A Formal Perspective on Byte-Pair Encoding

Vilém Zouhar[E] **Clara Meister**[E] **Juan Luis Gastaldi**[E] **Li Du**[J]
Tim Vieira[J] **Mrinmaya Sachan**[E] **Ryan Cotterell**[E]

ETH Zürich[E] Johns Hopkins University[J]
{vzouhar,meistecl,gjuan,msachan,ryan.cotterell}@ethz.ch
{leodu,timv}@cs.jhu.edu

Abstract

Byte-Pair Encoding (BPE) is a popular algorithm used for tokenizing data in NLP, despite being devised initially as a compression method. BPE appears to be a greedy algorithm at face value, but the underlying optimization problem that BPE seeks to solve has not yet been laid down. We formalize BPE as a combinatorial optimization problem. Via submodular functions, we prove that the iterative greedy version is a $\frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1-e^{-\sigma(\boldsymbol{\mu}^\star)})$-approximation of an optimal merge sequence, where $\sigma(\boldsymbol{\mu}^\star)$ is the total backward curvature with respect to the optimal merge sequence $\boldsymbol{\mu}^\star$. Empirically the lower bound of the approximation is ≈ 0.37.

We provide a faster implementation of BPE which improves the runtime complexity from $\mathcal{O}(NM)$ to $\mathcal{O}(N \log M)$, where N is the sequence length and M is the merge count. Finally, we optimize the brute-force algorithm for optimal BPE using memoization.

1 Introduction

Byte-Pair Encoding (BPE) is a popular technique for building and applying an encoding scheme to natural language texts. It is one the most common tokenization methods used for language models (Radford et al., 2019; Bostrom and Durrett, 2020; Brown et al., 2020; Scao et al., 2022) as well as for various other conditional language modeling tasks, e.g., machine translation (Ding et al., 2019) and chatbots (Zhang et al., 2020). Despite having been popularized by Sennrich et al. (2016) in NLP as a tokenization scheme, BPE has its roots in the compression literature, where Gage (1994) introduce the method as a faster alternative to Lempel–Ziv–Welch (Welch, 1984; Cover and Thomas, 2006, 13.4). However, the ubiquity of BPE notwithstanding, the formal underpinnings of the algorithm are underexplored, and there are no existing proven guarantees about BPE's performance.

The training and applying of BPE are traditionally presented as greedy algorithms, but the exact optimization problems they seek to solve are neither presented in the original work of Gage (1994) nor in the work of Sennrich et al. (2016). We fill this void by offering a clean formalization of BPE training as maximizing a function we call compression utility[1] over a specific combinatorial space, which we define in Definition 2.3. Unexpectedly, we are then able to prove a bound on BPE's approximation error using total backward curvature $\sigma(\boldsymbol{\mu}^\star)$ (Zhang et al., 2015). Specifically, we find the ratio of compression utilities between the greedy method and the optimum is bounded below by $\frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1-e^{-\sigma(\boldsymbol{\mu}^\star)})$, which we find empirically ≈ 0.37 for $\widehat{\sigma}(\boldsymbol{\mu}^\star) = 2.5$. Our proof of correctness hinges on the theory of submodular functions (Krause and Golovin, 2014; Bilmes, 2022).[2] Indeed, we are able to prove that compression utility is a special kind of submodular function (Malekian, 2009) over a constrained space. And, despite the presence of the length constraint, which we expound upon formally in §3, we are able to prove a similar bound to $1-1/e$ as in the unconstrained case (Alaei et al., 2010).

Additionally, we give a formal analysis of greedy BPE's runtime and provide a speed-up over the original implementation (Gage, 1994; Sennrich et al., 2016). Our runtime improvement stems from the development of a nuanced data structure that allows us to share work between iterations of the greedy procedure and that lends itself to an amortized analysis. Specifically, given a string with N characters with a desired merge count of M (usually $N \gg M$), our implementation runs in $\mathcal{O}(N \log M)$, an improvement over the $\mathcal{O}(NM)$-time algorithm presented by Sennrich et al. (2016) and the $\mathcal{O}(N \log N)$ analysis presented by Kudo

[1] How much space the compression saves (Definition 2.5).

[2] The proof further relies on a specific property of problem which BPE optimizes that we term hierarchical sequence submodularity. Hierarchical sequence submodularity neither follows from nor implies sequence submodularity, but, nevertheless, bears some superficial similarity to sequence submodularity—hence, our choice of name.

and Richardson (2018). Finally, our formalism allows us to construct an exact program for computing an optimal solution to the BPE training problem. Unfortunately, the algorithm runs in exponential time, but it is still significantly faster than a naïve brute-force approach.

Our work should give NLP practitioners confidence that BPE is a wise choice for learning a subword vocabulary based on compression principles. In general, such constrained submodular maximization problems are hard (Lovász, 1983). While we do not have a proof that the BPE problem specifically is NP-hard, it does not seem likely that we could find an efficient algorithm for the problem. Regarding the runtime, our implementation of greedy BPE runs nearly linearly in the length of the string which would be hard to improve unless we plan to not consider the entire string.

2 Formalizing Byte-Pair Encoding

We first provide a brief intuition for the BPE training problem and the greedy algorithm that is typically employed to solve it. Then, we will develop a formalization of BPE using the tools of combinatorial optimization, rather than as a procedure.[3]

2.1 A Worked Example

merge 1	p i c k e d	p i c k l e d	p i c k l e s
merge 2	pi c k e d	pi c k l e d	pi c k l e s
merge 3	pi ck e d	pi ck l e d	pi ck l e s
merge 4	pick e d	pick l e d	pick l e s
merge 5	pick ed	pick l ed	pick l e s
final	pick ed	pickl ed	pickl e s

Example 1: Compression of the text *picked pickled pickles* using 5 greedy merges according to the greedy BPE algorithm. The most frequently occurring pair of vocabulary items is highlighted and subsequently merged. The merge sequence is ⟨*[p,i], [c,k], [pi,ck], [e,d], [pick,l]*⟩ (notation simplified for clarity).

Consider the string in Example 1: *picked pickled pickles*. We wish to create a compact representation of this string, where compactness is quantified in terms of the number of symbols (i.e., vocabulary units) required to precisely encode the string. The free parameter is the vocabulary that we will use to construct this representation, albeit the total size of the chosen vocabulary is often a constraint.[4] In our example, let's assume we are allowed a maximum number of 13 symbols in the vocabulary[5] with which we can encode our string. The question is: "How can we select these symbols to achieve our goal of compactness under this constraint?"

Let us first consider the simple choice of using all the characters present in the string as our vocabulary: This scheme leads to a representation with a length of 22 units, including spaces. In order to decrease this length (while retaining all information present in the original string), we would need to add an additional symbol to our vocabulary: one with which we can replace co-occurrences of two symbols. But how should we choose this entry? One strategy—the one employed by the BPE algorithm—is to use the concatenation of the adjacent units a b that occur with the highest frequency in our string; all occurrences of these adjacent units could then be replaced with a single new unit ab. We refer to this as a **merge**, which we later define and denote formally as $[a, b]$. In Example 1, the first merge is $[p, i]$, and leads to a representation of length 19 with vocabulary size of 9+1. We can iteratively repeat the same process; the application of 5 total merges results in the vocabulary units *pick*, *pickl*, *ed*, *e*, and *s*. These **subwords**[6] allow us to represent our original string using just 9+1 symbols. If we continued merging, the text representation would become shorter (in terms of number of symbols required to create the representation) but the merge count (and vocabulary size) would grow. Therefore, the number of merges M, or also the merge count, is a hyperparameter to the whole procedure. The procedure outlined above is exactly the greedy algorithm for BPE proposed by Gage (1994). We provide a minimal implementation in Python in Code 1.

We will define the compression gain of a merge at any given step of the algorithm, corresponding to the number of occurrences where a merge can be applied. The compression gain of a merge does not always correspond to the frequency of adjacent merge components in that string, due to possible overlaps. Consider, for instance, the string *aaa* and

[3] Note that we are interested in the optimality of the algorithm for creating the subword vocabulary in terms of compression and not the optimality of the encoding in terms of coding-theoretic metrics such as *efficiency*. This aspect of BPE is explored by Zouhar et al. (2023).

[4] We require a unique encoding for each item, which implies that encoding size will be dependent on the total number of vocabulary items (e.g. the dimension of a one-hot encoding or the number of bits required to encode the text).

[5] Typically, all the symbols in Σ are part of the vocabulary so that all texts can be represented, even with lower efficiency.

[6] The term **subword** corresponds to a merge yield (Definition 2.4). We use 'subword' and 'merge' interchangeably.

the merge $[a, a]$. The frequency of aa is 2, but the merge can be applied only once ($[a, a]a$). While Gage (1994) and Sennrich et al. (2016) admit overlapping pair counts, Kudo and Richardson (2018)'s popular implementation adjusts the algorithm to disregard the overlaps. We stick to the latter, which is more suitable from the optimization standpoint adopted here.

```
1  from collections import Counter
2  from typing import Union, Tuple, List
3
4  def bpe(xs: Union[str, List], V: int):
5    for _ in range(V):
6      pairs = Counter(zip(xs, xs[1:]))
7      top_pair = pairs.most_common(1)[0][0]
8      xs = merge(list(xs), top_pair)
9    return xs
10
11 def merge(xs: List, pair: Tuple):
12   ys = []
13   while xs:
14     if tuple(xs[:2]) == pair:
15       ys.append(pair)
16       xs = xs[2:]
17     else:
18       ys.append(xs.pop(0))
19   return ys
```

Code 1: A minimal implementation of Sennrich et al.'s (2016) greedy algorithm for BPE in Python. See Code 2 for a version with overlap-adjusted counts.

