# Fast, High-Fidelity LLM Decoding with Regex Constraints: Technical Appendix

Vivien Tran-Thien

February 2024

**Abstract**

This document formally describes and proves the algorithms mentioned in the blog post entitled *Fast, High-Fidelity LLM Decoding with Regex Constraints* [2].

## 1 Notations and Introduction

### 1.1 Tokenizers Based on a Merge Table

Let $\Sigma$ be an alphabet. In this document we focus on the tokenizers, such as byte-pair encoding tokenizers [1], that are based on a merge table. We formally define such a tokenizer $\mathcal{T}$ as a couple $(T, \mathcal{M})$:

- $T$ is a finite set of strings over $\Sigma$ (i.e. a subset of $\Sigma^*$) which includes all single-letter strings of $\Sigma$. The elements of $T$ are called *tokens*;
- $\mathcal{M} = ((a_1, b_1), ..., (a_n, b_n))$ is a *merge table*, i.e. a sequence of pairs of tokens so that, for each pair, the concatenation of the tokens is also a token.

In this context, we define the $\texttt{decode}_{\mathcal{T}}$, $\texttt{tokenize}_{\mathcal{T}}$ and $\texttt{merge}_{a,b}$ functions:

- $\texttt{decode}_{\mathcal{T}} : T^* \mapsto \Sigma^*$ simply concatenates the input tokens;
- $\texttt{tokenize}_{\mathcal{T}} : \Sigma^* \mapsto T^* = \texttt{merge}_{a_n,b_n} \circ \cdots \circ \texttt{merge}_{a_1,b_1}$
- $\texttt{merge}_{a,b} : T^* \mapsto T^*$ is recursively defined:

$$\texttt{merge}_{a,b}(\epsilon) = \epsilon \text{ where } (\epsilon \text{ is the empty string})$$

$$\texttt{merge}_{a,b}(c_1) = c_1 \text{ (where } c_1 \in \Sigma)$$

$$\texttt{merge}_{a,b}(c_1 c_2 \dots c_n) = \begin{cases} c_1 \, \texttt{merge}_{a,b}(c_2 \dots c_n) & \text{if } c_1 \neq a \text{ or } c_2 \neq b \\ (ab) \, \texttt{merge}_{a,b}(c_3 \dots c_n) & \text{if } c_1 = a \text{ and } c_2 = b \end{cases}$$

... where $(ab)$ denotes the token resulting from the concatenation of $a$ and $b$. For the sake of simplicity, $\mathcal{T}$ is later omitted when referring to `decode` and `tokenize`

### 1.2 Deterministic Finite Automata

Let $\mathcal{A}$ be a deterministic finite automaton (DFA) recognizing a language $\mathcal{L} \subset \Sigma^*$. We use the following notations:

- $\mathcal{S}$ is the set of states;
- $\delta \subset \mathcal{S} \times \Sigma \times \mathcal{S}$ is the partial transition function;
- $\hat{\delta} \subset \mathcal{S} \times \Sigma^* \times \mathcal{S}$ is the partial extended transition function;
- $S_0 \in \mathcal{S}$ is the start state;
- $F \subset \mathcal{S}$ is the set of accept states.

We assume that each state of $\mathcal{A}$ is *relevant*, i.e. it is part of a path from the start state to an accept state.

### 1.3 Objective and Outline of the Document

Our objective in this document is to build a DFA $\mathcal{A}_\mathcal{T}$ that recognizes $\texttt{tokenize}(\mathcal{L}) \subset T^*$ and only includes relevant states.

We can assume that the merge table only contains a single merge operation $(a, b)$ because if we know how to address this case, we can recursively handle a longer merge table. The rest of the document is structured as follows:

- Starting with the case $a \neq b$, we introduce `DirectMerge`, an algorithm to build $\mathcal{A}_\mathcal{T}$;
- We then extend `DirectMerge` to the case $a = b$;
- We present `CartesianMerge`, a more scalable variant of `DirectMerge`;
- We finally discuss some minor implementation details for `DirectMerge` and `CartesianMerge`.

## 2 `DirectMerge`: Case of $\texttt{merge}_{a,b}$ with $a \neq b$

We now assume $a \neq b$. Let us consider the string $c_0 \dots c_{i-1} c_i c_{i+1} c_{i+2} \dots c_m$ with $c_i = a$ and $c_{i+1} = b$. Since $a \neq b$, we know that $c_i \neq b$ and $c_{i+1} \neq a$. As a result, $c_{i-1}$ and $c_i$ on one hand and $c_{i+1}$ and $c_{i+2}$ on the other hand cannot be merged by $\texttt{merge}_{a,b}$ while $c_i$ and $c_{i+1}$ are necessarily merged.

Otherwise said, when $a \neq b$, a token $a$ followed by a token $b$ are merged, regardless of the preceding or following tokens. This means that we can locally examine and modify the DFA without taking into account more distant states. Let us see this more concretely with some simple examples, as a stepping stone to the more general case.

## 2.1 Simple Examples

**Example 2.1.** A state $S$ has no incoming $a$ transition or no outgoing $b$ transition (Figure 1a). In this case, there is no reason to modify this state. Please note that the figure and the following ones show the states as distinct but the reasoning still applies if some states among $S_x$, $S$ and $S_y$ are actually the same (for example, if there is a loop).

**Example 2.2.** $S$ has one incoming $a$ transition from $S_a$ and one outgoing $b$ transition to $S_b$ (Figure 1c). In this case, $\texttt{merge}_{a,b}$ will necessarily merge token $a$ and token $b$. This means that we need to add an $ab$ transition from $S_a$ to $S_b$. Moreover, $S$ has become irrelevant: it is neither a start state nor an accept state and its only incoming transition and its only outgoing transition cannot both be used. Therefore we need to remove $S$ and this would not modify the recognized language.

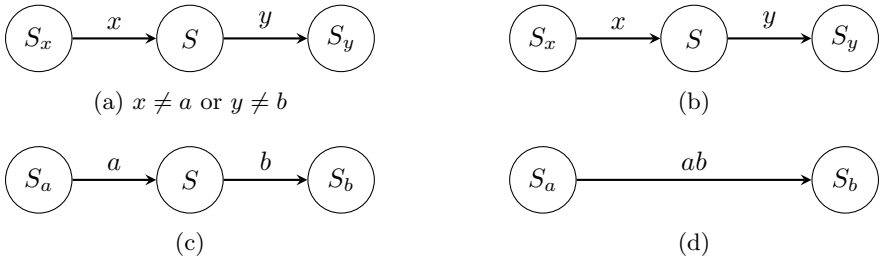



Figure 1: Simple examples with $\texttt{merge}_{a,b}$ and $a \neq b$. We assume that $S$ has no incoming or outgoing transition other than those represented here. In contrast, $S_a$, $S_b$ and $S_x$ can have other incoming or outgoing transitions and can be start or accept states. The left figures depict the initial DFA while the right figures correspond to the transformations made necessary by $\texttt{merge}_{a,b}$

**Example 2.3.** Same as Example 2.2 but with an added outgoing $y$ transition ($y \neq b$) from $S$ to a state $S_y$ (Figure 2a). In this case, the $ab$ transition from $S_a$ to $S_b$ remains necessary but $S$ cannot be discarded because it is on the path from $S_a$ to $S_y$. However, we need to remove the outgoing $b$ transition to prevent a token $a$ from being followed by a token $b$.

