ is a tokenization of a string w if the concatenation of its tokens yields w. The concatenated string obtained from τ is denoted by $\pi(\tau)$.

For a deterministic finite automaton A, we let $\mathcal{L}(A)$ denote the set of all strings accepted by A.

2.2 BPE tokenization semantics

BPE tokenization is used by different LLMs such as the OpenAI GPT models, see GPT-SW3 model [7], as well as by the Meta AI Llama models[1]. Slightly different implementations of BPE tokenization exist, with the two dominant semantics defined by the Hugging Face[2] and SentencePiece [4] tokenizers, respectively. In this section, we introduce the semantics of BPE tokenization and how they tokenize text. We also clarify the difference between proper and improper dictionaries.

2.2.1 Dictionaries

A dictionary is a list of merge rules, used to tokenize input strings. The definition of a dictionary can be seen in Definition 1.

[1] https://github.com/meta-llama/llama/tree/main
[2] https://huggingface.co/learn/nlp-course/chapter6/5

Definition 1 (based on Definition 1 in [2]). *A* byte-pair dictionary $D = [u_1 \wr v_1, \ldots, u_n \wr v_n]$ *over* Σ *is a sequence of token pairs, where every token is in* $\Sigma^* \setminus \{\epsilon\}$. *Each* $u_i \wr v_i$ *is called a* rule, *and any rule* $u_i \wr v_i$ *has* higher priority *than the rule* $u_j \wr v_j$ *if* $i < j$.

Definition 2 formalizes the notion of a *proper* dictionary, a key property when it comes to describing the semantics of byte-pair dictionaries.

Definition 2 (based och Definition 5 in [2]). *A dictionary is* proper *if for each rule* $r_i = u_i \wr v_i$ *in* D *and* $|u_i| > 1$, *there exists some rule* r_j, *where* $i < j$, *that produces the token* u_i. *Similarly, for each rule* $r_i = u_i \wr v_i$ *in* D *and* $|v_i| > 1$, *there exists some rule* r_j, *where* $i < j$, *that produces the token* v_i.

While improper dictionaries exist, most practical ones are proper by construction. At each step, the current most frequent token pair is added as the next rule in the dictionary and merged into a single token in the data. Frequencies are updated after each merge, and the process repeats until the target vocabulary size is reached.

2.2.2 Hugging Face and SentencePiece BPE

As earlier mentioned, different implementations of BPE tokenization exist, notably Hugging Face and SentencePiece. While both are similar, they differ in the frequency with which they select new rules during tokenization (see Algorithm 1[3] [1] and Algorithm 2 [1], respectively). In the Hugging Face variant, once the highest-priority applicable rule is selected for a token sequence, it is applied repeatedly as many times as possible before choosing a new rule. In contrast, SentencePiece re-evaluates and selects the highest-priority rule after each individual merge.

While it may seem like a minor difference, the frequency of rule selection can lead to different tokenizations when the dictionary is not proper [2]. In such cases, applying a lower-priority rule may enable a higher-priority rule to become applicable. In the Hugging Face implementation, repeated application of the lower-priority rule before re-evaluating priorities can block merges that would have occurred had the higher-priority rule been selected immediately, as in SentencePiece. In contrast, when the dictionary is proper, no higher-priority rule can become applicable after selecting a lower-priority one; as such, both implementations behave identically, see Example 1.

Example 1. The dictionary $D = [ab \wr ab, c \wr ab, a \wr b]$ is not proper since the token ab used in the merge rules $ab \wr ab$ and $c \wr ab$, is generated by a lower-priority rule. Let $w = c \wr a \wr b \wr a \wr b$. Hugging Face BPE and SentencePiece BPE produce the tokenizations $c \wr abab$ and $cab \wr ab$ respectively, when applied to w. This is because Hugging Face BPE merges all token pairs $a \wr b$ before going back to higher-priority rules. SentencePiece BPE merges only the first occurence of token pair $a \wr b$, before selecting the now available highest priority rule $c \wr ab$, and repeat the process. It merges $c \wr ab$ to cab, and then $a \wr b$ to ab. ◇

We let the tokenization of the string w using a proper dictionary D be denoted $\mathbb{T}^D(w)$. Thus, the tokenization $\mathbb{T}^D(w)$ is the final result of the tokenization process when merging tokens using the rules in the dictionary in sequence.

[3] https://huggingface.co/learn/nlp-course/chapter6/5

Algorithm 1 Hugging Face BPE tokenization (see Definition 4 in [2])

Input: an input string $\alpha_1 \cdots \alpha_n$, a dictionary $D = [u_1 \wr v_1, \ldots, u_m \wr v_m]$
$\tau \leftarrow \alpha_1 \wr \ldots \wr \alpha_n$
$\tau' \leftarrow \tau$
$i \leftarrow 1$
while $i < m$ **do**
    $temp \leftarrow []$
    $j \leftarrow 1$
    while $j \leq |\tau| - 1$ **do**
        if $\tau_j \wr \tau_{j+1} = u_i \wr v_i$ **then**
            append $u_i v_i$ to $temp$
            $j \leftarrow j + 2$
        else
            append τ_j to $temp$
            $j \leftarrow j + 1$
        end if
    end while
    if $j = |\tau|$ **then**
        append $\tau_{|\tau|}$ to $temp$
    end if
    $i \leftarrow i + 1$
    $\tau \leftarrow tokenizelist(temp)$
    if $i \leq m$ AND $\tau \neq \tau'$ **then**
        $i \leftarrow 1$
        $\tau' \leftarrow \tau$
    end if
end while
Output: the tokenization τ

Algorithm 2 SentencePiece BPE tokenization (see Definition 3 in [2])

Input: an input string $\alpha_1 \cdots \alpha_n$, a dictionary $D = [u_1 \wr v_1, \ldots, u_m \wr v_m]$
$\tau_0 \leftarrow \alpha_1 \wr \ldots \wr \alpha_n$
$i \leftarrow 1$
while $i < m$ **do**
    $temp \leftarrow []$
    $j \leftarrow 1$
    while $j \leq |\tau| - 1$ **do**
        if $\tau_j \wr \tau_{j+1} = u_i \wr v_i$ **then**
            append $u_i v_i$ to $temp$
            $j \leftarrow j + 3$
            $\tau_{:j} \leftarrow tokenizelist(temp)$
            $i \leftarrow 0$
            $break$
        else
            append τ_j to $temp$
            $j \leftarrow j + 1$
        end if
    end while
    $i \leftarrow i + 1$
end while
Output: the tokenization τ

2.3 BPE Token DFA

A BPE token DFA is a DFA that recognizes valid tokenizations of input strings for a given proper dictionary. We define a BPE token DFA $A = (Q, \Gamma, q_0, \delta, F)$ as a 5-tuple where

- Q is a finite set of states,
- Γ is the alphabet consisting of tokens present in the dictionary,
- q_0 is the initial state,
- δ is the transition functions between states, and
- F is the set of accepting states.

Creating a BPE token DFA for a dictionary has several advantages, such as being able to analyze the DFA using algorithms specifically designed for DFAs. In this section, we briefly present the construction algorithm and its time complexity, as first described by Berglund et al.