2.2 Merges

The fundamental building block of the BPE problem is a merge, which we define formally below. Informally, a merge is the action of creating a new symbol out of two existing ones. Out of convention, we also refer to the resulting object as a merge.

Definition 2.1. *Let Σ be an alphabet, a finite, non-empty set. The set of all **merges** over Σ is the smallest set of pairs Υ_Σ with the following closure property:*

- $\sigma \in \Sigma \Longrightarrow \sigma \in \Upsilon_\Sigma$ *(called **trivial merges**);*
- $\mu', \mu'' \in \Upsilon_\Sigma \Longrightarrow [\mu', \mu''] \in \Upsilon_\Sigma$

*where we denote the non-trivial elements of Υ_Σ as $\mu = [\mu', \mu'']$. A **merge sequence** is a sequence of merges, which we denote $\boldsymbol{\mu} = \langle \mu_1, \ldots, \mu_N \rangle \in \Upsilon_\Sigma^*$.*[7]

It is perhaps easiest to understand the concept of a merge through an example.

Example 2.2. *Given the alphabet $\Sigma = \{a, b, c\}$, the following are some of the elements of Υ_Σ: $[a, b]$, $[a, [a, b]]$, and $[[a, b], [a, c]]$. We obtain a merge sequence by arranging these merges into an ordering $\boldsymbol{\mu} = \langle [a, b], [a, [a, b]], [[a, b], [a, c]] \rangle \in \Upsilon_\Sigma^*$.*

[7] $(\cdot)^*$ is the Kleene closure.

Note that the strings corresponding to the merges in a merge sequence—along with the characters that make up the set of trivial merges—determine a **vocabulary**, to be used in downstream applications.[8] The greedy BPE algorithm constructs a merge sequence iteratively by picking each merge as the pairing of neighbouring symbols in the current sequence of symbols that is being processed. For instance, the sequence $\boldsymbol{\mu}$ in Example 2.2 is not valid since it does not contain the merge $[a, c]$ before the third element $[[a, b], [a, c]]$.

Definition 2.3. *We define a merge sequence $\boldsymbol{\mu} = \langle \mu_1, \ldots, \mu_N \rangle \in \Upsilon_\Sigma^*$ to be **valid** if, for every μ_n, it holds that $\mu_n = [\mu', \mu'']$, where for $\mu \in \{\mu', \mu''\}$, $\mu = \mu_k$ with $k < n$, or $\mu \in \Sigma$. We denote the set of valid merge sequences $\mathcal{M}_{\Upsilon_\Sigma}$.*

Note that $\mathcal{M}_{\Upsilon_\Sigma}$ is closed under concatenation, i.e., for two valid merge sequences $\boldsymbol{\mu}', \boldsymbol{\mu}'' \in \mathcal{M}_{\Upsilon_\Sigma}$, we have that $\boldsymbol{\mu}'\boldsymbol{\mu}'' \in \mathcal{M}_{\Upsilon_\Sigma}$,[9] where we use $\boldsymbol{\mu}\boldsymbol{\mu}'$ to denote the sequence concatenation of $\boldsymbol{\mu}$ and $\boldsymbol{\mu}'$.

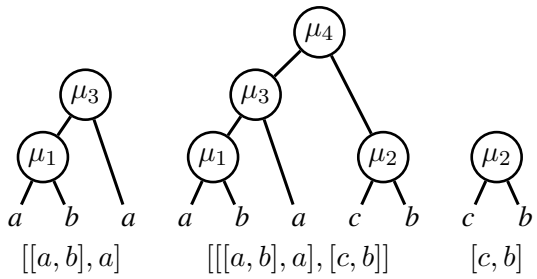

Figure 1: Application of the merge sequence $\boldsymbol{\mu} = \langle [a, b], [c, b], [[a, b], a], [[[a, b], a], [c, b]] \rangle$ on the string $\boldsymbol{x} = abaabacbcb$. The result can be represented as an ordered forest. Each tree is associated with a subword in the text: aba, $abacb$, and cb.

Applying Merge Sequences. Given some string $\boldsymbol{x} \in \Sigma^*$, we can derive the representation of that string according to the merge sequence $\boldsymbol{\mu} = \langle \mu_1, \ldots, \mu_N \rangle$ by iteratively **applying** each merge μ_n. Note that by the definition of Υ_Σ^*, we can trivially lift a string $\boldsymbol{x} = \langle \sigma_1, \sigma_2, \ldots \rangle$ to a merge sequence by treating each of its characters $\sigma_i \in \Sigma$ as merges. Thus, we define this procedure more generally in terms of some arbitrary $\overline{\boldsymbol{\mu}} \in \Upsilon_\Sigma^*$. Concretely, we denote the application of a merge μ_n to $\overline{\boldsymbol{\mu}}$ as $\textsc{Apply}_{\mu_n}(\overline{\boldsymbol{\mu}})$. As suggested by Code 1 (line 11), this action consists of replacing all $\overline{\mu}_k, \overline{\mu}_{k+1}$ in $\overline{\boldsymbol{\mu}}$ such that $(\overline{\mu}_k, \overline{\mu}_{k+1}) = \mu_n$ by μ_n itself, in a left-to-right fashion. We thus obtain a

[8] I.e., the size of the vocabulary is $|\boldsymbol{\mu}| + |\Sigma|$.

[9] The merge sequence can contain the same merges multiple times and still be valid. Only the later occurrences of the merge will not reduce the representation size.

new $\overline{\mu} \in \Upsilon^*_\Sigma$, to which a new single merge can be applied. We lift APPLY to a merge sequence $\boldsymbol{\mu}$ by simply repeating the application of APPLY on $\overline{\boldsymbol{\mu}}^{(n)}$ for the successive μ_n in $\boldsymbol{\mu}$; accordingly, we denote this procedure as $\text{APPLY}_{\boldsymbol{\mu}}(\overline{\boldsymbol{\mu}})$. As a result, we obtain $\overline{\boldsymbol{\mu}}^{(|\boldsymbol{\mu}|)}$, which is a non-overlapping ordered forest, i.e., a partial bracketing of the original string $\boldsymbol{x}$. We provide an example in Fig. 1. Note that the application of the merge sequence is deterministic.

String Yields. We now define a conceptually reverse operation to applying merges, i.e., deriving a string from structured $\overline{\boldsymbol{\mu}}^{(n)}$.

Definition 2.4. *The **yield** of a single $\overline{\mu} \in \Upsilon_\Sigma$, denoted as $\text{YIELD}(\overline{\mu})$, is defined recursively:*

$$\text{YIELD}(\overline{\mu}) = \begin{cases} \text{YIELD}(\overline{\mu}')\text{YIELD}(\overline{\mu}'') & \textbf{\textit{if}}\ \overline{\mu} = [\overline{\mu}', \overline{\mu}''] \\ \mu & \textbf{\textit{if}}\ \overline{\mu} \in \Sigma \end{cases} \tag{1}$$

As an example, $\text{YIELD}([[a, a], [[c, b], c]])$ is $aacbc$. For a given $\overline{\boldsymbol{\mu}}$, YIELD is applied sequentially. The resulting characters can then be concatenated to derive a single string. The yield operation can also be used to derive vocabulary units—often referred to as subwords; explicitly, the yields of individual merges in a sequence $\boldsymbol{\mu}$ can be used to form a vocabulary.

Strictly speaking, in Sennrich et al.'s (2016) implementation of BPE, the elements of the merge sequences $\boldsymbol{\mu}$ are not of the form $\mu_n = [\mu', \mu''] \in \Upsilon_\Sigma$, but rather $\mu_n = [\text{YIELD}(\mu'), \text{YIELD}(\mu'')] \in \Sigma^* \times \Sigma^*$, i.e., rather than consisting of prior merges as in our formalization, the merges of Sennrich et al.'s (2016) consist of the yields of those merges. This introduces an ambiguity with respect to our formalization since: for a given merge sequence in that implementation, more than one sequence $\boldsymbol{\mu} \in \Upsilon^*_\Sigma$ could correspond, some of which would not be valid. As an example, consider the sequence $\langle [a, b], [ab, c], [abc, d] \rangle$ which could correspond to either $\langle [a, b], [[a, b], c], [[[a, b], c], d] \rangle$ or $\langle [a, b], [[a, b], c], [[a, [b, c]], d] \rangle$, the last of which is invalid. However, it turns out that this is not an issue for us: by construction, the successive elements of the sequence are determined by the previous ones (cf. Alg. 1), which means that, in fact there is no ambiguity, and the merge sequences in Sennrich et al.'s (2016) implementation always correspond to what our formalization defines as a valid merge sequence.

Algorithm 1 Iterative Greedy BPE (slow).
Inputs: sequence $\boldsymbol{x}$, merge count M
Output: merge sequence $\boldsymbol{\mu}$, tokenized sequence $\boldsymbol{x}$
PAIRFREQ are non-overlapping pair frequencies

```
μ ← ⟨⟩
for i in {0, ..., M} do
    μ ← argmax_{(μ', μ'') ∈ set(x)^2} PAIRFREQ(x, (μ', μ''))
    x ← APPLY(μ, x)
    μ ← μ ∘ ⟨μ⟩
end for
return μ, x
```

2.3 The BPE Training Optimization Problem

We now define the BPE training task as a combinatorial optimization problem. The objective we seek to optimize is the compression utility of the chosen merge sequence (taken with respect to a string), which we define below.