**Example 2.4.** Same as Example 2.2 but with $S$ assumed to be an accept state (Figure 2c). In this case, we need to keep $S$ which is now relevant because it is an accept state and remove its outgoing $b$ transition. This example can be seen as a particular case of the previous one because being an accept state is equivalent to having an outgoing $\epsilon$-transition to an accept state. Similarly, being the start state is equivalent to having an incoming $\epsilon$-transition from the start state. This means that when considering the various potential configurations, we can always restrict ourselves to examining the incoming and outgoing transitions, without handling separately the specific cases of start or accept states.

**Example 2.5.** Same as Example 2.2 but with an added incoming $x$ transition ($x \neq a$) from $S_x$ to $S$ (Figure 2e). In this case, $S$ is relevant because it is on the path from $S_x$ to $S_b$: we need to keep it while removing the incoming $a$ transition. As discussed in the previous example, it would have been the same if $S$ had been a start state.

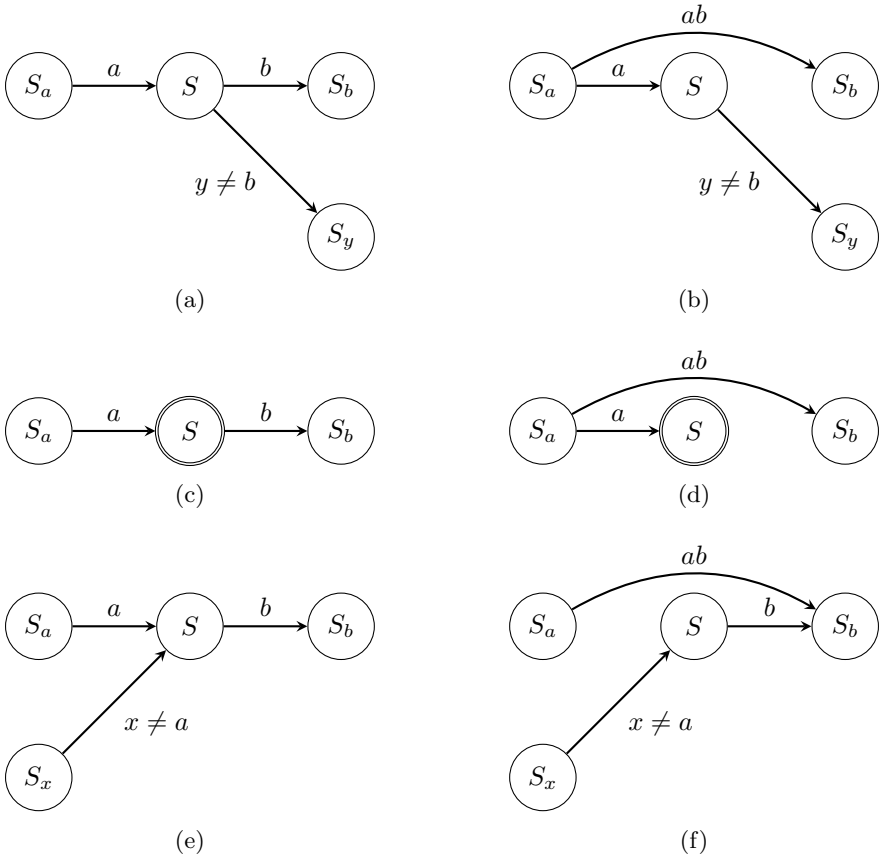



Figure 2: Some additional examples with $\texttt{merge}_{a,b}$ and $a \neq b$.

## 2.2 General Case with $a \neq b$

We now introduce Algorithm 1 and prove its correctness, i.e. that it yields a modified automaton which recognizes $\texttt{merge}_{a,b}(\mathcal{L})$ and only includes relevant states.

**Algorithm 1:** `DirectMerge` for $\texttt{merge}_{a,b}$ with $a \neq b$

**Input:**
- DFA $(\Sigma, \mathcal{S}, \delta, F, S_0)$ recognizing a language $\mathcal{L}$
- Tokens $a$ and $b$ with $a \neq b$

**Output:** DFA recognizing the language $\texttt{merge}_{a,b}(\mathcal{L})$

```
1  function DirectMerge((Σ, 𝒮, δ, F, S_0), a, b)
2      Add ab to Σ
3      for S ∈ 𝒮 with incoming a transitions and an outgoing b transition
        do
4          (Σ, 𝒮, δ, F, S_0) = ProcessState(S, a, b)
5      return (Σ, 𝒮, δ, F, S_0)
6  function ProcessState(S, a, b)
7      S_b := δ(S, b)
8      for S_a ∈ 𝒮 with S = δ(S_a, a) do
9          Add transition (S_a, ab, S_b)
10     if S = S_0 or S has incoming x transitions (x ≠ a) then
11         if S ∈ F or S has outgoing y transitions (y ≠ b) then
12             Add S' to 𝒮
13             for S_a ∈ 𝒮 with S = δ(S_a, a) do
14                 Remove transition (S_a, a, S)
15                 Add transition (S_a, a, S')
16             for S_y ∈ 𝒮 with S_y = δ(S, y) and y ≠ b do
17                 Add transition (S', y, S_y)
18             if S ∈ F then
19                 Add S' to F
20         else
21             for S_a ∈ 𝒮 with S = δ(S_a, a) do
22                 Remove transition (S_a, a, S)
23     else
24         if S ∈ F or S has outgoing y transitions with y ≠ b then
25             Remove transition (S, b, S_b)
26         else
27             Remove S
28     return (Σ, 𝒮, δ, F, S_0)
```

*Proof.* As discussed with Example 2.1, an $(a, b)$ merge cannot happen at the level of a state without an incoming $a$ transition or an outgoing $b$ transition so we should not modify such states.

We now cover the general case of a state $S$ with at least one incoming $a$ transition and one outgoing $b$ transition. As depicted on Figure 3, $S$ can have

several other incoming or outgoing transitions. There can be more than one incoming $a$ transition but there cannot be other outgoing $b$ transitions because $\mathcal{A}$ is a DFA. Some of these transitions can be loops. Finally, $S$ can be the start state or an accept state.