2.3.1 Algorithm for constructing a BPE token DFA

Berglund et al. [1] describe the construction of a BPE token DFA A for an arbitrary regular language L and a proper dictionary D over Σ of length $|D| = n$, such that $\mathcal{L}(A) = \mathbb{T}^D(L)$. The construction starts from a DFA accepting L, for example the universal DFA A_0 for the language Σ^*. An example of the universal DFA for the language over $\Gamma = \{a, b\}$ is shown in Figure 2. We build upon such a DFA using the rules $u_i \wr v_i$ in the dictionary D. We merge one rule at a time using a merging algorithm, in order of priority. At each step, the algortihm is applied to DFA A_{i-1} with rule $u_i \wr v_i$ as input, where $i \in [1, n]$. By analyzing the algorithm, we see that at most one new state gets added to the DFA, which is the state q_i.

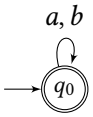

Figure 2: Universal DFA for the language Σ^*, where $\Sigma = \{a, b\}$.

We define the rules in the dictionary D as $D = [a \wr b, a \wr a]$ and run the algorithm. We start with the first merge rule $a \wr b$, since it is the highest priority rule, and we use the universal DFA in Figure 2 as the starting point.

The algorithm does as follows. First, for each pair of transitions in the DFA A of the form $\delta(s_1, u) = s_2$ and $\delta(s_2, v) = s_3$, we record the triplet (s_1, s_2, s_3). These correspond to the paths where reading u from s_1 leads to s_2, and then reading v from s_2 leads to s_3. For every such state, we create a new transition from s_1 to s_3 by defining $\delta(s_1, uv) = s_3$. In our case, since we start with the universal DFA, we have the transitions $\delta(q_0, a) = q_0$ and $\delta(q_0, b) = q_0$ for the rule $a \wr b$ i.e. the triplets (q_0, q_0, q_0). For every s_2 in the list of transitions, which in this case is q_0, we add a new state to the DFA, and if the state is an accepting state, we inherit that property, which we do in this case for the one state that is added. The resulting DFA can be seen in Figure 3.

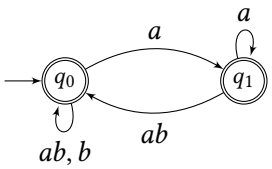

Figure 3: DFA for the token $a \wr b$

For every s_2 in the list of transitions, we now check if the u and v are the same. If they are not, as in our case, we add new transitions from the new state to the states that s_2 transitions to, except for the state that s_2 transitions when it reads the token v. Otherwise, the same happens, but we ignore states containing the rule uv as well. Finally, for every state with a transition using the token u, we replace the transition with a transition to the new state. The result show the DFA in Figure 3.

To get a clearer understanding of the process, we show the process for when the rule $a \wr a$ after the rule $a \wr b$ is run, we get the DFA in Figure 4.

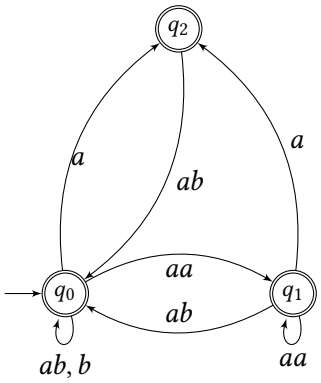

Figure 4: DFA for the token $a \wr a$ after the token $a \wr b$

The process is the same for the token $a \wr a$ as it was for the token $a \wr b$, where we also observe the same state transition patterns. In this case, we get two different patterns that get added to the list, (q_0, q_1, q_1) and (q_1, q_1, q_1). The token aa is added from q_0 to q_1 and q_1 to q_1. A new final state is then added to the DFA and since u and v are the same, we add all transitions from q_1 to the new state except for the transitions with the token a and aa. Finally we replace the transitions with the token u thoughout the DFA to go to the new state. The result is shown in Figure 4 and an example usage of the BPE token DFA is shown in Example 2.

Example 2. Take the string $w = abaaaab$ for example. The tokenization produced is $\mathbb{T}^D(w) = ab \wr aa \wr a \wr ab$. In the DFA we start at the initial state q_0 and read that the first token is ab, which takes us back to the state q_0. The next token is aa, which takes us to state q_1. Next, since the rule $a \wr a$ has a lower-priority than the rule $a \wr b$, we transition to q_2 when reading the token a. Finally, we read the token ab, which takes us back to the state q_0. Since we are in an accepting state, the tokenization is valid. ◇

2.3.2 Time complexity of dictionary to BPE token DFA algorithm

Berglund et al. [1], demonstrate that the procedure to construct the BPE token DFA $A' = (Q', \Gamma, q_0', \delta', F')$ from the DFA $A = (Q, \Sigma, q_0, \delta, F)$ using a proper dictionary D can be done in time $O(|Q||\Gamma||D|^2)$.

To give an idea why this is the case, we need the notion of a target count of a token in a DFA as seen in Definition 3.

Definition 3 (based on Definition 4 in [1])**.** *For a DFA $A = (Q, \Gamma, q_0, \delta, F)$ and symbol $\alpha \in \Gamma$ the* target count *of α in A is denoted targetc(A, α), is defined as targetc$(A, \alpha) = |\{q' \mid \exists q \in Q, \delta(q, \alpha) = q'\}|$ i.e. the number of distinct states q' in A that is the destination of a transition with some token.*

In the work of Berglund et al., the time complexity was formalized as seen in Theorem 1.

Theorem 1 (based on Theorem 3 in [1])**.** *The merge algorithm can be applied to a BPE token DFA $A = (Q, \Sigma, q_0, \delta, F)$ in time $O(|Q||\Sigma|)$. Applying the algorithm iteratively with a proper dictionary D can be done in time $O(|Q||\Gamma||D|^2)$, where Γ is the token alphabet induced by D.*

Observe that applying the merge algorithm to an input token DFA $A = (Q, \Gamma, q_0, \delta, F)$ with a merge rule $u \wr v$ produces a new token DFA $A' = (Q', \Gamma', q_0, \delta', F')$ such that $max_{w \in \Gamma}\{\text{targetc}(A, w)\} = max_{w \in \Gamma'}\{\text{targetc}(A', w)\}$ *and* for all $w \in \Gamma$, targetc$(A, w) =$ targetc(A', w).

The number of states added every time the merge algorithm runs is bounded by targetc(A, u) and targetc(A, v). All states s_2 in the triplets (s_1, s_2, s_3) where s_1 has a transition on the token u to state s_2, and the state s_2 has a transition on the token v to state s_3, are saved to a set S_2. We loop over each unique state s_2 in S_2 and create a new state $new(s_2)$. The number of unique $s_2 \in S_2$ is less than or equal to targetc(A, u) since there can exist many s_1 for every s_2. Similarly, targetc(A, v) limits the number of new symbols that can be added, since the states that get an incoming transition with the token uv must have an incoming transition with the token v already. This means that the number of new states created is limited by $k = \max\{\text{targetc}(A, \alpha) \mid \alpha \in \Gamma\}$ for every merge rule in the dictionary D that we input to the algorithm. Thus, the number of states in the resulting DFA is at most $|Q| + k|D|$, where Q is the states in the original DFA and k is the highest target count of some token in the DFA.