Definition 2.5. *Let $\boldsymbol{x} \in \Sigma^*$ be a string. We define the **compression utility** of a valid merge sequence $\boldsymbol{\mu}$ applied to $\boldsymbol{x}$ as the following function:*

$$\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}) = |\boldsymbol{x}| - |\text{APPLY}_{\boldsymbol{\mu}}(\boldsymbol{x})| \tag{2}$$

Note that for any merge sequence $\boldsymbol{\mu}$, $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}) \geq 0$ and we take $\kappa_{\boldsymbol{x}}(\langle\rangle) = 0$. Then, for any merge sequence $\boldsymbol{\mu}' = \langle \mu'_1, \ldots, \mu'_{|\boldsymbol{x}|-1} \rangle$ of length $|\boldsymbol{x}| - 1$ where every merge produces replacements, we have $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}') = |\boldsymbol{x}| - 1$ (see proof of Theorem 4.2).

We can further define the **compression gain** of two merge sequences with respect to each other.

Definition 2.6. *The **compression gain** of $\boldsymbol{\mu}'$ with respect to a sequence $\boldsymbol{\mu}$, denoted as $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}' \mid \boldsymbol{\mu})$, is defined as*

$$\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}\boldsymbol{\mu}') - \kappa_{\boldsymbol{x}}(\boldsymbol{\mu}). \tag{3}$$

Similarly, the compression gain of a single merge μ with respect to a sequence $\boldsymbol{\mu}$, denoted as $\kappa_{\boldsymbol{x}}(\mu \mid \boldsymbol{\mu})$, is defined as $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}\mu) - \kappa_{\boldsymbol{x}}(\boldsymbol{\mu})$.

We use the compression gain to later make a sequence of observations which leads to proving the function submodularity and eventually its approximation bound of the BPE training algorithm.

Now, armed with Definition 2.5, we can formally state our optimization problem. In words, *we seek to find a valid merge sequence $\boldsymbol{\mu}$ with length of M that maximizes the compression utility $\kappa_{\boldsymbol{x}}(\cdot)$ for a pre-specified string $\boldsymbol{x} \in \Sigma^*$.* We write this combinatorial optimization problem more formally

as follows:[10]

$$\boldsymbol{\mu}^{\star} = \underset{\substack{\boldsymbol{\mu} \in \mathcal{M}_{\Upsilon_\Sigma} \\ |\boldsymbol{\mu}| = M}}{\operatorname{argmax}} \kappa_{\boldsymbol{x}}(\boldsymbol{\mu}) \tag{4}$$

The most common procedure found in the NLP literature for solving Eq. (4) is a greedy algorithm (Gage, 1994; Sennrich et al., 2016). The implementation of Gage's (1994) algorithm presented by Sennrich et al. (2016) runs in $\mathcal{O}(NM)$ time ($N = |\boldsymbol{x}|, M = |\boldsymbol{\mu}^{\star}|$). We describe this greedy algorithm in detail in §3 and provide a novel theoretical result: *The algorithm comes with a* $\frac{1}{\sigma(\boldsymbol{\mu}^{\star})}(1 - e^{-\sigma(\boldsymbol{\mu}^{\star})})$ *bound on its approximation error of Eq.* (4). In §4, we further offer an asymptotic speed-up to Sennrich et al.'s (2016) algorithm, reducing its runtime to $\mathcal{O}(N \log M)$. Finally, for completeness, we offer an exact program for finding an optimal valid merge sequence in §5. While this algorithm runs in exponential time, which prevents it to be used in real applications, it is still faster than the brute-force counterpart.

3 A Greedy Approximation of BPE

We demonstrate that, for any string $\boldsymbol{x} \in \Sigma^*$, the following bound holds

$$\frac{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^{\dagger})}{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^{\star})} \geq \frac{1}{\sigma(\boldsymbol{\mu}^{\star})}(1 - e^{-\sigma(\boldsymbol{\mu}^{\star})}) \tag{5}$$

where, as in the previous section, $\boldsymbol{\mu}^{\dagger}$ is the valid merge sequence output by the greedy algorithm and $\boldsymbol{\mu}^{\star}$ is an optimal valid merge sequence. To prove this bound, we rely heavily on the theory of submodularity (Krause and Golovin, 2014; Bilmes, 2022).

3.1 Properties of Compression Utility (κ)

We start by proving some useful facts about the compression utility function $\kappa_{\boldsymbol{x}}$. Specifically, we first show that $\kappa_{\boldsymbol{x}}$ is a specific type of monotone non-decreasing submodular sequence function, which we make precise in the following definitions.

Definition 3.1. *A real-valued function f over valid merge sequences is* ***monotone non-decreasing*** *if, for all $\boldsymbol{\mu} \in \mathcal{M}_{\Upsilon_\Sigma}$ and for all $n \in \mathbb{N}$, it holds that $f(\boldsymbol{\mu}_{<n}) \geq f(\boldsymbol{\mu}_{<n-1})$, where $\boldsymbol{\mu}_{<n} \stackrel{\text{def}}{=} \langle \mu_1, \ldots, \mu_{n-1} \rangle$.*

[10] In practice, it does not happen that $|\boldsymbol{x}| < M$ and so we use $|\boldsymbol{\mu}^{\star}| = M$ for convenience instead of $|\boldsymbol{\mu}^{\star}| \leq M$.

Proposition 3.2. *Let $\kappa_{\boldsymbol{x}}$ be the compression utility function. Then, for a fixed $\boldsymbol{x} \in \Sigma^*$, $\kappa_{\boldsymbol{x}}(\cdot)$ is monotone (Definition 3.1).*

Proof. For all $n \in \mathbb{N}$, we have that $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}_{<n}) = \kappa_{\boldsymbol{x}}(\boldsymbol{\mu}_{<n-1}) + \underbrace{\kappa_{\boldsymbol{x}}(\mu_n \mid \boldsymbol{\mu}_{<n-1})}_{\geq 0}$. It follows that $\kappa_{\boldsymbol{x}}(\cdot)$ is monotone non-decreasing. ■

Next, we turn to a definition of sequence submodularity from Alaei et al. (2010). In contrast to Alaei et al.'s (2010) definition, we add the additional constraint that a merge-sequence function must take a *valid* merge sequence as an argument.

Definition 3.3. *A real-valued function f over valid merge sequences is* ***submodular*** *if, for all $\boldsymbol{\mu}, \boldsymbol{\mu}' \in \mathcal{M}_{\Upsilon_\Sigma}$ such that $\boldsymbol{\mu}' \preccurlyeq \boldsymbol{\mu}$,*[11] *and for all $\nu \in \Upsilon_\Sigma$ such that both $\boldsymbol{\mu}'\nu$ and $\boldsymbol{\mu}\nu$ are valid, we have*

$$f(\nu \mid \boldsymbol{\mu}') \geq f(\nu \mid \boldsymbol{\mu}). \tag{6}$$

Proposition 3.4. *Let $\kappa_{\boldsymbol{x}}$ be the compression utility function. Then, for a fixed $\boldsymbol{x} \in \Sigma^*$, $\kappa_{\boldsymbol{x}}(\cdot)$ is submodular (Definition 3.3) when the domain is restricted to the set of valid merges $\mathcal{M}_{\Upsilon_\Sigma}$.*

Proof. Let $\boldsymbol{\mu}, \boldsymbol{\mu}' \in \mathcal{M}_{\Upsilon_\Sigma}$ such that $\boldsymbol{\mu}' \preccurlyeq \boldsymbol{\mu}$, and let $\nu = [\nu', \nu'']$ be any merge such that $\boldsymbol{\mu}\nu, \boldsymbol{\mu}'\nu \in \mathcal{M}_{\Upsilon_\Sigma}$. First, notice that, once a merge μ_n in a merge sequence $\boldsymbol{\mu}$ is applied, the number of occurrences of μ_n in $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}_{\leq n})$ cannot be increased by any sequence of further applications, because all submerges of μ_n where applied exhaustively (i.e., to all consecutive occurrences of their immediate submerges). Now, from $\boldsymbol{\mu}'\nu \in \mathcal{M}_{\Upsilon_\Sigma}$, it follows that both ν' and ν'' are in $\boldsymbol{\mu}'$. Therefore, the number of occurrences ν' and ν'', and *a fortiori* of successive occurrences of them, cannot be greater in $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu})$ than in $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}')$, and hence $\kappa_{\boldsymbol{x}}(\nu \mid \boldsymbol{\mu}) \leq \kappa_{\boldsymbol{x}}(\nu \mid \boldsymbol{\mu}')$, which proves the submodularity of $\kappa_{\boldsymbol{x}}$ over $\mathcal{M}_{\Upsilon_\Sigma}$. ■

In the context of compression, the submodularity property means, that the compression gain achieved after adding a specific merge to a merge sequence can never increase with merge sequence length. However, the requirement that the added merge does not create an invalid merge sequence is important. We highlight this importance in the following example.

[11] I.e., we have that $\boldsymbol{\mu}'$ is a prefix of $\boldsymbol{\mu}$.