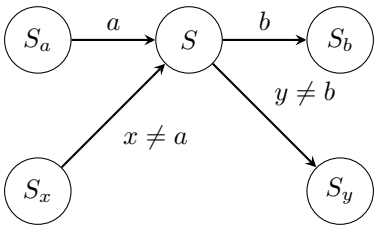
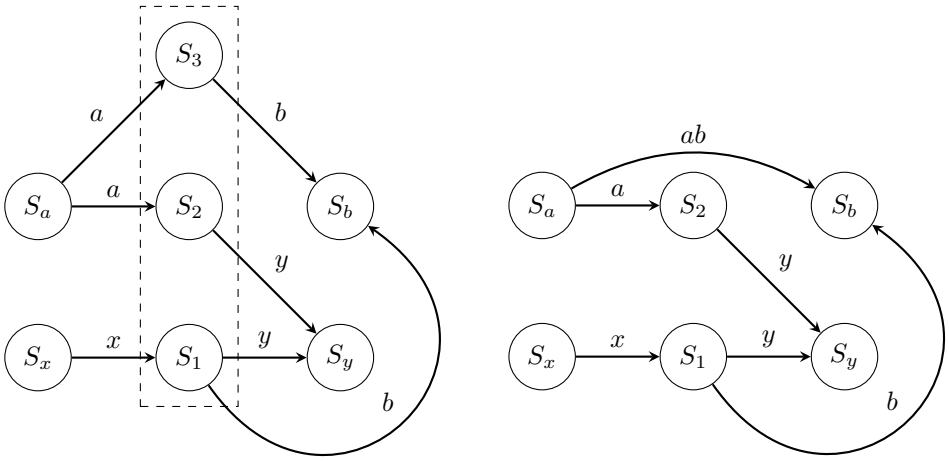



Figure 3: General case with $\texttt{merge}_{a,b}$ and $a \neq b$. There might be several states with transitions leading to $S$ or coming from $S$. Only one of each is represented here. Moreover, loops on $S$ are possible, e.g. $y = x, S_x = S_y = S$ or $y = a, S_a = S_y = S$.

Such a general case is more complex than the simple examples examined before but we can actually transform the DFA to return to these basic configurations. To do so, we remove $S$ and replace it with three states that collectively perform the same function. More precisely and as illustrated on Figure 4a, we introduce:

- $S_1$ that processes all $x$ incoming transitions with $x \neq a$ and all outgoing transitions. $S_1$ is an accept (respectively start) state if $S$ is an accept (respectively start) state;
- $S_2$ that processes all incoming $a$ transitions and all outgoing $y$ transitions with $y \neq b$. $S_2$ is an accept state if $S$ is an accept state;
- $S_3$ that processes all incoming $a$ transitions and the outgoing $b$ transition.




(a) $S$ is decomposed in 3 states that collectively perform the same function.

(b) $S_1$ and $S_2$ are unaffected by $\texttt{merge}_{a,b}$ while $S_3$ is removed.

Figure 4: `DirectMerge` applied to a state without self-loops.

It is easy to see that the recognized language of the automaton remains unaffected by this transformation. However, the automaton has become a non-deterministic finite automaton (NFA) because each state $S_a$ with an $a$ transition to $S$ now has one $a$ transition to $S_2$ and one to $S_3$. As we see below, this is only temporary.

$S_1$ and $S_2$ fall into the case of Example 2.1: $S_1$ has no incoming $a$ transition while $S_2$ has no outgoing $b$ transition. As a consequence, they should remain unaffected by $\texttt{merge}_{a,b}$. In contrast, $S_3$ is exactly in the situation of Example 2.2: it should be removed and an $ab$ transition should be added from all states $S_a$ to state $S_b$, as mentioned on lines 4-6 of Algorithm 1. We are then in the configuration of Figure 4b and the automaton is deterministic again.

We now need to determine whether $S_1$ and $S_2$ are relevant. It depends on which transitions exist in addition to the incoming $a$ transitions and the outgoing $b$ transition. More precisely, we consider the following questions:

1. Is $S$ the start state or is there a non-$a$ incoming transition?
2. Is $S$ an accept state or is there a non-$b$ outgoing transition?

The yes/no answers to these questions lead to four situations:

- **1. Yes, 2. Yes**. In this case, $S_1$ and $S_2$ are both relevant and, by the way $S_1$ and $S_2$ were defined, the changes compared to the initial situation are exactly those described on lines 12-19 of Algorithm 1 with $S = S_1$ and $S' = S_2$: all incoming $a$ transitions are redirected to $S'$ instead of $S$ and all non-$b$ outgoing transitions from $S$ are added to $S'$ (this potentially includes an $\epsilon$-transition to an accept state so $S'$ becomes an accept state if $S$ is one).

- **1. Yes, 2. No**. $S_2$ has no outgoing transition and is not an accept state so $S_2$ is not relevant. We then just keep $S_1$ that we rename as $S$. Compared to the initial situation, the only change for $S$ is that the incoming $a$ transitions have been removed, as indicated on lines 21-22 of Algorithm 1.

- **1. No, 2. Yes**. Without an incoming transition and without being the start state, $S_1$ is unreachable. We then need to remove it and we can rename $S_2$ as $S$. Compared to the initial situation, the only change for $S$ is that the outgoing $b$ transitions has been removed. This corresponds to line 25 of Algorithm 1.

- **1. No, 2. No**. Neither $S_1$ nor $S_2$ are relevant and they should be both removed. Compared to the initial situation, we removed $S$ without replacing it with any new state, as foreseen on line 27 of Algorithm 1.

The reasoning above also applies if the DFA includes self-loops are present. The only slight difference is that if there is an $a$ transition from $S$ to itself, we need to be careful when "rewiring" the transitions between $S_1$, $S_2$ and $S_3$: by definition of $S_1$, $S_2$ and $S_3$, the $a$ self-loop must be converted into $a$ transitions from $S_1$ to $S_2$ and $S_3$ (because all incoming $a$ transitions are redirected to $S_2$ and $S_3$) and an $a$ self-loop on $S_2$ (because all outgoing transitions from $S$ other than $b$ are reproduced by $S_2$), as illustrated by Figure 5a. We can then check that Algorithm 1 applies in the same way when self-loops are present.

To sum up, after the steps described in Algorithm 1:

- The only changes affecting the language recognized by the DFA are consecutive $a$ and $b$ transitions that were converted into an $ab$ transition;

- The remaining states are all relevant;

- The resulting automaton is deterministic.