A single merge of a BPE token DFA can be done in time complexity $O(|Q||\Gamma|)$. The final DFA A' has the alphabet size Γ', which is of size $|\Gamma| + |D|$, i.e. the size of the single token alphabet Γ plus the number of merge rules. Thus, the time complexity of one merge is at worst $O((|Q| + k|D|)|\Gamma'|)$. We must apply the merge $|D|$ times, which gives us the final time complexity of $O(|Q||\Gamma'||D| + k|D||\Gamma'|)$ to produce the final DFA A'. Since k is limited by $|Q|$, we write the time complexity as $O(|Q||\Gamma'||D| + |Q||\Gamma'||D|^2) = O(|Q||\Gamma'||D|^2)$.

Consider again the exempel in Subsection 2.3.1, which starts from the universal DFA in Figure 2. In this case, the initial state set has the size $|Q| = 1$. Therefore, $|Q|$ can be ignored in asymptotic time complexity. From now on, to simplify the analysis of the time complexity, without introducing unnecessary complications, we refer to the time complexity of the BPE token DFA construction algorithm as $O(|\Gamma'||D|^2)$.

3 Indirect dictionary equivalence

As explained earlier, it is a challenge to determine if two dictionaries produce the same tokenizations for any input given to them, since as little as one rule swap can produce an entirely different tokenization. We define dictionary equivalence as in Definition 4.

Definition 4. *Two proper dictionaries D and D' are said to be* equivalent*, denoted $D \cong D'$, if $\mathbb{T}^D(\Sigma^*) = \mathbb{T}^{D'}(\Sigma^*)$, meaning for any input string, the resulting tokenization is the same for both dictionaries.*

We give an example of two dictionaries that are equivalent and two that are not in Example 3.

Example 3. Let $D = [a \wr b, c \wr d]$ and $D' = [c \wr d, a \wr b]$. The order of the rules in this example does not matter, therefore, $D \cong D'$. However, if we take the dictionary $D'' = [a \wr b, c \wr a]$ and $D''' = [c \wr a, a \wr b]$, $D'' \ncong D'''$, since, for example, on string cab, D'' yields the tokenization $c \wr ab$ and D''' yields the tokenization $ca \wr b$. ◇

This chapter focuses on the time complexity of the indirect procedure for determining the dictionary equivalence.

3.1 Indirect procedure

Building upon the work of Berglund et al. [1], we are able to associate a dictionary D with a token DFA A such that $L(A) = T^D(\Sigma^*)$, as explained in Section 2.3. Then, two dictionaries D and D' must clearly be equivalent if their associated token DFAs are equivalent. The two DFA are constructed using the algorithm described in Subsection 2.3.1 and the resulting DFA A and A' accepts the same languages $\mathcal{L}(A) = \mathbb{T}^D(\Sigma^*)$ and $\mathcal{L}(A') = \mathbb{T}^{D'}(\Sigma^*)$. Having two different DFA, we can run algorithms meant for analyzing DFA and in our case, we want to check for equivalence between two DFA specifically.

3.1.1 Time complexity for DFA equivalence

There are different approaches to determine if a pair of DFAs is equivalent. One such approach involves minimizing the DFAs and checking if the two resulting minimized DFAs are identical. Let $A = (Q, \Sigma, q_0, \delta, F)$ and $A' = (Q', \Gamma, q_0', \delta', F')$ be two DFAs. The process of determining equivalence has a time complexity of $O(n \log n)$, where n is the number of states in larger of the two DFA, i.e. $n = \max\{|Q|, |Q'|\}$[3]. Hopcroft and Karp [3] introduced a new method for testing whether two finite automata are equivalent in linear time, avoiding the minimization step of the previous approach. The theorem for the time complexity of this algorithm can be seen in Theorem 2.

Theorem 2 (based on Theorem 1 in [3])**.** *For two DFA, $A = (Q, \Sigma, q_0, \delta, F)$ and $A' = (Q', \Gamma, q_0', \delta', F')$, the execution time of the algorithm determining whether they are equivalent is bounded by a constant times the product of the number of input symbols, $|\Gamma|$, with the sum of the number of states of each of the automata, $|Q| + |Q'|$. Thus, the time complexity of the algorithm is $O(|\Gamma|(|Q| + |Q'|))$.*

The proof of this theorem is explained in detail by Hopcroft and Karp. Parts of the algorithm executes in a constant $|\Gamma|$ times $|Q| + |Q'|$, respectively. Other parts run at constant times $|\Gamma|$ times the number of pairs popped from the stack that is used. The number of pairs popped is proven to be bounded by $|Q|+|Q'|$ with the reasoning that each time a pair is placed on the stack, the number of sets used in the algorithm is reduced by one since two sets are merged. Since there are only $|Q| + |Q'|$ sets, the authors conclude that at most $|Q| + |Q'| - 1$ sets are pushed onto the stack. The algorithm is then bounded by $|\Gamma|$ times $|Q| + |Q'|$ plus $|Q| + |Q'|$ times $|\Gamma|$, which has a time complexity of $O(|\Gamma|(|Q| + |Q'|))$, where $|\Gamma|$ is the number of symbols in the alphabet and $|Q|$ and $|Q'|$ is the number of states in the two DFAs.

Note that BPE token DFAs are deterministic finite automata that operate over tokens rather than individual characters. Each input string corresponds to exactly one token sequence, meaning token DFAs accept a unique tokenization per string. As a result, tokens can be treated as indivisible characters, allowing standard DFA algorithms such as the equivalence check by Hopcroft and Karp to be applied directly.

3.2 Time complexity of the indirect procedure

As we saw in Subsection 2.3.2 and Subsection 3.1.1, the complexity of the algorithms needed for the indirect procedure is $O(|\Gamma||D|^2)$ and $O(|\Gamma|(|Q|+|Q'|))$ respectively. Here, $|Q|$ and $|Q'|$ is the total number of states in respective completed DFA, $|\Gamma|$ is the number of symbols in the alphabet, and $|D|$ is the number of merge rules in the dictionary. The indirect procedure is a combination of the two algorithms, where we first create the two BPE token DFAs from two dictionaries and then run the algorithm for DFA equivalence. The time complexity of the indirect procedure is the sum of the two algorithms, resulting in the time complexity seen in Theorem 3.

Theorem 3. *Let D and D' be two proper dictionaries. The indirect procedure to check the equivalence of the two dictionaries is carried out in time $O(|\Gamma| \max\{|D|, |D'|\}^2)$, where $|\Gamma|$ is the token alphabet induced by D and D', and $\max\{|D|, |D'|\}$ is the number of merge rules in the larger dictionary.*

Proof. The procedure begins by applying the algorithm presented by Berglund et al. [1] to the universal DFA for respective dictionaries D and D'. Creating the two BPE token DFAs A and A' from the universal DFA can be done in time $O(|\Gamma||D|^2)$ and $O(|\Gamma||D'|^2)$ respectively.

The time complexity to determine equivalence depends on the number of states in the token DFAs. As universal DFA only has one state, the initial target count is 1 for all tokens in the alphabet Σ. This means that the BPE token DFAs, A and A', have at most $|Q| = 1 + |D|$ and $|Q'| = 1+|D'|$ states, respectively. Hence, the time complexity for determining the equivalence of two BPE token DFA is $O((2 + |D| + |D'|) \cdot |\Gamma|)$.