Example 3.5. *Consider* $\Sigma = \{a, b, c, d, e\}$*, the string* $\boldsymbol{x} = aabcde$*, and the valid merge sequences* $\boldsymbol{\mu}' = \langle [a, a] \rangle$ *and* $\boldsymbol{\mu} = \langle [a, a], [c, d] \rangle$*. Note that* $\boldsymbol{\mu}' \preccurlyeq \boldsymbol{\mu}$*. These merge sequences have compression utilities* $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}') = 6-5 = 1$ *and* $\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}) = 6-4 = 2$*, respectively. Next, consider the merge sequence* $\boldsymbol{\nu} = \langle [b, [c, d]], [[b, [c, d]], e] \rangle$*. Now,* $\kappa_{\boldsymbol{x}}(\boldsymbol{\nu} \mid \boldsymbol{\mu}') = 0$ *and* $\kappa_{\boldsymbol{x}}(\boldsymbol{\nu} \mid \boldsymbol{\mu}) = 2$*, which violates submodularity because* $\mu' \preccurlyeq \boldsymbol{\mu}$*. What went wrong? The problem is that* $\boldsymbol{\mu\nu}$ *is not a* valid *merge sequence.*

In order to formally prove our desired guarantee regarding the approximation bound of the greedy BPE algorithm, it is not enough that compression utility is sequence submodular over valid merge sequences. For this reason, we identified another property of the compression utility function that allows us to push through our result.

Definition 3.6. *We define the following* ***partial order on merges****. For merges* $\mu, \mu' \in \Upsilon_\Sigma$*, we say* $\mu' \subset \mu$ *iff* μ' *is a submerge of* μ*. The merge* μ' *is a* ***submerge*** *of* $\mu = [\mu_1, \mu_2]$ *iff:*

- $\mu_1 = \mu'$*, or,* $\mu_2 = \mu'$*, or*
- $\mu' \subset \mu_1$*, or* $\mu' \subset \mu_2$*.*

Definition 3.7. *A real-valued function over valid merge sequences is* ***hierachically sequence submodular*** *if, for every valid merge sequence of the form* $\boldsymbol{\mu}'\nu'\boldsymbol{\mu}\nu$ *where* $\nu' \subset \nu$ *according to the partial order given in Definition 3.6, we have that*

$$f(\nu' \mid \boldsymbol{\mu}') \geq f(\nu \mid \boldsymbol{\mu}'\nu'\boldsymbol{\mu}). \tag{7}$$

Note that hierarchical sequence submodularity is a different concept from function modularity, described in Definition 3.3. Indeed, in the case of functions over valid merge sequences, neither submodularity nor hierarchical sequence submodularity implies the other. To see this, note that roughly speaking, submodularity describes the difference in the value of a function when the same element is given as an argument, albeit conditioned on the presence of two different (but related) other arguments. However, if the same argument is considered in Eq. (7), we have

$$\kappa_{\boldsymbol{x}}(\nu' \mid \boldsymbol{\mu}') \geq \kappa_{\boldsymbol{x}}(\nu' \mid \boldsymbol{\mu}'\nu'\boldsymbol{\mu}) = 0, \tag{8}$$

which is a trivial bound due to the non-negativity of $\kappa_{\boldsymbol{x}}(\cdot)$. The naming is inspired by the fact we require the partial order over merges, which creates the hierarchy.

Proposition 3.8. *Let* $\kappa_{\boldsymbol{x}}$ *be the compression utility function. Then, for a fixed* $\boldsymbol{x} \in \Sigma^*$*,* $\kappa_{\boldsymbol{x}}(\cdot)$ *is hierarchically submodular (Definition 3.3) when the domain is restricted to the set of valid merges* $\mathcal{M}_{\Upsilon_\Sigma}$*.*

Proof. Let $\boldsymbol{x} \in \Sigma^*$ be a string and $\boldsymbol{\mu}$, $\boldsymbol{\mu}'$ be valid merge sequences. Furthermore, let ν, ν' be merges such that $\nu' \subset \nu$ and $\boldsymbol{\mu}'\nu'\boldsymbol{\mu}\nu$ is itself a valid merge sequence. Combinatorially, $\kappa_{\boldsymbol{x}}(\nu \mid \boldsymbol{\mu}'\nu'\boldsymbol{\mu})$ is the number of replacements made in $\boldsymbol{x}$ by the single merge of ν, after applying $\boldsymbol{\mu}'\nu'\boldsymbol{\mu}$. However, since $\nu' \subset \nu$, every new tree in $\boldsymbol{x}$ resulting from that by applying ν must have ν' as a descendant. Thus, $\kappa_{\boldsymbol{x}}(\nu' \mid \boldsymbol{\mu}')$, which is the number of new nodes in the forest created by applying ν', must be at least equal to $\kappa_{\boldsymbol{x}}(\nu \mid \boldsymbol{\mu}'\nu'\boldsymbol{\mu})$, if not greater. ■

Proposition 3.8 gives us a different notion of submodularity, which is important for the proof of the greedy BPE training guarantee. As an illustrative example of the proposition, we return to Fig. 1. In this case, $\boldsymbol{\mu}' = \langle [a, b] \rangle$, $\nu' = [[a, b], a]$, $\boldsymbol{\mu} = \langle [c, b] \rangle$, $\nu = [[[a, b], a], [c, b]]$. Clearly, $\nu' \subset \nu$ and ν' appears twice, while ν only once.

Finally, we adapt the definition of total backward curvature from (Zhang et al., 2015) to our needs. Intuitively, the total backward curvature is related to how much the utility of $\boldsymbol{\mu}$ can decrease if ν is applied before, at the beginning.

Definition 3.9. *The* ***total backward curvature*** *of the compression utility function* κ *with respect to an optimal merge sequence* $\boldsymbol{\mu}^\star$ *is denoted with* $\sigma(\boldsymbol{\mu}^\star)$*:*

$$\sigma(\boldsymbol{\mu}^\star) = \max_{\substack{\boldsymbol{\mu} \in \Upsilon_\Sigma^* \\ |\boldsymbol{\mu}| \leq M}} \left\{ 1 - \frac{\kappa(\boldsymbol{\mu}\boldsymbol{\mu}^\star) - \kappa(\boldsymbol{\mu}^\star)}{\kappa(\boldsymbol{\mu})} \right\}. \tag{9}$$

3.2 The Greedy Algorithm for BPE

In words, the greedy algorithm proceeds as follows: For each of the M iterations, the algorithm chooses the next merge that is both valid and (locally) maximizes the objective in Eq. (4). We give pseudocode in Alg. 1. In practice, as shown in Code 1, this is done by choosing the merge that occurs most frequently (can be adjusted for pair overlaps). The main loop occurs M times. In the subsequent theorem we show the approximation bound for the greedy algorithm.

Theorem 3.10. *The greedy algorithm for BPE training, i.e., for learning a length* M *merge sequence* $\boldsymbol{\mu}^\dagger$*, is* $\left(\frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1 - e^{-\sigma(\boldsymbol{\mu}^\star)})\right)$*-optimal: for every string* $\boldsymbol{x} \in \Sigma^*$

$$\frac{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^\dagger)}{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^\star)} \geq \frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1 - e^{-\sigma(\boldsymbol{\mu}^\star)}) \tag{10}$$

Sequence	Pair frequencies
Greedy	
[a,b]a[a,b]baa	ab : 2, ba: 2, aa: 2, bb: 1
[[a,b],a][a,b]baa	[a,b]a : 1, [a,b]b: 1, ba: 1, aa:1, [a,[a,b]]: 1
Optimal	
a[b,a]ab[b,a]a	ab: 2, ba : 2, aa: 2, bb: 1
a[[b,a],a]b[[b,a],a]	ab: 2, a[b,a]: 1, [b,a]a : 2, b[b,a]: 1

Example 2: In case of $\boldsymbol{x} = abaabbaa$ the greedy BPE yields a suboptimal compression utility (5 vs 4 subwords). Highlighted pairs show which one was chosen.

with respect to the optimal length M merge sequence $\boldsymbol{\mu}^\star$.

Proof. The proof is shown in App. A. ■

3.3 Measuring Total Backward Curvature

We do not have a formal bound for $\sigma(\boldsymbol{\mu}^\star)$ and estimate it by enumerating all strings of maximum length $|\boldsymbol{x}| \leq 15$ given a finite alphabet $|\Sigma| = 5$ and maximum merge sequence size $|\boldsymbol{\mu}^\star| < 5$. The found maximum is $\hat{\sigma}(\boldsymbol{\mu}^\star) = 2.5$, from which follows an optimality bound of ≈ 0.37. When we restrict our search to texts from a natural language (English), we obtain a slightly lower estimate $\hat{\sigma}(\boldsymbol{\mu}^\star)_N = 2.0$ and hence optimality bound ≈ 0.43. We leave the further study of the backward curvature constant to future work.

Notice that in the main proof of Theorem 3.10 in App. A, we used σ to bound only one particular type of sequence that becomes the prefix to $\boldsymbol{\mu}^\star$, namely $\boldsymbol{\mu}^\dagger$. We may then check for prefixing only greedy sequences instead of taking the maximum across $\boldsymbol{\mu} \in \Upsilon^*_\Sigma, |\boldsymbol{\mu}| \leq M$ as in Definition 3.9:

$$\sigma'(\boldsymbol{\mu}^\star, \boldsymbol{\mu}^\dagger) = \left\{ 1 - \frac{\kappa(\boldsymbol{\mu}^\dagger_{<M}\boldsymbol{\mu}^\star) - \kappa(\boldsymbol{\mu}^\star)}{\kappa(\boldsymbol{\mu}^\dagger_{<M})} \right\} \quad (11)$$

This yields $\hat{\sigma}'(\boldsymbol{\mu}^\star, \boldsymbol{\mu}^\dagger) = 1.5$ and therefore the bound of ≈ 0.52. More important than the particular bound value is that it is constant and that the BPE training algorithm can not be arbitratily suboptimal with sequence length.