□

# 3 `DirectMerge`: Case of $\texttt{merge}_{a,a}$

## 3.1 Why Parity Matters

In the case of $\texttt{merge}_{a,a}$, two consecutive $a$ may or may not be merged depending on the preceding characters. More precisely, given the definition of `merge`, these two consecutive $a$ are merged only if they are immediately preceded by an even (potentially zero) number of consecutive $a$. For this reason, we define the parity of a word ($\texttt{parity}_a : \Sigma^* \mapsto \{0, 1\}$) as the parity of the number of trailing $a$ in this word:

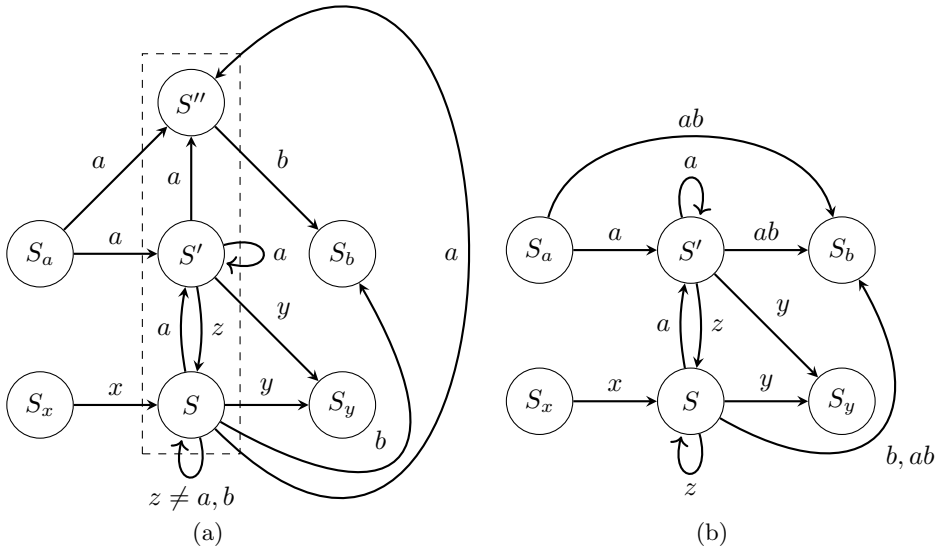



Figure 5: `DirectMerge` applied to a state with self-loops.

$$\texttt{parity}_a(\epsilon) = 0$$

$$\texttt{parity}_a(c_1 c_2 \dots c_n) = \begin{cases} 0 & \text{if } c_n \neq a \\ 1 - \texttt{parity}_a(c_1 c_2 \dots c_{n-1}) & \text{if } c_n = a \end{cases}$$

We also define the parity of a state ($\texttt{parity}_a : \mathcal{S} \mapsto \mathcal{P}(\{0,1\})$) as the set of values taken by the parity of the words recognized by this state:

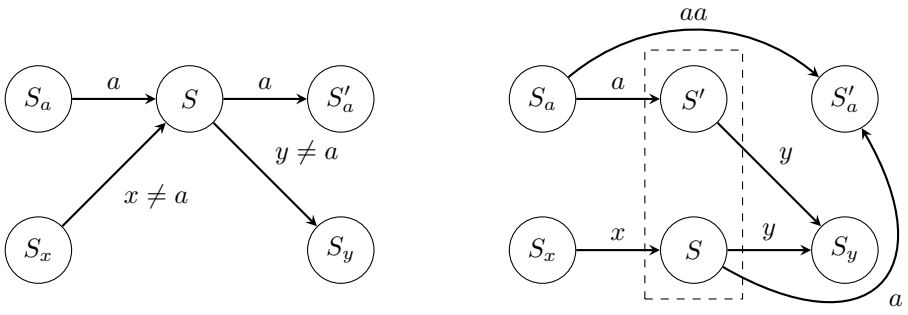
$$\texttt{parity}_a(S) = \texttt{parity}_a(\{w, \hat{\delta}(S_0, w) = S\})$$

Since all states are assumed to be relevant, the parity of a state may be $\{0\}$, $\{1\}$ or $\{0,1\}$ but not $\emptyset$.

## 3.2 Two Simple Examples

We now consider two simple examples to illustrate how parity affects the required transformations.

**Example 3.1.** $S$ has an incoming $a$ transition from $S_a$, an incoming $x$ transition ($x \neq a$) from $S_x$, an outgoing $a$ transition to $S'_a$ ($S \neq S'_a$) and an outgoing $y$ transition to $S_y$ (Figure 6a). Additionally, $\texttt{parity}_a(S) = \{1\}$. In this case, $\texttt{parity}_a(S_a) = 0$. This means that all words recognized by $S_a$ have an even number of trailing $a$. Consequently, a subsequent pair of $a$ has to be merged. We can then apply the same reasoning as with $\texttt{merge}_{a,b}$ and notice that the same transformations should take place. We end up in the situation depicted on Figure 6b (which is naturally very similar to Figure 4b).



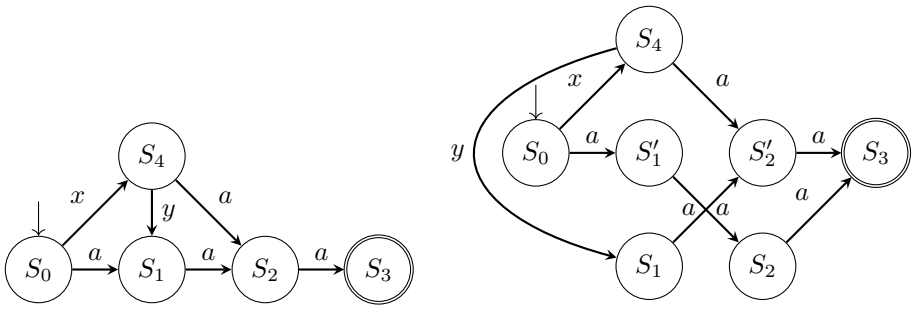

(a) (b) Case of $\texttt{parity}_a(S) = \{1\}$

Figure 6: Case of $\texttt{merge}_{a,a}$ for a state $S$ without an $a$ self-loop. The required transformations depend on $\texttt{parity}_a(S)$.

**Example 3.2.** Same as Example 3.1 but with $\texttt{parity}_a(S) = \{0\}$. In this case, $\texttt{parity}_a(S_a) = 1$ and the $(S_a, a, S)$ transition will be absorbed by a previous merge. As a result, no transformation should take place at the level of $S$.

## 3.3 General Case for $\texttt{merge}_{a,a}$

Examples 3.2 and 3.1 cover the cases $\texttt{parity}_a(S) = \{0\}$ and $\texttt{parity}_a(S) = \{1\}$, which are both quite straightforward, but not $\texttt{parity}_a(S) = \{0, 1\}$. Fortunately, directly handling this configuration is unnecessary because we can decompose a state of parity $\{0, 1\}$ in two states, one with parity $\{0\}$ and one with parity $\{1\}$, that collectively perform the same function as the original state, as illustrated on Figure 7.