Putting this together, we can see that the time complexity of the indirect procedure is $O(|\Gamma| \max\{|D|, |D'|\}^2)$, as the time complexity for BPE token DFA construction dominates. □

The time complexity of the BPE token DFA construction by Berglund et al. dominates over testing for BPE token DFA equivalence, so even if some faster algorithm showing equivalence between two DFAs would be introduced, the time complexity of the indirect procedure would remain the same and the increased performance would only be marginal.

4 Normal form for proper dictionaries

The normal form of a data structure refers to a unique representation within a set of equivalent objects that is typically derived through a process called normalization. Our goal is to define a normal form for proper dictionaries. It is meant to be the default order of merge rules, so that the normal form of two dictionaries is identical if and only if the original dictionaries are equivalent. Naturally, the dictionaries we compare need to contain the same applicable rules if they are equivalent; otherwise, the dictionaries could not produce the same tokenizations for any input $w \in \Sigma^*$.

To construct a normal form for proper dictionaries, we first need to show that swapping the order of certain merge rules in a dictionary does not affect the final tokenization of a string and, therefore, can be swapped freely. Next, we provide a normalization procedure and an algorithm to construct a normal form for proper dictionaries, and also provide a complexity analysis of the said algorithm.

4.1 Swappable rules

An important definition when proving that we can swap the order of rules is a refinement, see Definition 5.

Definition 5 (based on a description at p.24 in [2])**.** *We say that tokenization φ is a refinement of φ' if we can obtain φ' from φ by merging some of the subtokenizations in φ, i.e. $\phi \wr \phi' \wr \phi''$ is a refinement of $\phi \wr \pi(\phi') \wr \phi''$.*

Throughout, for any dictionary $D = [r_1, \ldots, r_k]$ and $i \leq k$, let $D_i = [r_1, \ldots, r_i]$ denote the prefix dictionary consisting of the first i rules of D. This means that $\mathbb{T}^{D_i}(w)$ is a refinement of $\mathbb{T}^{D_j}(w)$ if $i < j$, see Example 4.

Example 4. Let $D = [a \wr b, c \wr d, e \wr f, ab \wr c]$, then $D_3 = [a \wr b, c \wr d, e \wr f]$ and $D_2 = [a \wr b, c \wr d]$. We say that $\mathbb{T}^{D_2}(w)$ is a refinement of $\mathbb{T}^{D_3}(w)$ and $\mathbb{T}^{D}(w)$, since D_2 is a prefix of D_3 and D. Applying tokenization $\mathbb{T}^{D_3}(\mathbb{T}^{D_2}(w))$ to some string w, gives us the same result as applying tokenization $\mathbb{T}^{D_3}(w)$ and applying the rule r_3 to tokenization $\mathbb{T}^{D_2}(w)$. ◇

Another important notion is the notion of a useful rule. A rule is useful if there exists some string such that the rule is applied when tokenizing the said string. Berglund and Van Der Merwe show that this coincides with whether the rule is used when tokenizing the string to which the rule applies. We define a useful rule as in Definition 6.

Definition 6 (based on Corollary 1 in [2])**.** *Let D be a dictionary and $r_i = u \wr v$ be a rule in the dictionary. The rule r_i is said to be* useful *if and only if $\mathbb{T}^{D_{i-1}}(uv) = u \wr v$, meaning that the rule r_i is applied when tokenizing the string uv.*

We can see an example of a useful rule in Example 5 and a not useful rule in Example 6.

Example 5. let $D = [a \wr b, c \wr d, ab \wr cd]$. To see that the third rule $ab \wr cd$ is useful, consider the string $abcd$. We first apply rule $a \wr b$ to $a \wr b \wr c \wr d$, yielding token sequence $ab \wr c \wr d$. Next, we

apply the second rule, yielding $ab \wr cd$. Now clearly the last rule $ab \wr cd$ can be applied which means that the rule is useful. ◇

Example 6. Let $D = [b \wr c, a \wr b, c \wr d, ab \wr cd]$ and we check if the fourth rule $ab \wr cd$ is useful. Again, consider the same string $abcd$. When applying the first rule, we get $\mathbb{T}^{D_1}(w) = a \wr bc \wr d$. The next two rules cannot be applied, meaning $\mathbb{T}^{D_3}(w) = a \wr bc \wr d$. Now we can see that $ab \wr cd$ cannot be applied, thus the rule is not useful. ◇

In Lemma 1, we show that we are any prefix dictionary D_i of D with $i < |D| - 1$, yields the tokenization $u \wr v \wr y$ when tokenizing the string uvy if the last two rules are $u \wr v$ and $v \wr y$. This is important because the order of the last two rules are able to affect the tokenization of the string uvy in some cases.

Lemma 1. Let the two rules $u \wr v$ and $v \wr y$ be the last two rules in some dictionary D of length k, i.e. rules with index $k-1$ and k in some arbitrary order. Further, suppose both rules are useful. Then, $\mathbb{T}^{D_{k-2}}(uvy) = u \wr v \wr y$.

Proof. We can prove this by induction. We assume that for all $i \leq k-2$ that $\mathbb{T}^{D_i}(uvy) = \mathbb{T}^{D_i}(u) \wr \mathbb{T}^{D_i}(v) \wr \mathbb{T}^{D_i}(y)$, where r_{k-1} and r_k are the two useful rules $u \wr v$ and $v \wr y$ in some arbitrary order, i.e. $\mathbb{T}^{D_{k-2}}(uv) = u \wr v$ and $\mathbb{T}^{D_{k-2}}(vy) = v \wr y$.

Base case: $i = 0$**.** In this case, we have that $\mathbb{T}^{D_0}(uvy) = \mathbb{T}^{D_0}(u) \wr \mathbb{T}^{D_0}(v) \wr \mathbb{T}^{D_0}(y)$, which is trivially true since for any arbitrary string $w = \alpha_1 \cdots \alpha_n \in \Sigma^*$, $\mathbb{T}^{D_0}(w) = \alpha_1 \wr \cdots \wr \alpha_n$.

Inductive hypothesis: $i < k-2$**.** We show that if

$$\mathbb{T}^{D_i}(uvy) = \mathbb{T}^{D_i}(u) \wr \mathbb{T}^{D_i}(v) \wr \mathbb{T}^{D_i}(y),$$

then also

$$\mathbb{T}^{D_{i+1}}(uvy) = \mathbb{T}^{D_{i+1}}(u) \wr \mathbb{T}^{D_{i+1}}(v) \wr \mathbb{T}^{D_{i+1}}(y).$$

First, recall that to determine if a rule $x \wr z$ can be applied to a tokenization $w_1 \wr \cdots \wr w_n$, we scan each adjacent token pair $w_i \wr w_{i+1}$ from $i = 1$ to $i = n-1$. If a match is found, we merge the pair into $w_i w_{i+1}$, yielding the new token sequence $w_1 \wr \cdots \wr w_{i-1} \wr w_i w_{i+1} \wr w_{i+2} \wr \cdots \wr w_n$. We then continue checking, starting from $i+2$.