4 A Runtime Speed-up

We now introduce a speed-up of the greedy BPE algorithm. Assuming constant-time comparison of strings, finding the maximum pair count over the whole string is $\mathcal{O}(N)$, which is the same as applying one merge. Therefore, this implementation has a runtime complexity of $\mathcal{O}(NM)$. A large amount of time in the slow BPE implementation,

Algorithm 2 Iterative Greedy BPE (faster).
Inputs: string $\boldsymbol{x}$, merge count M
Output: tokenized string $\boldsymbol{x}$, merge sequence $\boldsymbol{\mu}$

1: $\boldsymbol{\mu} \leftarrow \langle\rangle$
2: $\boldsymbol{x} \leftarrow$ LINKEDLIST($\boldsymbol{x}$)
3: $h \leftarrow$ MAXHEAP(PAIRS($\boldsymbol{x}$))
4: **for** i in $0..M$ **do**
5: $\quad pos \leftarrow h$.TOP
6: $\quad$ **for** (w_1, w_2) in pos **do**
7: $\quad\quad h$.REMOVEPOSITION($w_1.prev, w_1$)
8: $\quad\quad h$.REMOVEPOSITION($w_2, w_2.next$)
9: $\quad\quad w_1.val \leftarrow w_1.val + w_2.val$
10: $\quad\quad w_1.next \leftarrow w_2.next$
11: $\quad\quad w_2.next.prev \leftarrow w_1$
12: $\quad\quad h$.ADDPOSITION($w_1.prev, w_1$)
13: $\quad\quad h$.ADDPOSITION($w_1, w_1.next$)
14: $\quad$ **end for**
15: $\quad \boldsymbol{\mu} \leftarrow \boldsymbol{\mu} \circ \langle\mu\rangle$
16: **end for**
17: **return** $\boldsymbol{x}, \boldsymbol{\mu}$

presented by Sennrich et al. (2016) and shown in Alg. 1, is spent on (1) recalculating the frequencies of pairs (Alg. 1, line 3) which are not affected by the most recent merge, and (2) scanning the whole string to apply a single merge (Alg. 1, line 4). To make this explicit, consider the following example.

Example 4.1. *Consider $\boldsymbol{x} = abba(cddc)^n$ and merge $[a, b]$ for $n \geq 1$. We can only apply the merge at the beginning of the string, which results in the forest $[a, b]ba(cddc)^n$. However, Alg. 2 still scans the entirety of the sequence to recalculate the pair frequencies of $[c, d]$, $[d, c]$ and $[c, c]$. This additional work is unnecessary.*

Our idea to speed up Alg. 1 stems from the insight that we do not have to iterate over the entire sequence, an $\mathcal{O}(N)$ operation, on each of the M iterations.[12] Indeed, on the tth iteration, we show that one only has to do work proportional to the number of new nodes that are added to the forest (Alg. 2, line 6). To achieve this, we introduce a more efficient data structure for BPE.[13] Our first step is to treat the string as a linked list of subwords, initialized as a linked list of characters, that we destructively modify at each iteration. With each possible merge, we store a list of pointers where the merge operation could happen. The max heap is then sorted by the size of the sets. Lines 6 to 14 in Alg. 2 show the necessary operations needed to be performed on the linked list. Notably REMOVE-

[12] N is the string length $|\boldsymbol{x}|$ and M the number of merges.
[13] Kudo and Richardson (2018) make a similar observation, however, we prove a tighter bound on the runtime.

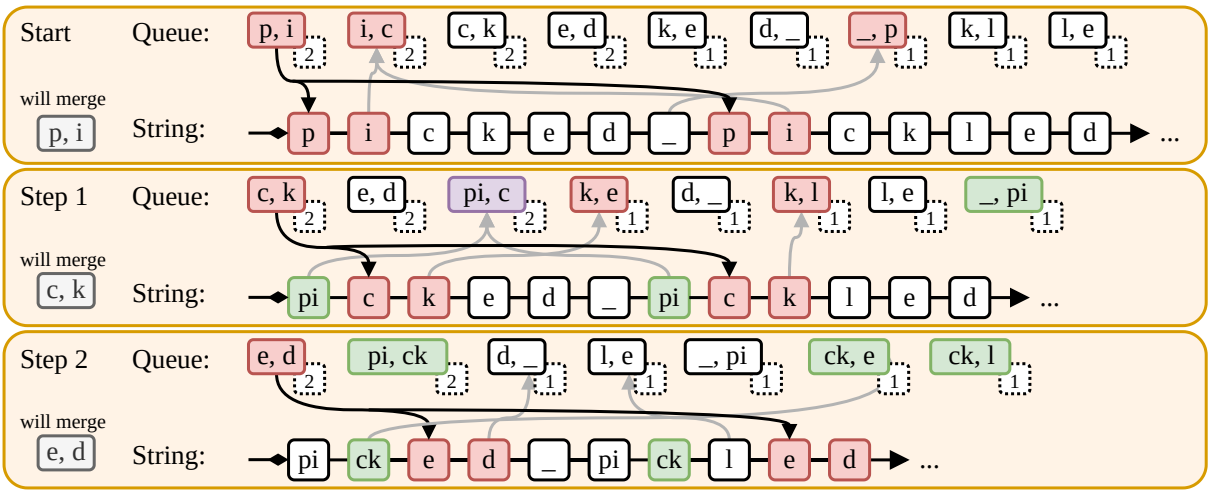

Figure 2: Visualization of linked list representation of the string and the associated priority queue (frequency values in dashed boxes) with merges. Nodes in red will be removed in the next step, nodes in green were added in contrast to the previous step and nodes in purple were just added but will be removed. Black lines from queue to the string show which nodes to merge. Grey lines show which pairs in the priority queue will have reduced frequencies.

POSITION removes the specific pair position from the set in the max heap and ADDPOSITION adds it.

See Fig. 2 for an illustration of applying a single merge in one place based on the introductory example in Example 1. The possible merge pairs are stored with a priority queue with their frequency as the sort key. During one operation, we need to remove the top merge pair and add counts for the newly created possible merge. The cost of one merge then becomes $\mathcal{O}(R_t \log M)$ where R_t is the number of pairs in the string where the merge occurs and $\log M$ the complexity of adding and updating frequency of a new merge pair. Note that it is not $\log N$, because we are keeping only top-M possible pairs in the heap.

At first glance, this suggests the overall runtime of $\mathcal{O}\left(\sum_{t=1}^{M} R_t \log M\right)$ with the worst case of the merge being applied along the whole string, therefore $\mathcal{O}(MN \log M)$.

Theorem 4.2. *Let N be the length of the string $\boldsymbol{x} \in \Sigma^*$ that is given as input. Then, Alg. 2 runs in $\mathcal{O}(N \log M)$ time.*

Proof. Let R_t be the amount of work performed at each iteration modifying the data structure. We additionally do $\mathcal{O}(\log M)$ work updating the priority queue on lines 6 to 14 in Alg. 2 since it has at most M elements. Thus, Alg. 2 clearly runs in $\mathcal{O}\left(\sum_{t=1}^{M} R_t \log M\right)$. We perform an amortized analysis. For this, we first make an observation about the upper bound on the number of merges and then show amortized analysis. However, for a string $\boldsymbol{x}$ of length N, there are at most $N-1$ merges that can be applied to $\boldsymbol{x}$. This implies that $\sum_{t=1}^{M} R_t \leq N$. Thus, $\mathcal{O}\left(\sum_{t=1}^{M} R_t \log M\right) = \mathcal{O}(N \log M)$, which proves the result. ■

5 An Exact Algorithm

In this section, we turn to developing an algorithm for exactly solving the BPE problem, i.e., Eq. (4). We change algorithmic paradigms and switch to memoization. While we are not able to devise a polynomial-time scheme, we are able to find an exact algorithm that is, in some cases, faster than the brute-force technique of enumerating all valid merge sequences. We first analyze the brute-force method of enumerating all valid merge sequences.

Proposition 5.1. *The set of valid merges of length M over a string $\boldsymbol{x} \in \Sigma^*$ is $\mathcal{O}\left(\min\left(|\Sigma|^{2M}, N^M\right)\right)$.*

Proof. The proof can be found in App. A. ■

A simple direct enumeration of all possible merge sequences with the time complexity of one merge $\mathcal{O}(NM)$ gives us a brute-force algorithm that runs in $\mathcal{O}\left(NM \min\left(|\Sigma|^{2M}, N^M\right)\right)$ time. The brute-force program explores all possible sequences of merges—including many that are redundant. For instance, both $\langle [p,o], [h,a] \rangle$ and $\langle [h,a], [p,o] \rangle$ induce the same partial bracketing when applied to another merge sequence, as in §2.2. Luckily, we are able to offer an exact characterization of when two merge sequences induce the same bracketing. To this end, we provide the following

definitions. We use the term **transposition** to refer to the swapping of items; i.e., a transposition (i, j) over a merge sequence $\boldsymbol{\mu}$ refers to the swapping of μ_i and μ_j.