(a) (b)

Figure 7: We can replace states of parity $\{0, 1\}$ (here, $S_1$ and $S_2$) with pairs of states of parity $\{0\}$ and $\{1\}$ (here $S_1$, $S'_1$ and $S_2$, $S'_2$).

We formalize this approach in Algorithm 2, whose correctness is proven below.

**Algorithm 2:** `DirectMerge` for $\texttt{merge}_{a,a}$
`ProcessState` is the same function as in Algorithm 1.
`RemoveIrrelevantStates` is based on a standard breadth-first graph exploration approach (first from the start state and then from the accept states with reverse edges).

**Input:**
- DFA $(\Sigma, \mathcal{S}, \delta, F, S_0)$ recognizing a language $\mathcal{L}$
- Token $a$

**Output:** DFA recognizing the language $\texttt{merge}_{a,a}(\mathcal{L})$

1 **function** `DirectMerge`$((\Sigma, \mathcal{S}, \delta, F, S_0), a, a)$
2  Add $aa$ to $\Sigma$
3  $E = \{S \in \mathcal{S}, S$ has incoming $a$ transitions and an outgoing $a$ transition$\}$
4  **for** $S \in E$ **do**
5  Add $S'$ to $\mathcal{S}$
6  **if** $S \in F$ **then**
7  Add $S'$ to $F$
8  **for** $S \in E$ **do**
9  **for** $S_y \in \mathcal{S}$ with $S_y = \delta(S, y)$ **do**
10  **if** $y = a$ and $S_y \in E$ **then**
11  Remove transition $(S, a, S_y)$
12  Add transitions $(S, a, S'_y)$ and $(S', a, S_y)$
13  **else**
14  Add transition $(S', y, S_y)$
15  **for** $S_a \in \mathcal{S}$ with $S = \delta(S_a, a)$ and $S_a \notin E$ **do**
16  Add transition $(S_a, a, S')$
17  Remove transition $(S_a, a, S)$
18  **for** $S \in E$ **do**
19  $(\Sigma, \mathcal{S}, \delta, F, S_0) :=$ `ProcessState`$(S', a, a)$
20  **return** `RemoveIrrelevantStates`$(\Sigma, \mathcal{S}, \delta, F, S_0)$

*Proof.* Let $E$ be the set of the states with an incoming and an outgoing $a$ transitions. We consider the DFA obtained after line 17 of Algorithm 2. We first prove by induction that each string $w \in \Sigma^*$ of length $n$ is accepted by the same state in both the initial DFA and this transformed DFA if this state does not belong to $E$ or that it is accepted by $S$ or $S'$ in the new DFA if it was accepted by $S \in E$ in the initial DFA.

$$P(n) : \forall w \in \Sigma^*, |w| = n, \begin{cases} \hat{\delta}(S_0, w) = \hat{\delta}'(S_0, w) \text{ if } \hat{\delta}(S_0, w) \notin E \\ \forall S \in E, \hat{\delta}(S_0, w) = S \implies \hat{\delta}'(S_0, w) \in \{S, S'\} \end{cases}$$

$n = 0$: $\hat{\delta}(S_0, w) = \hat{\delta}(S_0, \epsilon) = S_0$. The initial state of the new DFA is also $S_0$ so $P(0)$ is true whether $S_0$ belongs to $E$ or not.

$n > 0$: Let us write $w = vc$ with $c \in \Sigma$, $\hat{\delta}(S_0, v) = S_v$ and $\hat{\delta}(S_0, w) = S$. We can check all combinations, whether $c = a$ or not and whether $S_v$ belongs to $E$ or not, as summarized in Table 1.

If $S_v \notin E$, by assumption of induction, $\hat{\delta'}(S_0, v) = S_v$. If $c \neq a$ or $S \notin E, \hat{\delta'}(S_0, w) = \hat{\delta'}(S_v, c) = S$ because only the $a$ transitions to states of $E$ are modified. If $c = a$ and $S \in E, \hat{\delta'}(S_0, w) = \hat{\delta'}(S_v, a) = S'$.

Now, if $S_v \in E$, by assumption of induction, $\hat{\delta'}(S_0, v) \in \{S_v, S'_v\}$. The non-$a$ outgoing transitions from $S_v$ and $S'_v$ are the same (cf. line 14 of Algorithm 2) so $\hat{\delta'}(S_0, w) = \hat{\delta}(S_0, w)$ if $c \neq a$. Finally, if $c = a$, whatever the value taken by $\hat{\delta'}(S_0, v)$ in $\{S_v, S'_v\}$, $\hat{\delta'}(S_0, w) = \hat{\delta'}(S_v, a)$ will be in $\{S, S'\}$. $P(n)$ is verified in all cases. Given that for all $S \in E$ in the initial DFA, $S$ and $S'$ are accepting states of the DFA (cf. lines 6-7 of Algorithm 2), this shows that both DFA recognize the same language.

| | $S_v \in E$ and, by assumption of induction, $\hat{\delta'}(S_0, v) \in \{S_v, S'_v\}$ | $S_v \notin E$ and, by assumption of induction, $\hat{\delta'}(S_0, v) = S_v$ |
|---|---|---|
| $c = a$ | $\hat{\delta'}(S_0, w) \in \{S, S'\}$ | $\hat{\delta'}(S_0, w) \in \{S, S'\}$ |
| $c \neq a$ | $\hat{\delta'}(S_0, w) = S$ | $\hat{\delta'}(S_0, w) = S$ |

Table 1

We can also prove by induction that, for all $S \in E$, $\texttt{parity}_a(S) = \{0\}$ and $\texttt{parity}_a(S') = \{1\}$. Indeed, the last transition for a word $w = vx$ recognized by $S$ is either an $a$ transition from $S'_a$ with $S_a \in E$ or a non-$a$ transition from $S_x \notin E$. In the first case, by assumption of induction, $v$ has an odd number of trailing $a$ and therefore $w = va$ has an even number of trailing $a$. In the second case, $w$ has no trailing $a$. If now, $w = vx$ is recognized by $S'$, $x$ is necessarily equal to $a$ because the new states $S'$ only have $a$ incoming transitions. This transitions comes from either $S \in E$ or $S_a \notin E$. In both cases, $v$ has an even number of trailing $a$ and therefore $w = va$ has an odd number of trailing $a$. Otherwise said, the states with an incoming $a$ transition and an outgoing $a$ transition now all have a parity of $\{0\}$ or $\{1\}$. As discussed with Examples 3.2 and 3.1, applying `processState` to the states with a parity of $\{1\}$ as in Algorithm 1 yields the expected result. □

**Example 3.3.** Figure 8 illustrates the effect of Algorithm 2 on a state with an $a$ self-loop and $\texttt{parity}_a(S_a) = \{0\}$ for all states $S_a$ such that $\delta(S_a, a) = S$. This example will be useful in the next section.