Let $\mathbb{T}^{D_i}(u) = u_1 \wr \cdots \wr u_{n_1}$, $\mathbb{T}^{D_i}(v) = v_1 \wr \cdots \wr v_{n_2}$, and $\mathbb{T}^{D_i}(y) = y_1 \wr \cdots \wr y_{n_3}$. If the inductive hypothesis does not hold, it must be the case that rule r_{i+1} matches either $u_{n_1} \wr v_1$ or $v_{n_2} \wr y_1$.

Suppose that applying r_{i+1} to $\mathbb{T}^{D_i}(uvy)$ results in the merging of u_{n_1} and v_1. Then r_{i+1} also merges the pair in $\mathbb{T}^{D_i}(uv)$, because $\mathbb{T}^{D_i}(uv) = u_1 \wr \cdots \wr u_{n_1} \wr v_1 \wr \cdots \wr v_{n_2}$. However, $\mathbb{T}^{D_{i+1}}(uv)$ is a refinement of $\mathbb{T}^{D_j}(uv)$ for $j \geq i$ *and* $\mathbb{T}^{D_{k-2}}(uv) = u \wr v$ since $u \wr v$ is useful. Therefore the merging of tokens u_{n_1} and v_1 implies a contradiction to the initial assumption that $u \wr v$ is useful. The same reasoning applies to the case where r_{i+1} merges the tokens v_{n_2} and y_1, and combining these observations we can conclude that $\mathbb{T}^{D_{i+1}}(uvy) = \mathbb{T}^{D_{i+1}}(u) \wr \mathbb{T}^{D_{i+1}}(v) \wr \mathbb{T}^{D_{i+1}}(y)$. □

We are able to swap the order of certain rules in a dictionary without affecting the tokenization of strings, but this is not possible for every single case. To show this we need to define a new notation. Let $\varphi = u_1 \wr u_2 \wr \ldots \wr u_l$. We define the subset $pos^i_{z \wr w}(\varphi)$ of indexes in φ as

$$pos^i_{z \wr w}(\varphi) = \{j + i - 1 | \forall j \in \{1, \ldots, l-1\} \text{ s.t. } u_j = z, u_{j+1} = w\}$$

where $i \in \{1, 2\}$. Thus, $pos^1_{z \wr w}(\varphi)$ is the set of positions of z in the tokenization φ where token z is followed by token w, and $pos^2_{z \wr w}(\varphi)$ is the set of positions of w in the tokenization φ where token z is followed by token w.

In Lemma 2, we show when we are able to swap the order of the two last rules in a dictionary.

Lemma 2. Let $D = [r_1, \ldots, r_{k-1}, r_k]$ be a proper dictionary over token alphabet Γ with $k \geq 2$. Assume that $r_{k-1} = u \wr v$ and $r_k = x \wr y$ for some $u,\ v,\ x,\ y \in \Gamma$, and that neither x nor y is equal to the token uv.

Let D' be the dictionary obtained from D by swapping the order of r_{k-1} and r_k, i.e. $D' = [r_1, \ldots, r_{k-2}, r_k, r_{k-1}]$. Then, $D' \cong D$ if and only if $u \neq y$ and $v \neq x$.

Proof. **If $u \neq y$ and $v \neq x$, then $D' \cong D$.** The first $k-2$ rules of dictionary D and D' are identical, hence $\mathbb{T}^{D_{k-2}}(w) = \mathbb{T}^{D'_{k-2}}(w)$ for all $w \in \Sigma^*$. Denote $\mathbb{T}^{D_{k-2}}(w)$ as φ.

Suppose for some $w \in \Sigma^*$, either r_{k-1} or r_k, but not both, can be applied to φ. Then, $\mathbb{T}^{D}(w) = \mathbb{T}^{D'}(w)$: say that r_{k-1} cannot be applied, i.e., φ contains no matching token pair. By the assumption that neither x nor y is equal to the token uv and D is proper, no matching token pairs are introduced when applying r_k first instead.

This leaves us to consider the case where both rules r_{k-1} and r_k are applicable. Observe that $pos^1_{w \wr z}(\varphi)$ and $pos^2_{w \wr z}(\varphi)$ are subsets of the set of positions of tokens w and z in φ. Since both $u \neq y$ and $v \neq x$ by assumption, the intersection of $pos^1_{u \wr v}(\varphi) \cap pos^2_{x \wr y}(\varphi)$ and $pos^2_{u \wr v}(\varphi) \cap pos^1_{x \wr y}(\varphi)$ must be empty. Clearly, if also $pos^1_{u \wr v}(\varphi) \cap pos^1_{x \wr y}(\varphi)$ and $pos^2_{u \wr v}(\varphi) \cap pos^2_{x \wr y}(\varphi)$ are empty, then the rules r_{k-1} and r_k cannot be applied to overlapping token pairs in φ and are thus independent of each other. That is, the order of the merge rules does not matter.

Suppose that $pos^1_{u \wr v}(\varphi) \cap pos^1_{x \wr y}(\varphi)$ is not empty, then clearly $u = x$, since $pos^1_{u \wr v}(\varphi)$ and $pos^1_{x \wr y}(\varphi)$ are subsets of the positions of tokens u and x in φ. However, since $pos^1_{u \wr v}(\varphi)$ consists exactly of those positions followed by a v, the non-emptiness of the intersection implies that $v = y$ also. But then the two rules r_{k-1} and r_k apply at the same position in φ, i.e. $u = x$ and $v = y$, and they are identical, which makes the lower-priority rule not useful. We can similarly reason that $pos^2_{u \wr v}(\varphi) \cap pos^2_{x \wr y}(\varphi)$ must be empty.

If $D' \cong D$, then $u \neq y$ and $v \neq x$. Suppose that the converse holds, i.e., that there exists some D such that $u = y$ or $v = x$, yet $D' \cong D$. Start by assuming $x = v$, then by Lemma 1, since both r_{k-1} and r_k are useful, we have that $\mathbb{T}^{D_{k-2}}(uvy) = u \wr v \wr y$. But then in D we get the tokenization $\mathbb{T}^{D}(uvy) = uv \wr y \neq u \wr vy = \mathbb{T}^{D'}(uvy)$, which is a contradiction. The same reasoning applies to the case where $y = u$ using the tokenization $\mathbb{T}^{D_{k-2}}(xuv) = x \wr u \wr v$. Thus, there is always a tokenization produced by the dictionary D that is not equal to the tokenization produced by D', meaning that $D \ncong D'$, contradicting the assumption that $D' \cong D$. □

Theorem 4 contains the restrictions that we can abide by to change the order of rules while still keeping equivalence and the proper attribute for any proper dictionary.

Theorem 4. *Let $D = [r_1, \ldots, r_k]$ be a proper dictionary with $k \geq 2$ such that all merge rules r_i for $i \in [k]$ are useful. Let $r_l = u \wr v$ and $r_{l+1} = x \wr y$ with $l \in [k-1]$ be any arbitrary pair of subsequent merge rules in D such that neither x nor y are equal to uv. Let D' be the proper dictionary obtained from D by swapping the order of the merge rules r_l and r_{l+1} in D. Then, $D' \cong D$ if and only if $u \neq y$ and $v \neq x$.*

Proof. **If $u \neq y$ and $v \neq x$, then $D' \cong D$.** In Lemma 2 we showed that if r_k and r_{k-1} are the last two rules in the dictionary, then Theorem 4 holds. Therefore, assume that r_l and r_{l+1} are not the last two rules. We know that $\mathbb{T}^{D_{l+1}}(w) = \mathbb{T}^{D'_{l+1}}(w)$ for all $w \in \Sigma^*$. Further, no other changes have been made to the order of the remaining rules when comparing D with D', thus we have

$$\mathbb{T}^D(w) = \mathbb{T}^D_k(\mathbb{T}^D_{k-1}(...(\mathbb{T}^{D_{l+1}}(w))...)) = \mathbb{T}^D_k(\mathbb{T}^D_{k-1}(...(\mathbb{T}^{D'_{l+1}}(w))...)) = \mathbb{T}^{D'}(w),$$

meaning that $D \cong D'$.