Definition 5.2. *A pair of merges $\mu = [\mu_n, \mu_m]$ and $\mu' = [\mu_{n'}, \mu_{m'}]$ **conflicts** if for a symbol $a \in \Sigma$ and strings $\boldsymbol{x}, \boldsymbol{x}' \in \Sigma^*$, the yield of $[\mu_n, \mu_m]$ is $\boldsymbol{x}a$ and $[\mu'_n, \mu'_m]$ is $a\boldsymbol{x}'$.*

Definition 5.3. *A transposition (i, j) is **safe** if and only if, for all $k < j$, μ_k does not conflict with μ_j and, for all $k > i$, μ_k does not conflict with μ_i. A permutation $\pi = \langle \rho_1 \rho_2 \cdots \rho_n \rangle$, decomposed into transpositions, that maps one valid merge sequence $\boldsymbol{\mu}$ to another valid merge sequence $\pi(\boldsymbol{\mu}) = \boldsymbol{\mu}'$ is **safe** if and only if all transpositions are safe.*

Informally, Definition 5.3 says that for a permutation to produce a valid merge sequence, there should be no conflicts between the swapped merges and all merges in between. For example, given the merge sequence $\boldsymbol{\mu} = \langle [a, b], [d, d], [c, a] \rangle$, the permutation $\pi = \langle (1, 3) \rangle$ would not be safe.

The reason for this definition is that safe permutations characterize when two merge sequences always give the same results. Indeed, for $\boldsymbol{x} = ddabcacab$, applying the first merge sequence: $\textsc{Apply}_{\boldsymbol{\mu}}(\boldsymbol{x}) = [d, d][a, b][c, a][c, a]b$. In contrast, applying the permuted one gives an alternative outcome: $\textsc{Apply}_{\pi(\boldsymbol{\mu})}(\boldsymbol{x}) = [d, d][a, b][c, a]c[a, b]$.

Definition 5.4. *Two merge sequences $\boldsymbol{\mu}$ and $\boldsymbol{\mu}'$ are **equivalent** if and only if, for all $\boldsymbol{x} \in \Sigma^*$, $\textsc{Apply}_{\boldsymbol{\mu}}(\boldsymbol{x}) = \textsc{Apply}_{\boldsymbol{\mu}'}(\boldsymbol{x})$. Symbolically, we write $\boldsymbol{\mu} \equiv \boldsymbol{\mu}'$ if $\boldsymbol{\mu}$ and $\boldsymbol{\mu}'$ are equivalent.*

Proposition 5.5. *Two valid merge sequences $\boldsymbol{\mu}$, $\boldsymbol{\mu}' \in \mathcal{M}_{\Upsilon_\Sigma}$ are equivalent, i.e., $\boldsymbol{\mu} \equiv \boldsymbol{\mu}'$, if and only if there exists a safe permutation π such that $\pi(\boldsymbol{\mu}) = \boldsymbol{\mu}'$.*

Proof. The proof can be found in App. A. ■

Following the previous example, it is easy to verify that $\langle [a, b], [d, d], [c, a] \rangle \equiv \langle [a, b], [c, a], [d, d] \rangle$. In contrast to synthetic examples with a constrained alphabet of, e.g., $\{a, b, c\}$, far fewer merge conflicts arise in natural language. We can leverage this to develop a faster algorithm that only explores paths that are not equivalent to each other. We first define the concept of partial ordering between merges.

Definition 5.6. *The **merge partial ordering** $\mu' \gtrdot \mu''$ is defined as $\neg$conflicts$(\mu', \mu'') \wedge \neg(|\textsc{Yield}(\mu')| < |\textsc{Yield}(\mu'')|) \wedge \neg(\textsc{Yield}(\mu') <_L \textsc{Yield}(\mu''))$ where $>_L$ is lexicographical ordering.*

All valid merge sequences are equivalent to some merge sequence which is partially ordered using $\gtrdot$ so that no neighbouring elements violate this partial ordering. The brute-force algorithm works as depth-first search through an acyclic graph: each state corresponds to a unique sequence of merges and each transition corresponds to appending a merge to the end of the current state's merges. For the improved version, we make sure that only sequences which are ordered using $\gtrdot$ are searched and the rest are pruned. The pseudocode for the program is shown in Alg. 3. Even though the runtime is still prohibitively slow for application, Fig. 3 demonstrates how much speed is gained over the brute-force version which explores all states.

Algorithm 3 Exact BPE with memoization guard. Removing segments marked with ▒ would result in the brute-force version.
Inputs: string $\boldsymbol{x}$, merge count M
Output: tokenized string $\boldsymbol{x}$, merge sequence $\boldsymbol{\mu}$

```
q ← STACK( )
q.PUSH(⟨⟩, x)
μ*, x* ← ⟨⟩, x
while |q| ≠ 0 do
    μ, x ← q.POP()
    if |μ| = M then continue
    for μ ∈ PAIRS(x) do
        if |μ| = 0 ∨ μ ≫ μ_{-1}
            x' ← SINGLEAPPLY(x, μ)
            μ' ← μ ∘ μ
            if |x'| < |x*|
                μ*, x* ← μ', x'
            end if
            q.PUSH(μ', x')
        end if
    end for
end while
return μ*, x*
```

6 Conclusion

In this paper, we developed the formalisms surrounding the training task of BPE, a very popular tokenization algorithm in NLP. This allowed us to prove a lower bound on the compression utility by greedy BPE as $1 - e^{-\sigma(\boldsymbol{\mu}^\star)}$. We further analyzed the runtime of the naïve and faster greedy BPE algorithms and provided a speedup for finding an optimal BPE merge sequence. Future works should focus on providing either formal guarantees for $\sigma(\boldsymbol{\mu}^\star)$ or studying $\sigma(\boldsymbol{\mu}^\star)'$ across natural languages.

7 Limitations

Our work has focused strongly on the formal aspects of BPE. NLP practictioners should not be dissuaded from using BPE for subword tokenization, despite our presentation of examples where greedy BPE fails. Indeed, in contrast to synthetic examples on toy alphabet, on real data we made an observation that greedy BPE may be close to optimal.

Acknowledgements

We would like to thank Andreas Krause and Giorgio Satta for discussing the proof of Theorem 3.10. Clara Meister was supported by the Google PhD Fellowship. Juan Luis Gastaldi has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 839730.

References

Saeed Alaei, Ali Makhdoumi, and Azarakhsh Malekian. 2010. Maximizing sequence-submodular functions and its application to online advertising. *arXiv preprint arXiv:1009.4153*.

Jeff Bilmes. 2022. Submodularity in machine learning and artificial intelligence. *arXiv preprint arXiv:2202.00132*.

Kaj Bostrom and Greg Durrett. 2020. Byte pair encoding is suboptimal for language model pretraining. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4617–4624.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory*, 2 edition. Wiley-Interscience.

Shuoyang Ding, Adithya Renduchintala, and Kevin Duh. 2019. A call for prudent choice of subword merge operations in neural machine translation. In *Proceedings of Machine Translation Summit XVII: Research Track*, pages 204–213.

Ahmed El-Kishky, Vishrav Chaudhary, Francisco Guzmán, and Philipp Koehn. 2020. CCAligned: A massive collection of cross-lingual web-document pairs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5960–5969.

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Andreas Krause and Daniel Golovin. 2014. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press.

Taku Kudo 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.

László Lovász. 1983. Submodular functions and convexity. In *Mathematical programming the state of the art*, pages 235–257. Springer.

Azarakhsh Malekian. 2009. *Combinatorial Problems in Online Advertising*. Ph.D. thesis, University of Maryland, College Park.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Rico Sennrich, 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, Berlin, Germany.

Terry A. Welch. 1984. A technique for high-performance data compression. *Computer*, 17(06):8–19.

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and William B Dolan. 2020. DIALOGPT: Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278.

Zhenliang Zhang, Edwin KP Chong, Ali Pezeshki, and William Moran. 2015. String submodular functions with curvature constraints. *IEEE Transactions on Automatic Control*, 61(3):601–616.

Vilém Zouhar, Clara Meister, Juan Gastaldi, Li Du, Mrinmaya Sachan, and Ryan Cotterell. 2023. 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, Toronto, Canada. Association for Computational Linguistics.

A Proofs

Our proof of approximate optimality is based on the proof of greedily sequence maximizing submodular functions by Alaei et al. (2010); Zhang et al. (2015). However, we leverage a problem-specific property, which we dub hiearchical submodularity. We restate the definition here for ease.