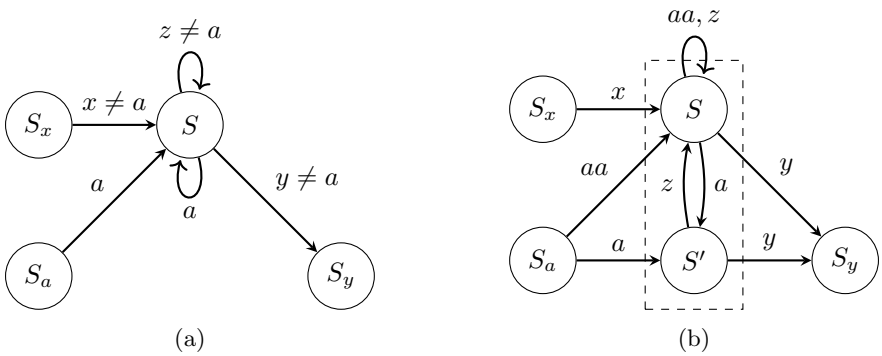



Figure 8: Effect of Algorithm 2 on a state with an $a$ self-loop and $\texttt{parity}_a(S_a) = \{0\}$ for all states $S_a$ such that $\delta(S_a, a) = S$.

# 4 `CartesianMerge` as a More Scalable Approach

`DirectMerge` can decrease but also increase the number of states and transitions to the point that it becomes a significant practical obstacle. In this section, we introduce `CartesianMerge`, a variant of `DirectMerge`, which helps circumvent this impediment.

The starting point is to notice that, for a regular language $\mathcal{L} \subset \Sigma^*$ and given that $\texttt{decode} \circ \texttt{tokenize} = \mathrm{id}_{\Sigma^*}$:

$$\texttt{tokenize}(\mathcal{L}) = \texttt{tokenize}(\Sigma^*) \cap \texttt{decode}^{-1}(\mathcal{L})$$

Since $\mathcal{L}$ and $\Sigma^*$ are both regular languages, we know from the previous sections that $\texttt{tokenize}(\mathcal{L})$ and $\texttt{tokenize}(\Sigma^*)$ are also regular languages. Moreover, $\texttt{decode}^{-1}(\mathcal{L})$ was shown to be a regular language [3]. We can then simulate the DFA recognizing $\texttt{tokenize}(\mathcal{L})$, without explicitly building it, by using a cross-product construction with the DFAs of $\texttt{tokenize}(\Sigma^*)$ and $\texttt{decode}^{-1}(\mathcal{L})$.

Fortunately, the DFA for $\texttt{tokenize}(\Sigma^*)$ has an interesting property which makes applying `DirectMerge` easier, as shown with Algorithm 3.

**Algorithm 3:** Building the DFA recognizing $\texttt{tokenize}(\Sigma^*)$ and using it to determine the disallowed tokens for a given token sequence. The disallowed tokens only depend on the latest token of this sequence.

**Input:**

- $\Sigma$, set of individual characters
- $\mathcal{M} = ((a_1, b_1), ..., (a_n, b_n))$, merge table

**Output:**

- $\mathcal{S}$, set of states
- $\delta(S_0, .) : T \mapsto \mathcal{S}$, transition function restricted to $S_0$
- $f : \mathcal{S} \mapsto \mathcal{P}(T)$, function listing the disallowed tokens per state

**function** `BuildTokenizerDFA`$(\Sigma, \mathcal{M})$
  $\mathcal{S} := \{S_0\}$
  $f(S_0) := \emptyset$
  **for** $t \in \Sigma$ **do**
    $\delta(S_0, t) := S_0$
  **for** $i \in [\![1, n]\!]$ **do**
    $a, b := a_i, b_i$
    $S, S_b := \delta(S_0, a), \delta(S_0, b)$
    **if** $b \in f(S)$ **then**
      `continue`
    **if** $a = b$ or $|\{t \in T, S = \delta(S_0, t)\}| > 1$ **then**
      Add $S'$ to $\mathcal{S}$
      $\delta(S_0, a) := S'$
      $f(S') := f(S) \cup \{b\}$
    **else**
      $f(S) := f(S) \cup \{b\}$
    $\delta(S_0, ab) := S_b$
    **for** $S_{\bar{a}} \in \mathcal{S}$ with $a \in f(S_{\bar{a}})$ **do**
      $f(S_{\bar{a}}) := f(S_{\bar{a}}) \cup \{ab\}$
  **return** $(\mathcal{S}, \delta(S_0, .), f)$
**function** `ListAllowedTokens`$(t, \delta(S_0, .), f)$
  **return** $T \setminus f(\delta(S_0, t))$

*Proof.* Let us prove by induction over the number $n$ of merge operations that not only Algorithm 3 yields the same result as `DirectMerge` but also that the DFA $(T, \mathcal{S}, \delta, F, S_0)$ recognizing $\texttt{tokenize}(\Sigma^*)$ has the following properties:

$F = \mathcal{S}$ (i.e. all states are accept states)

$\forall t \in T, \forall S \in \mathcal{S}$ such that $\delta(S, t)$ exists, $\delta(S_0, t)$ exists and $\delta(S, t) = \delta(S_0, t)$

As illustrated by Figure 9, the second property means that all transitions corresponding to a certain token point to the same state and that, if such transitions exist, there is one coming from $S_0$. This allows a compact representation

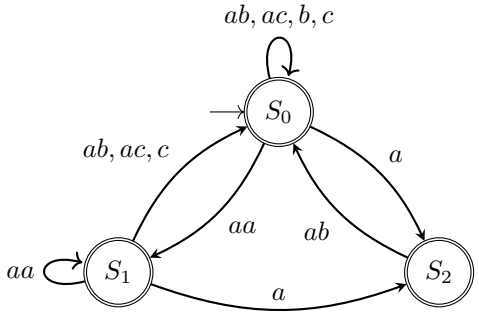



Figure 9: DFA recognizing $\texttt{tokenize}(\{a, b, c\}^*)$ with $\mathcal{M} = ((a, b), (a, a), (a, c))$. All states are accept states, all transitions with the same token point to the same state and if there is an $x$ transition, there is one starting from $S_0$

of the target DFA because we can then keep track only of:

- the transitions from $S_0$, i.e. $\delta(S_0, .)$ with the notation of Algorithm 3);
- for each state $S$, the tokens allowed for $S_0$ but not for $S$, i.e. $f(S)$ with the notation of Algorithm 3).

Let us go back to the proof. The base case $n = 0$ is straightforward because the DFA is then made of a single state $S_0$, which is both the start state and an accept state and which has one self-loop for each element of $\Sigma$.