If $D' \cong D$, then $u \neq y$ and $v \neq x$. Assume that the converse holds, i.e., $u = y$ or $v = x$, yet $D \cong D'$. In particular, suppose $u = y$. If the rules r_k and r_{k+1} are the last two rules in the dictionary, we know from Lemma 2 that the theorem must hold. Therefore, assume that they are not the last rules. In the proof of Lemma 2 we see that $\mathbb{T}^{D_l}(uvy) = uv \wr y$ and $\mathbb{T}^{D'_l}(uvy) = u \wr vy$. We prove that none of the subsequent rules in D and D' can ensure that $\mathbb{T}^D(uvy) = \mathbb{T}^{D'}(uvy)$, hence restoring equivalence, by contradiction.

Assume that $\mathbb{T}^D(uvy) = \mathbb{T}^{D'}(uvy)$. Then necessarily, $\mathbb{T}^D(uvy) = \mathbb{T}^{D'}(uvy) = uvy$ and D must contain the rule $uv \wr y$, while D' must contain the rule $u \wr vy$. However, since the only difference between D and D' is the order of the rules r_l and r_{l+1}, both dictionaries must contain the rules $uv \wr y$ and $u \wr vy$. Both of these rules cannot be useful at the same time, since they cannot both be used during the tokenization of uvy. This contradicts the assumption that all rules in D are useful. Thus, we can conclude that there exist no subsequent rules that can restore the equality between the two dictionaries. □

4.2 Normal form

The normal form of a dictionary is based on sorting the merge rules in lexicographical order, while respecting the swapping restrictions imposed by Theorem 4. This is because normalization procedure must account for cases where certain swaps are not allowed to ensure that the resulting dictionary is equivalent to the original dictionary.

We once again assume that the dictionary D only consists of useful rules. The idea is to sort the rules in the dictionary in lexicographical order, but since dictionaries can contain different unuseful rules and still be equivalent, we remove these rules first to construct the normal form.

Remark 1. We begin by removing any rules that are not useful in the dictionary. This is done by creating a new dictionary D', a set Γ, and iterating the rules in the dictionary D. For each rule r_i, if $\pi(r_i) \notin \Gamma$, we append the rule r_i to the dictionary D' and add the token $\pi(r_i)$ to the set Γ. The result is a dictionary D' that contains only the useful rules in the dictionary D. This is not a limitation since we can always remove any rules that are not useful from the dictionary without affecting the tokenizations produced by said dictionary. That is, rules that are not useful would never be applied to any string $w \in \Sigma^*$.

In the context of strings, a lexicographic order is defined as follows. We say that $w = a_1 \cdots a_n$ is lexicographically smaller than $w' = b_1 \cdots b_m$, and write $w < w'$, if $a_i < b_i$ where $i \leq k = \min\{n, m\}$ is the first index so that $a_i \neq b_i$, or, if no such index exists and $n \leq m$.

We define the lexicographical order between two token rules in Definition 7.

Definition 7. *The merge rule $u \wr v$ is lexicographically smaller than merge rule $x \wr y$, we write $u \wr v < x \wr y$, if $u < x$, or, if $u = x$ then $v < y$.*

Note that the lexicographical order of merge rules is total, meaning that it is transitive,

reflexive, antisymmetric, and strongly connected. Implying this, we can always compare two merge rules against each other and determine which one is lexicographically smaller.

We define a normal form of a dictionary as in Definition 8.

Definition 8. *Let ν be a transformation that reorders the rules in a dictionary D containing only useful rules. The resulting order is then $\nu(D)$, such that another dictionary D', which is a permutation of D, is equivalent to D if and only if $\nu(D) = \nu(D')$. The resulting dictionary $\nu(D)$ is defined as* normal form *of D.*

If we only used the lexicographic order as in Definition 7 as well as the restrictions posed by Theorem 4 to reorder the rules in the dictionary, some cases would cause problems, as we can see in Example 7.

Example 7. Let the dictionary $D = [r_1, r_2, r_3]$ and the rules $r_1 = b \wr c$, $r_2 = a \wr b$ and $r_3 = a \wr c$. We try to sort the rules in the lexicographical order defined above. The rule r_2 is lexicographically smaller than r_1, but the rules cannot be move past each other according to Theorem 4. We still want to move the rule r_3 to the position of r_1 because it is lexicographically smaller than r_1, but it is not lexicographically smaller than r_2 which is blocked by r_2. This poses a problem. Let $D' = [r_3, r_1, r_2]$. The dictionary would then be equivalent with D, but applying the lexicographical order would not produce the same resulting order, since r_3 would be placed before r_1 in D'.

This would mean that we need to consider more than just the lexicographical order and the restrictions of moving rules when constructing the normal form of a token dictionary, otherwise equivalent dictionaries would not construct the same order of rules.

To determine the new position of the rule r_i in D, we check all rules prior to r_i until we find the first rule r_j so that r_i cannot move past r_j due to Theorem 4. Next, we move r_i to the highest priority position $k < j$ so that $r_i < r_k$. This disregards the lexicographical order of certain rules which are prevented from moving up in the dictionary. However, by iteratively applying this process to each rule r_i for $i = 2$ to k, we ensure that every rule is placed in the highest possible priority position according to the lexicographical order and the swapping restrictions.

We conjecture that this results in an order of rules such that two dictionaries have the same normal form if and only if they produce equivalent tokenizations for all input strings $w \in \Sigma^*$. The algorithm for constructing a normal form of a proper dictionary is shown in Algorithm 3.

Algorithm 3 Normalization procedure for proper dictionaries

Input: A proper dictionary $D = [u_1 \wr v_1, \ldots, u_k \wr v_k]$
$i \leftarrow 2$
while $i \leq k$ **do**
    $j \leftarrow 1$
    $p\_index \leftarrow NULL$
    while $i - j \geq 1$ **do**
        if $u_i \notin \{v_{i-j}, u_{i-j}v_{i-j}\}$ and $v_i \notin \{u_{i-j}, u_{i-j}v_{i-j}\}$ **then**
            if $u_i \wr v_i < u_{i-j} \wr v_{i-j}$ **then**
                $p\_index \leftarrow i - j$
            end if
        else
            if $p\_index \neq NULL$ **then**
                move $u_i \wr v_i$ between the rules $u_{p\_index} \wr v_{p\_index}$ and $u_{p\_index-1} \wr v_{p\_index-1}$
                break
            end if
        end if
        $j \leftarrow j + 1$
        if $i - j < 1$ **and** $p\_index \neq NULL$ **then**
            move $u_i \wr v_i$ between the rules $u_{p\_index} \wr v_{p\_index}$ and $u_{p\_index-1} \wr v_{p\_index-1}$
        end if
    end while
    $i \leftarrow i + 1$
end while
Output: A normal form $D' = [r'_1, \ldots, r'_k]$ of D.