Definition 3.7. *A real-valued function over valid merge sequences is* ***hierachically sequence submodular*** *if, for every valid merge sequence of the form $\boldsymbol{\mu}'\nu'\boldsymbol{\mu}\nu$ where $\nu' \subset \nu$ according to the partial order given in Definition 3.6, we have that*

$$f(\nu' \mid \boldsymbol{\mu}') \geq f(\nu \mid \boldsymbol{\mu}'\nu'\boldsymbol{\mu}). \tag{7}$$

Lemma A.1. *Let $\boldsymbol{\mu}', \boldsymbol{\mu} \in \mathcal{M}_{\Upsilon_\Sigma}$ be valid merge sequences. Then, there exists a merge ν in $\boldsymbol{\mu}$ such that $\boldsymbol{\mu}'\nu$ is a valid merge sequence and $\kappa_{\boldsymbol{x}}(\nu \mid \boldsymbol{\mu}') \geq \frac{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}|\boldsymbol{\mu}')}{|\boldsymbol{\mu}|}$. In words, the compression gain of some element in $\boldsymbol{\mu}$ with respect to $\boldsymbol{\mu}'$ is greater or equal to the average compression gain per element of $\boldsymbol{\mu}$ with respect to $\boldsymbol{\mu}'$*

Proof. Let us choose on of the possible maximums, $t = \operatorname{argmax}_{1 \leq t' \leq |\boldsymbol{\mu}|} \kappa_{\boldsymbol{x}}(\mu_{t'} \mid \boldsymbol{\mu}'\boldsymbol{\mu}_{<t'})$. Because we are taking the maximum, which is always equal to or greater than the average,[14] then $\kappa_{\boldsymbol{x}}(\mu_t \mid \boldsymbol{\mu}'\boldsymbol{\mu}_{<t}) \geq \frac{1}{|\boldsymbol{\mu}|}\sum_{t'=1}^{|\boldsymbol{\mu}|} \kappa_{\boldsymbol{x}}(\mu_{t'} \mid \boldsymbol{\mu}'\boldsymbol{\mu}_{<t'})$. Then, we have that either:

- $\boldsymbol{\mu}\mu_t \in \mathcal{M}_{\Upsilon_\Sigma}$, in which case the result follows by submodularity, or
- $\boldsymbol{\mu}\mu_t \notin \mathcal{M}_{\Upsilon_\Sigma}$, in which case there exists a $\mu_{t'}$ such that:
  - $\mu_{t'} \subset \mu_t$
  - $\boldsymbol{\mu}'\mu'_t \in \mathcal{M}_{\Upsilon_\Sigma}$
  - $\mu_{t'}$ in $\boldsymbol{\mu}$
  - $\kappa_{\boldsymbol{x}}(\mu_t \mid \boldsymbol{\mu}'\boldsymbol{\mu}_{<t}) \leq \kappa_{\boldsymbol{x}}(\mu_{t'} \mid \boldsymbol{\mu}'\boldsymbol{\mu}_{<t'}) \leq \kappa_{\boldsymbol{x}}(\mu_{t'} \mid \boldsymbol{\mu}')$

  In particular, all trivial submerges of μ_t (i.e., all submerges of μ_t whose constituents are in Σ) fulfill all four conditions: the first one by definition, the second by definiton of $\mathcal{M}_{\Upsilon_\Sigma}$, the third because $\boldsymbol{\mu} \in \mathcal{M}_{\Upsilon_\Sigma}$, and the fourth by hierarchical submodularity (first inequality) and by submodularity (second inequality).

■

We now proceed with the proof of approximate optimality of the greedy BPE merge sequence.

Theorem 3.10. *The greedy algorithm for BPE training, i.e., for learning a length M merge sequence $\boldsymbol{\mu}^\dagger$, is $\left(\frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1 - e^{-\sigma(\boldsymbol{\mu}^\star)})\right)$-optimal: for every string $\boldsymbol{x} \in \Sigma^*$*

$$\frac{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^\dagger)}{\kappa_{\boldsymbol{x}}(\boldsymbol{\mu}^\star)} \geq \frac{1}{\sigma(\boldsymbol{\mu}^\star)}(1 - e^{-\sigma(\boldsymbol{\mu}^\star)}) \tag{10}$$

with respect to the optimal length M merge sequence $\boldsymbol{\mu}^\star$.

Proof. We make use of the sequence $\boldsymbol{\mu}^\dagger_{<M}$ (rather than $\boldsymbol{\mu}^\dagger$) for reasons that will subsequently become clear. From Lemma A.1, we know that we can find $\mu_j^\star$ such that $\boldsymbol{\mu}^\dagger_{<M}\mu_j^\star$ is a valid merge sequence and

$$\kappa(\mu_j^\star \mid \boldsymbol{\mu}^\dagger_{<M}) \geq \frac{1}{M}\kappa(\boldsymbol{\mu}^\star \mid \boldsymbol{\mu}^\dagger_{<M}) \tag{12}$$

From the greedy property of $\boldsymbol{\mu}^\dagger$, we know:

$$\kappa(\mu_M^\dagger \mid \boldsymbol{\mu}^\dagger_{<M}) \geq \kappa(\mu_j^\star \mid \boldsymbol{\mu}^\dagger_{<M}) \tag{13}$$

$$\kappa(\mu_M^\dagger \mid \boldsymbol{\mu}^\dagger_{<M}) \geq \frac{1}{M}\kappa(\boldsymbol{\mu}^\star \mid \boldsymbol{\mu}^\dagger_{<M}) \qquad \text{(from Eq. 12)} \tag{14}$$

$$\kappa(\boldsymbol{\mu}^\dagger_{<M}\mu_M^\dagger) - \kappa(\boldsymbol{\mu}^\dagger_{<M}) \geq \frac{1}{M}(\kappa(\boldsymbol{\mu}^\dagger_{<M}\boldsymbol{\mu}^\star) - \kappa(\boldsymbol{\mu}^\dagger_{<M})) \qquad \text{(definition expansion)} \tag{15}$$

[14] Proof of this algebraic statement is omitted for brevity.

Now from backward curvature (Definition 3.9) and by substituting $\boldsymbol{\mu}^{\dagger}_{<M}$ for the prefix sequence:

$$\sigma(\boldsymbol{\mu}^{\star}) \geq 1 - \frac{\kappa(\boldsymbol{\mu}^{\dagger}_{<M}\boldsymbol{\mu}^{\star}) - \kappa(\boldsymbol{\mu}^{\star})}{\kappa(\boldsymbol{\mu}^{\dagger}_{<M})} \tag{16}$$

$$\sigma(\boldsymbol{\mu}^{\star})\kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \geq \kappa(\boldsymbol{\mu}^{\dagger}_{<M}) - \kappa(\boldsymbol{\mu}^{\dagger}_{<M}\boldsymbol{\mu}^{\star}) + \kappa(\boldsymbol{\mu}^{\star}) \tag{17}$$

$$\kappa(\boldsymbol{\mu}^{\dagger}_{<M}\boldsymbol{\mu}^{\star}) - \kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \geq \kappa(\boldsymbol{\mu}^{\star}) - \sigma(\boldsymbol{\mu}^{\star})\kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \tag{18}$$

Applying this result to the right-hand side of Eq. (15), we obtain the following:

$$\kappa(\boldsymbol{\mu}^{\dagger}_{<M}\mu^{\dagger}_{M}) - \kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \geq \frac{1}{M}(\kappa(\boldsymbol{\mu}^{\star}) - \sigma(\boldsymbol{\mu}^{\star})\kappa(\boldsymbol{\mu}^{\dagger}_{<M})) \qquad \text{(total backward curvature)} \tag{19}$$

$$\kappa(\boldsymbol{\mu}^{\dagger}) - \kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \geq \frac{1}{M}(\kappa(\boldsymbol{\mu}^{\star}) - \sigma(\boldsymbol{\mu}^{\star})\kappa(\boldsymbol{\mu}^{\dagger}_{<M})) \qquad \text{(definition)} \tag{20}$$

$$\kappa(\boldsymbol{\mu}^{\dagger}) \geq \frac{1}{M}(\kappa(\boldsymbol{\mu}^{\star}) - \sigma(\boldsymbol{\mu}^{\star})\kappa(\boldsymbol{\mu}^{\dagger}_{<M})) + \kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \qquad \text{(total backward curvature)} \tag{21}$$

$$\geq \frac{1}{M}\kappa(\boldsymbol{\mu}^{\star}) + \left(1 - \frac{\sigma(\boldsymbol{\mu}^{\star})}{M}\right)\kappa(\boldsymbol{\mu}^{\dagger}_{<M}) \qquad \text{(algebraic manipulation)} \tag{22}$$

$$\geq \frac{1}{M}\kappa(\boldsymbol{\mu}^{\star})\sum_{i=0}^{M-1}\left(1 - \frac{\sigma(\boldsymbol{\mu}^{\star})}{M}\right)^{i} \qquad \text{(recursive substitution of } \kappa(\boldsymbol{\mu}^{\dagger}_{<i})) \tag{23}$$

$$= \frac{1}{\sigma(\boldsymbol{\mu}^{\star})}\left(1 - \left(1 - \frac{\sigma(\boldsymbol{\mu}^{\star})}{M}\right)^{M}\right)\kappa(\boldsymbol{\mu}^{\star}) \qquad \text{(geometric sum)} \tag{24}$$

$$= \frac{1}{\sigma(\boldsymbol{\mu}^{\star})}\left(1 - \left(1 - \frac{\sigma(\boldsymbol{\mu}^{\star})}{M}\right)^{\frac{M}{\sigma(\boldsymbol{\mu}^{\star})}}\right)^{\sigma(\boldsymbol{\mu}^{\star})}\kappa(\boldsymbol{\mu}^{\star}) \qquad \text{(preparation)} \tag{25}$$

We substitute $x = \frac{M}{\sigma(\boldsymbol{\mu}^{\star})}$ in the inequality. From $x > 0 \Rightarrow \left(1 - \frac{1}{x}\right)^{x} \leq \frac{1}{e}$, we obtain and arrive at

$$\kappa(\mu^{\dagger}) \geq \frac{1}{\sigma(\boldsymbol{\mu}^{\star})}\left(1 - e^{-\sigma(\boldsymbol{\mu}^{\star})}\right) \tag{26}$$

■

Proposition 5.5. *Two valid merge sequences $\boldsymbol{\mu}$, $\boldsymbol{\mu}' \in \mathcal{M}_{\Upsilon_\Sigma}$ are equivalent, i.e., $\boldsymbol{\mu} \equiv \boldsymbol{\mu}'$, if and only if there exists a safe permutation π such that $\pi(\boldsymbol{\mu}) = \boldsymbol{\mu}'$.*

Proof. **(⇒):** We prove the first implication through contrapositive, i.e., we show that if there does *not* exist such a safe permutation π, then the merge sequences are *not* equivalent. By supposition, all non-safe permutations mapping $\boldsymbol{\mu}$ to $\boldsymbol{\mu}'$ either have a conflict or do not preserve validity. We handle each case separately.