We now assume that $n > 0$ and we define:

$$(\Sigma_{n-1}, \mathcal{S}_{n-1}, \delta_{n-1}, F_{n-1}, S_0) = \texttt{DirectMerge}_{a_{n-1}, b_{n-1}} \circ \cdots \circ \texttt{DirectMerge}_{a_0, b_0}(\Sigma, \mathcal{S}, \delta, F, S_0)$$

$$(\Sigma_n, \mathcal{S}_n, \delta_n, F_n, S_0) = \texttt{DirectMerge}_{a_n, b_n}(\Sigma_{n-1}, \mathcal{S}_{n-1}, \delta_{n-1}, F_{n-1}, S_{0,n-1})$$

We treat first the case $a_n \neq b_n$. It is easy to prove the first property mentioned above, i.e. that $F_n = \mathcal{S}_n$, because any state added by `DirectMerge` has the same accept status as another state of the initial DFA. By assumption of induction, $F_{n-1} = \mathcal{S}_{n-1}$ so any new state is necessarily an accept state.

We now select $t \in T_n, S \in \mathcal{S}_n$ such that $\delta_n(S, t)$ exists, to prove the second property mentioned above. We need to show that $\delta_n(S_0, t)$ exists and $\delta_n(S_0, t) = \delta_n(S, t)$. This is true by assumption of induction if $\texttt{DirectMerge}_{a_n, b_n}$ did not affect these transitions. If, however, these transitions were modified by $\texttt{DirectMerge}_{a_n, b_n}$, the changes were introduced at lines 8-9, 13-15, 16-17 or 24-25 of Algorithm 1[1]. Let us consider these cases one by one:

- Lines 8-9: if $S_a \in \mathcal{S}_{n-1}$ with $S = \delta_{n-1}(S_a, a)$ then, by assumption of induction, $\delta_{n-1}(S_0, a)$ exists and $\delta_{n-1}(S_0, a) = S$. Therefore, lines 8-9 also apply to $S_0$ and there will also be an $ab$ transition from $S_0$ to $S_b$.

[1]Lines 21-22 and 26-27 can be ignored because all states are accept states.

- Lines 13-15: as with lines 8-9, the affected states also include $S_0$ and there will also be an $a$ transition from $S_0$ to $S'$.
- Lines 16-17: if $S_y \in \mathcal{S}_{n-1}$ with $S_y = \delta_{n-1}(S, y)$ and $y \neq b$ then, by assumption of induction, $\delta_{n-1}(S_0, y)$ exists and $\delta_{n-1}(S_0, y) = S_y$. The added $y$ transition then points to the state which already has an incoming $y$ transition from $S_0$.
- Lines 24-25: this cannot affect $S_0$ because the condition on line 10 is always true for $S_0$. Of course, removing a transition not starting from $S_0$ cannot invalidate the second property.

We see that the second property, which was true by assumption of induction, is maintained throughout the changes of transitions triggered by $\texttt{DirectMerge}_{a_n,b_n}$.

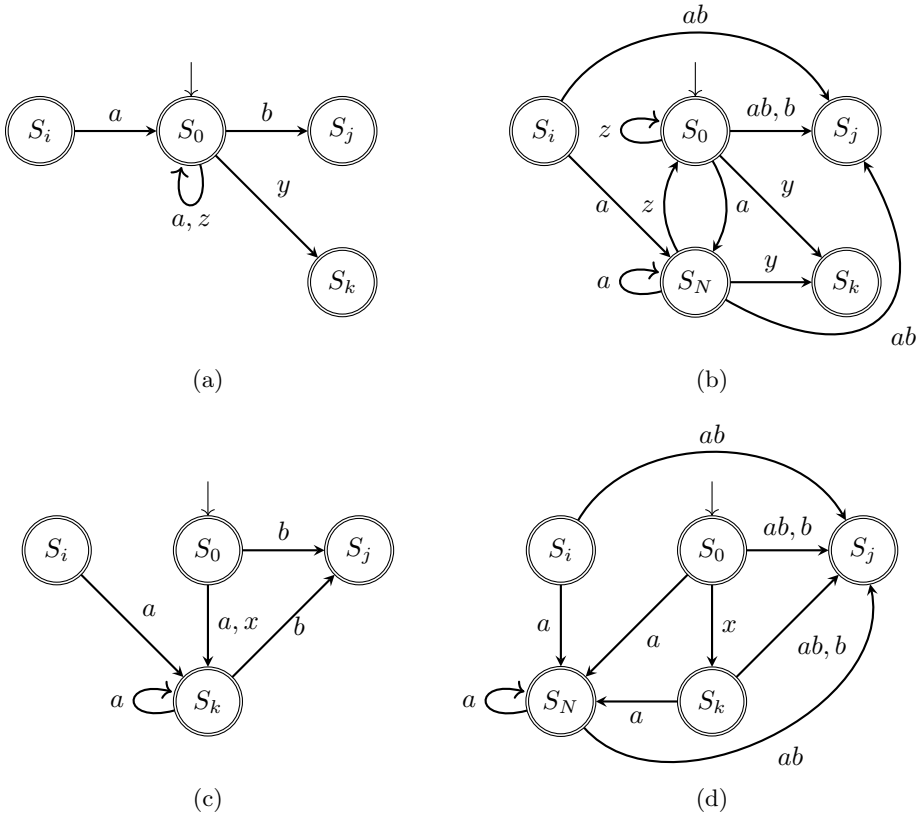



Figure 10: Effect of Algorithm 3 in the case of $\texttt{merge}_{a,b}$ with $a \neq b$ (left: before applying the algorithm, right: after).

Finally for the case $a \neq b$, we need to prove that the steps mentioned in Algorithm 3 yield the same result as Algorithm 1. Given the property just proven above, we can check that there is a direct correspondence between the

steps of Algorithm 3 and those of Algorithm 1, as shown in Table 2, which proves our result for $a \neq b$.

| **Lines in Algorithm 1** | **Equivalent lines in Algorithm 3** |
|---|---|
| 7-9 | 17-19 |
| 12 | 12 |
| 13-15 | 13 |
| 16-17 | 14 |
| 18-19 | None (because all states are accept states) |
| 21-22 | None (because all states are accept states and this case never occurs) |
| 24-25 | 16 |
| 27 | None (because all states are accept states and this case never occurs) |

Table 2: Equivalence of the steps in Algorithms 1 and 3 when $a \neq b$.

We now turn to the case $a = b$. By assumption of induction, all $a$ transitions point to the same state $S_a$. This means that only $S_a$ can have an incoming $a$ transition and an outgoing $a$ transition, which will necessarily be a self-loop, as shown on Figure 11. We are then exactly in the case of Example 3.3 because any distinct state $S$ with $\delta(S, a) = S_a$ will have a parity of $\{0\}$. Similarly as with $a \neq b$, we can then prove that the steps in Algorithm 3 are exactly the same as those shown in Example 3.3 and that both properties of the DFA are satisfied. The detailed proof in this case is left to the reader. □

## 4.1 A Brute-Force Algorithm

When proving Algorithm 3, we saw that all the transitions with a given token in the resulting DFA point to the same state. This implies that the allowed tokens after a sequence of tokens only depends on the last element of this sequence. This suggests Algorithm 4 as a brute-force alternative to build the same DFA as Algorithm 3. In practice, Algorithm 4 is not computationally efficient enough but it can be used to test an implementation of Algorithm 3.