4.3 Algorithm correctness

Since the algorithm can never move a merge rule past some other rule that would break the restrictions in Theorem 4, we know that the order produced would not break the equivalence or the proper attribute between the reordered dictionary and the original dictionary order. The tricky part of proving the equivalence of this algorithm is demonstrating that the normalization procedure actually results in a unique representation. We want to give the intuition that Conjecture 1 is correct and that the algorithm produces a unique dictionary for all equivalent dictionaries.

Conjecture 1. *Let $D = [r_1, \ldots, r_k]$ and $D' = [r'_1, \ldots, r'_k]$ be two proper dictionaries over token alphabet Γ with $k \geq 2$. The algorithm in Algorithm 3 produces normal form dictionaries $\nu(D)$ and $\nu(D')$ for D and D', respectively, such that $\nu(D)$ and $\nu(D')$ have the same order if and only if $D \cong D'$.*

We know that Algorithm 3 only swaps rules if the restrictions stated in TheoremTheorem 4 are not violated, i.e. if $u \neq y$ and $v \neq x$ for some rules $r_l = u \wr v$ and $r_{l+1} = x \wr y$ and the dictionary keeps its proper attribute. This means that for every swap of rules preserves equivalence, thus $D \cong \nu(D)$. Thus, if $D \cong D'$, then $\nu(D) \cong \nu(D')$. Thus, we can conclude that there can never be two non-equivalent dictionaries that produce the same order of rules.

Let $D \cong D'$. If D and D' would produce different orders of rules when performing the normalization procedure, then there must be a first position in $\nu(D)$ and $\nu(D')$ that does not contain the same merge rule. We call this rule r in $\nu(D)$ and r' in $\nu(D')$. This means that r

in $v(D)$ comes after r' in $v(D')$ if $r < r'$. Then r must be blocked by another rule r'' in D'. However, this rule r'' comes after r in $v(D)$. The idea is the same if $r > r'$. Our hypothesis is that no equivalent dictionaries exist such that rules that cannot be swapped have different relative priorites in those dictionaries. The Lemma 1 shows a part of the proof needed to prove this. In Example 8 we show that the algorithm produces the same order of rules for two equivalent dictionaries of different orders.

Example 8. Let $D = [k \wr f, d \wr b, b \wr f, d \wr c, f \wr a]$ and $D' = [d \wr c, k \wr f, b \wr f, f \wr a, d \wr b]$ which both are equivalent dictionaries. Following Algorithm 3, we get $v(D) = [d \wr b, b \wr f, d \wr c, k \wr f, f \wr a]$ and $v(D') = [d \wr b, b \wr f, d \wr c, k \wr f, f \wr a]$. The algorithm produces the same order of rules for both dictionaries. ◇

We take the resulting dictionary in Example 8 and apply the algorithm again in Example 9. This shows that $v(D) = v(v(D))$ and that the algorithm is idempotent.

Example 9. Let the normal form dictionary $v(D) = [d \wr b, b \wr f, d \wr c, k \wr f, f \wr a]$. For each rule after the first rule in $v(D)$, consider whether it can be moved past any higher-priority rule according to Theorem 4, and whether it is lexicographically smaller than the rule.

The rule $b \wr f$ cannot be moved past $d \wr b$. We then check rule $d \wr c$ and if it can be moved past $b \wr f$ and $d \wr b$, but is not lexicographically smaller than either, so it does not move. Next, $k \wr f$ can be moved past all preceding rules, but is not lexicographically smaller than any of them, so it does not move. Finally, $f \wr a$ cannot be moved past $k \wr f$. The resulting dictionary is then the same order as when we started, i.e. $v(D) = v(v(D)) = [d \wr b, b \wr f, d \wr c, k \wr f, f \wr a]$. ◇

4.4 Complexity analysis of constructing a normal form

The analysis of the time complexity of Algorithm 3 is mostly trivial, but there are some complications which might not be obvious at first.

Theorem 5. *The time complexity of Algorithm 3 is $O(t|D|^2)$ where $|D|$ is the number of rules in the dictionary, $t = \max_{u \in \Gamma}\{|u|\}$.*

Proof. The first interesting part is the outer loop at line 3. The loop iterates over all the rules in the dictionary ($O(|D|)$) and for each rule we may need to compare it with possibly all the earlier rules ($O(|D|)$), if Theorem 4 never prevents swaps in line 7. When line 7 do not prevent the swaps, we move to line 8 which involves determining the lexicographical ordering of the rules described in Section 4.2. This depends on the length of concatenation of the tokens in the rule since at worst, both tokens in each rule need to be compared against each other, which can be done in linear time. Thus, the comparison between rules are bounded by the longest token that can be produced by the dictionary i.e. the longest token in the token alphabet, $t = \max_{u \in \Gamma}\{|u|\}$. Therefore, the time complexity of Algorithm 3 is $O(t|D|^2)$ where $|D|$ is the number of rules in the dictionary and t is the length of the longest token in the alphabet Γ. □

In worst case scenario, each rule of a dictionary doubles the length of the of token generated by the previous rule. For example, consider dictionary $D = [a \wr a, aa \wr aa, aaaa \wr aaaa, \cdots]$. Then the value of t is thus bounded by $2^{|D|}$. However, clearly, line 7 would block any further comparisons at line 8 immediately in such case according to the restrictions in Theorem 4. Important to note is if we let this pattern continue until the last rule $r_{|D|} = u_{|D|} \wr v_{|D|}$, and let $u_{|D|} = u_{|D|-2}$ and $v_{|D|} = v_{|D|-3}$, then only the last rule in dictionary $|D|$ not be blocked by line 7 and thus t would be bound by $2^{|D|}$. This means that only the outer loop at line 3 and

the lexicographic comparison at line 8 would run and line 7 would only run once. In this case the algorithm runs in $O(t|D|)$ where t is bounded by $2^{|D|}$.

The algorithm is not efficient in the worst case scenario, but this worst case scenario is almost guaranteed to never happen outside of a controlled test environment since such a dictionary would not be useful in any real world application. To strengthen this argument, we can look at the average and maximum token length of the tokens used for two models in the OpenAI API, GPT-4o and GPT-4/3.5, which can be found using the GitHub repository by Kaito Sugimoto[1]. We modified the code to check for the average and maximum token length for both of these models. The average token length of the tokens is just over 5.5 and 6.3 symbols per token for the GPT-4o and GPT-4/3.5 models, respectively, and the maximum token length is 128 symbols for both models. This strengthens the claim that the worst case scenario is very unlikely to happen in practice, and the token length used in popular models today, if similar to these OpenAI models, would not cause any trouble for the algorithm. Thus, the expected performance by the algorithm is therefore $O(t|D|^2)$, where the dominant factor is the number of rules in the dictionary, which grows in polynomial time. The token length t is typically small in practical application, therefore, the overall complexity is mainly determined by the dictionary size in real-world scenarios.