- **Case 1:** Suppose that the permutation π re-orders two conflicting merges μ and μ'. By the definition of a conflict, μ has yield $\boldsymbol{x}a$ and μ' has yield $a\boldsymbol{x}'$ for $a \in \Sigma$ and $\boldsymbol{x}, \boldsymbol{x}' \in \Sigma^*$. Now, note the bracketing string $\boldsymbol{x}a\boldsymbol{x}'$ will be different under the original and permuted merge sequence.
- **Case 2:** Suppose that the permutation π does not preserve validity. Then, there exists a merge $\mu = (\mu', \mu'')$ such that either μ' or μ'' occurs *after* μ in the merge sequence. This also results in a different bracketing.

(⇐): Next, we want to show the converse, i.e., for any safe permutation π, we have $\boldsymbol{\mu} \equiv \pi(\boldsymbol{\mu})$. Let $\boldsymbol{\mu} = \langle \mu_1, \ldots, \mu_N \rangle$ be a merge sequence of length N, and let π be a safe permutation. We proceed by induction on the n.

- **Base Case:** Since π is safe, then for $[a, b] = \pi(\boldsymbol{\mu})_1$, a and b are necessarily characters in Σ.
- **Inductive Step**: Suppose for $k = n - 1$, $\pi(\boldsymbol{\mu})_{\leq k}$ applies merges which are applied by $\boldsymbol{\mu}$. We then show $\pi(\boldsymbol{\mu})_n$ also applies the same merges as $\boldsymbol{\mu}$. Consider $\pi(\boldsymbol{\mu})_n = (\mu_m, \mu_{m'})$; since π is safe, both μ_m and $\mu_{m'}$ already exist in $\textsc{Apply}_{\boldsymbol{\mu}_{\leq n}}(\boldsymbol{x})$. Moreover, since there are no conflicts, applying $\pi(\mu)_n$ results in the same encoded sequence.

■

Proposition 5.1. *The set of valid merges of length M over a string $\boldsymbol{x} \in \Sigma^*$ is $\mathcal{O}\left(\min\left(|\Sigma|^{2M}, N^M\right)\right)$.*

Proof. On one hand, we note that we have an upper bound of $N - 1$ possible merges that can occupy the first element of the sequence, assuming every symbol in $\boldsymbol{x}$ is distinct. Next, we have $N - 2$ possible merges that can occupy the second element of the sequence, again, assuming every symbol in $\boldsymbol{x}$ is distinct. Continuing this pattern, we arrive at a simple upper bound on the number of merges $\prod_{m=0}^{M-1}(N - 1 - m)$. This quantity is recognizable as a falling factorial, which gives us the closed form $\frac{(N-1)!}{(N-M-2)!}$. This can be trivially bounded by N^M. However, on the other hand, we know a valid merge sequence can produce merges with a yield up to length M, and there are $\binom{\Sigma^{\leq M}}{M}$ unique sequences. We can upper-bound the number of valid merge sequences by the total number of all possible merge sequences, of which there are $M!$. The size of $\Sigma^{\leq M}$ is the sum $|\Sigma|^1 + |\Sigma|^2 + \ldots |\Sigma|^M$ which is less than $M|\Sigma|^M$. Again, with $M!$, this leads to the falling factorial $\frac{(M|\Sigma|^M)!}{(M|\Sigma|^M - M)!}$ which we can upper bound by $(M|\Sigma|^M)^M$ which is in $\mathcal{O}\left(|\Sigma|^{2M}\right)$. Taking the min of these two upper bounds gives us the overall upper bound. ■

B BPE Modifications

In this section, we describe multiple modifications to the greedy BPE algorithm which speed up the runtime. We do not address popular heuristic modifications such as lowercasing the text or adding 20% of the most frequent words to the subword dictionary.

B.1 (Not) Merging Space

Currently, spaces are treated as any other characters and are allowed to be part of merges. Therefore in the string *"not_that_they_watch_the_watch"* the first merge is *[_,t]* and the string looks as *"not[_,t]hat[_,t]hey watch[_,t]he watch"*. The next merge may be across tokens: *[t,[_,t]]*. This is not desirable if we want only want to split tokens into subwords (i.e. use merges that do not contain spaces).

Furthermore, in §3 we are duplicating work by computing pair frequencies and merges multiple times across the same tokens that occur multiple times (see previous string example). In practice (Tab. 1), only 1.5% of all tokens are unique. We may then speed up our computation by considering only unique tokens. Therefore, the new runtime complexity is $\mathcal{O}(V \cdot |\boldsymbol{x}_u|)$ where $\boldsymbol{x}_u = \{t \mid \text{token } t \in \boldsymbol{x}\}$ which is $\frac{|\boldsymbol{x}|}{|\boldsymbol{x}_u|}\times$ faster.

B.2 Non-iterative BPE

A popular implementation of BPE-like algorithm in Python[15] uses a different speed-up mechanism to avoid $\mathcal{O}(NV)$ runtime. This is done by:

(1) collecting all possible merges observed in the data up until some maximum yield size which determines the maximum subword size, such as 5 and
(2) taking top-M frequent pairs as part of the subword dictionary.

Note that because of hiearchical submodularity (Definition 3.7), this will produce valid merges. This is because if $\mu = [\mu', \mu'']$ is chosen, so must μ' and μ'' because they have at least the same frequency as μ. For example, for $abcabcd$, and maximum yield width 3, the merges would be $[a, b], [[a, b], c], [b, c], [a, [b, c]], \ldots$. The runtime of this is $\mathcal{O}(|\boldsymbol{x}| \log M)$ because we are scanning the whole string and at each point are modifying maximum heap.

[15] pypi.org/project/bpe

However, it is easy to see that this approximation algorithm is not bounded. For a constant maximum yield width of w, consider $\boldsymbol{x} = a^{wn}$ and $V = w + k$. The shortest possible output of this algorithm will be μ^n. However, an optimal merge sequence can perform additional merge sequences, therefore producing $\boldsymbol{\nu}^{\frac{n}{2^k}}$. The compressions are $wn - n$ and $wn - \frac{n}{2^k}$ and the ratio $\frac{wn-n}{wn-\frac{n}{2^k}}$ with lower bound of 0 as supremum. This means that we can construct adversarial example for which the compression given by this algorithm is arbitrarily suboptimal.

C Additional Experimental Details and Results

Sentence count (train)	13M+13M
Sentence count (dev & test)	1M+1M
Total words	324M
Unique words	5M
Average sentence length (words)	12

Table 1: Overview of the used portion of the English-German CommonCrawl dataset (El-Kishky et al., 2020).

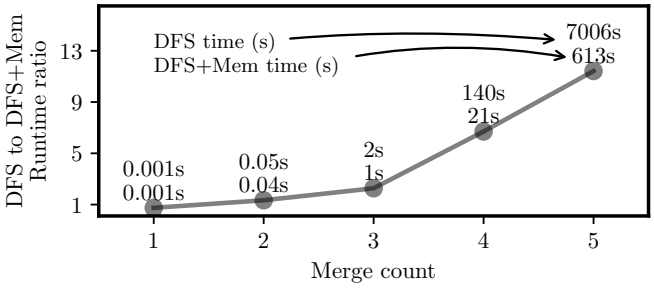

Figure 3: Comparison of runtimes for brute-force DFS and DFS with memoization. Values above 1 correspond to DFS+memoization being $\times$ faster than DFS. Points show average[16]of runs on 5 different input strings (each 2 randomly sampled English sentences of 64 characters).

[16]Time measured on desktop AMD Ryzen 9 5900X.

```python
from collections import Counter, defaultdict
from typing import Union, Tuple, List

def fixed_pair_freqs(xs: Union[str, List]):
    pairs = defaultdict(int)
    prev_pair = None
    for (x, y) in zip(xs, xs[1:]):
        # increment only if the prev suffix does not match prefix
        # otherwise wrong estimate on `aaa`
        if (x,y) != prev_pair:
            pairs[x, y] += 1
            prev_pair = (x, y)
        else:
            # make sure to clear it so that `aaaa` is counted twice
            prev_pair = None

    pairs = list(pairs.items())
    pairs.sort(key=lambda x: x[1], reverse=True)
    return pairs

def bpe(xs: Union[str, List], V: int):
  for _ in range(V):
    top_pair = fixed_pair_freqs(xs)[0]
    xs = merge(list(xs), top_pair)
  return xs

def merge(xs: List, pair: Tuple):
  ys = []
  while xs:
    if tuple(xs[:2]) == pair:
      ys.append(pair)
      xs = xs[2:]
    else:
      ys.append(xs.pop(0))
  return ys
```

Code 2: An implementation of Sennrich et al.'s (2016) greedy algorithm for BPE in Python with overlap-adjusted pair counts.