4.5 Dictionary equivalence using the normal form

Having two dictionaries D and D', we are now able to check if they are equivalent by performing the algorithm in Algorithm 3 on both dictionaries, producing two proper normal form dictionaries $\nu(D)$ and $\nu(D')$. By conjecture, two dictionaries are equivalent if and only if $\nu(D) = \nu(D')$, which means that the normal form dictionaries have the same ordering of the merge rules. Calculating the normal form for both dictionaries involves first removing all rules that are not useful, which can be done in linear time. The number of rules in the dictionaries should then be the same for equivalent dictionaries. The algorithm in Algorithm 3 is then run on both dictionaries, which has a time complexity of $O(2t|D|^2)$. Since the number of rules in the dictionaries is the same, and the token alphabet Γ is the same for equivalent dictionaries, we can conclude that, since $t = \max_{u \in \Gamma}\{|u|\}$ i.e. t is bounded by the longest token in Γ, the time complexity of comparing whether two dictionaries are equivalent is $O(2t|D|^2) = O(t|D|^2)$, plus a linear time comparison between the two dictionaries, comparing rule by rule, which does not affect the time complexity.

[1] https://github.com/kaisugi/gpt4_vocab_list

5 Discussion and conclusion

In this thesis we have analyzed the time complexity of the indirect procedure to determine the equivalence between two proper dictionaries in Section 3.2. We concluded that when using the universal DFA illustrated in Figure 2 as the arbitrary DFA for the algorithm presented by Berglund et al. [1], the time complexity of the algorithm is $O(|\Gamma||D|^2)$, where $|\Gamma|$ is the token alphabet induced by dictionary D and $|D|$ is the number of rules in the dictionary. We also show that it is possible to construct a normal form of a proper dictionary and conjecture that it is unique for all equivalent dictionaries that contain the same rules, and thus it can be used to check if two dictionaries are equivalent. The time complexity of the normalization procedure runs in $O(t|D|^2)$, where t is bounded by the longest token in the token alphabet. While in theory the maximum token length of a dictionary is bounded by $O(2^{|D|})$, in practice the token lengths are rather small, e.g. for the OpenAI GPT-4o model, the length of the longest token is 128 symbols. We expect that other LLM models would have similar token lengths.

The indirect procedure is much less performant, as we can see in some practical examples. The GPT-4o model has a token alphabet of size $|\Gamma| = 199,997$ and the GPT-4/3.5 model has the token alphabet size of $|\Gamma| = 100,256$[8]. Since the maximum token length of these models is 128 symbols each, i.e. $t = 128$, we can conclude that the normal form algorithm is more efficient than the indirect procedure for the dictionaries in these models and, in theory, for other model dictionaries as well, but this is not something we have tested.

Future work may involve formulating a proof of correctness for the Algorithm 3 by proving that the normal form is unique for all dictionaries in the same equivalence class. Also expanding this normal form to cover dictionaries that do not contain the same rules. Further, it would be interesting to do a deeper theoretical and experimental analysis of the time complexity, with a focus on real-world dictionaries as well as comparing this to the performance of the indirect procedure. Finally, it would be interesting to see if we can construct a normal form for improper dictionaries and, if so, how to construct it and their applications.

References

[1] Berglund, M., Martens, W., and van der Merwe, B. (2024). Constructing a BPE Tokenization DFA. In Fazekas, S. Z., editor, *Implementation and Application of Automata*, pages 66–78, Cham. Springer Nature Switzerland.

[2] Berglund, M. and Van Der Merwe, B. (2023). Formalizing BPE Tokenization. *Electronic Proceedings in Theoretical Computer Science*, 388:16–27.

[3] Hopcroft, J. E. and Karp, R. M. (1971). *A Linear Algorithm for Testing Equivalence of Finite Automata*.

[4] Kudo, T. and Richardson, J. (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, Brussels, Belgium. Association for Computational Linguistics.

[5] Luong, T., Sutskever, I., Le, Q., Vinyals, O., and Zaremba, W. (2015). Addressing the rare word problem in neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, page 11–19, Beijing, China. Association for Computational Linguistics.

[6] Sennrich, R., Haddow, B., and Birch, A. (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)*, page 1715–1725, Berlin, Germany. Association for Computational Linguistics.

[7] Stollenwerk, F. (2023). Training and evaluation of a multilingual tokenizer for gpt-sw3. (arXiv:2304.14780). arXiv:2304.14780 [cs].

[8] Yang, J., Wang, Z., Lin, Y., and Zhao, Z. (2024). Problematic tokens: Tokenizer bias in large language models. (arXiv:2406.11214). arXiv:2406.11214 [cs].

A First Appendix

In Algorithm 4, we see the algorithm that turns one rule $u \wr v$ in a dictionary D to a DFA A.

Algorithm 4 Applying a Merge

Input: a universal DFA $A = (Q, \Gamma, q_0, \delta, F)$, and the rule, $u \wr v$, in the dictionary D
Let the set S be the set of all triples $(s_1, s_2, s_3) \in Q$ such that $\delta(s_1, u) = s_2$ and $\delta(s_2, v) = s_3$
Let S_2 be the set of all s_2 such that $(s_1, s_2, s_3) \in S$
for each set in S **do**
    add new transition $\delta(s_1, uv) = s_3$
end for
add the token uv to Γ
for each state s_2 in S_2 **do**
    create a new state $new(s_2)$ and add it to Q
    if $s_2 \in F$ **then**
        add $new(s_2)$ to F
    end if
    $\Gamma_m = \{v\}$ if $u \neq v$ and $\Gamma_m = \{uv, v\}$ otherwise
    for each token $w \in \Gamma \setminus \Gamma_m$ **do**
        if $\delta(s_2, w)$ exists **then**
            $\delta(new(s_2), \alpha) = \delta(s_2, \alpha)$
        end if
    end for
end for
for each state s in Q **do**
    if $\delta(s, u) \in S_2$ **then**
        set the transition $\delta(s, u) = new(\delta(s, u))$
    end if
end for
Output: the resulting DFA

The algorithm seen in Algorithm 4 works by iteratively merging the rules r_i in the dictionary D into the DFA A_i. The initial DFA used is the universal DFA A_0 for the language Σ^*, which is a DFA that accepts all the possible strings produced by the alphabet Σ. We take all of the triplets (s_1, s_2, s_3) where the first state s_1 has a transition with the token u leading to state s_2, and the state s_2 has a transition with the token v leading to state s_3. Using this, we create a new transition, skipping s_2, from s_1 to s_3 with the token uv. For every unique state s_2 in the list of triplets, we create a new state $new(s_2)$. This only happens once, since when starting from the universal DFA, all tokens of a specific kind always leads to the same state, and there can never be two or more different states that has an incoming transition when reading some symbol. Also, for every s_2 in the list of triplets, we check if the two tokens u and v are the same. If they are, we copy all transitions from s_2 to the new state $new(s_2)$, except for when reading v. If they are not the same, we do the same but also ignore the transition with the

token uv from s_2. Finally, we go through all the states in the DFA and check if there are any transitions that lead to s_2 with the token u. For those that do, we replace the destination of the transition with the new state $new(s_2)$ to avoid having a transition that leads to a state that can read the token v after reading the token u since these should loop back to the new state.