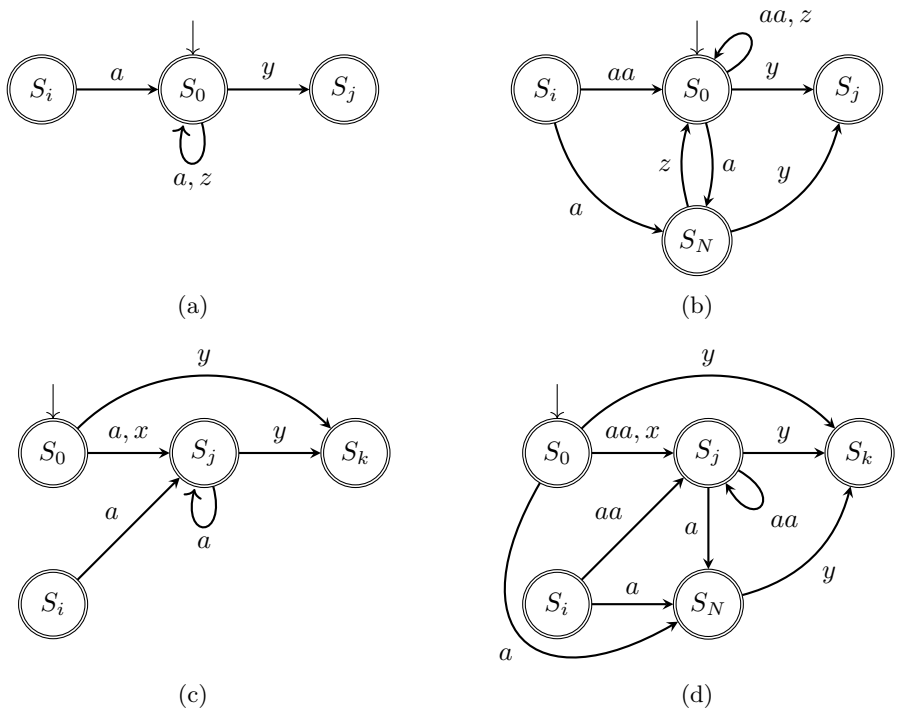



Figure 11: Effect of Algorithm 3 in the case of $\texttt{merge}_{a,a}$ (left: before applying the algorithm, right: after).

**Algorithm 4:** Brute-force algorithm to build a DFA recognizing $\texttt{tokenize}(\Sigma^*)$

**Input:** $\mathcal{T}$, a tokenizer based on a merge table
**Output:** $f : T \mapsto \mathcal{P}(T)$, function listing the allowed tokens per token

```
function TestAllTokenPairs(𝒯)
    for t₁ ∈ T do
        f(t₁) := ∅
        for t₂ ∈ T do
            if tokenize(decode((t₁, t₂))) = (t₁, t₂) then
                f(t₁) := f(t₁) ∪ {t₂}
    return f
```

## 4.2 `CartesianMerge`

We now describe `CartesianMerge`. It consists of a preprocessing procedure (Algorithm 5) which needs to be executed once per regular expression and a function (Algorithm 6) to determine the potential next tokens at decoding time.

**Algorithm 5:** `CartesianMerge` (preprocessing)
`BuildOutlinesDFA` is the algorithm described in [3].
`FindRelevantProductStates` is a breadth-first approach starting from the product start state and then from the product accept states with reverse transitions.

**Input:**

- $\mathcal{L}$, regular language
- $\Sigma$, set of individual characters
- $\mathcal{M} = ((a_1, b_1), ..., (a_n, b_n))$, merge table

**Output:**

- $(T, \mathcal{S}_1, \delta_1, F, S_0)$, DFA recognizing $\texttt{decode}^{-1}(\mathcal{L})$
- $(\mathcal{S}_2, \delta_2, f)$, representation of a DFA recognizing $\texttt{tokenize}(\Sigma^*)$
- $R \subset \mathcal{S}_1 \times \mathcal{S}_2$, list of relevant product states

```
1 function CartesianMerge(L, Σ, M)
2     (T, S_1, δ_1, F, S_0) := BuildOutlinesDFA(L)
3     (S_2, δ_2, f) := BuildTokenizerDFA(Σ, M)
4     R := FindRelevantProductStates(T, S_1, δ_1, F, S_0, δ_2, f)
5     return (T, S_1, δ_1, F, S_0), (S_2, δ_2, f), R
```

At preprocessing time (cf. Algorithm 5), we use Algorithm 3 and the algorithm described in [3]to respectively build $\texttt{DFA}_1$, the DFA recognizing $\texttt{decode}^{-1}(\mathcal{L})$ and $\texttt{DFA}_2$, the one recognizing $\texttt{tokenize}(\Sigma^*)$. Once we have done so, we can simulate a DFA recognizing $\texttt{tokenize}(\mathcal{L})$ by keeping track of a product state made of the states of both DFA. This is however not enough because we also need to make sure that a transition is both allowed and lead to a relevant state. Even though both individual DFA may only have relevant states, the product states may not be relevant. For this, we can explore the product DFA (without explicitly listing all its transitions) with a breadth-first approach starting from the product start state and then from the product accept states with reverse transitions.

At decoding time (cf. Algorithm 6), the potential tokens given the last token $t_0$ and the current state $S$ of $\texttt{DFA}_1$ verify the following conditions:

- They are allowed by $\texttt{DFA}_1$, i.e. they correspond to an outgoing transition from $S$;
- They are allowed by $\texttt{DFA}_2$, i.e. they correspond to an outgoing transition from the state corresponding to $t_0$;
- They lead to a relevant product state.

**Algorithm 6:** `CartesianMerge` (decoding)

**Input:**
- $\texttt{DFA}_1 = (T, \mathcal{S}_1, \delta_1, F, S_0)$, DFA recognizing $\texttt{decode}^{-1}(\mathcal{L})$
- $\texttt{DFA}_2 = (\mathcal{S}_2, \delta_2, f)$, DFA recognizing $\texttt{tokenize}(\Sigma^*)$
- $R \subset \mathcal{S}_1 \times \mathcal{S}_2$, list of relevant product states
- $S \in \mathcal{S}_1$, current state for $\texttt{DFA}_1$
- $t_0 \in T$, last token

**Output:** Set of allowed tokens, given the last token and the current state of $\texttt{DFA}_1$

```
1 function ListPotentialNextTokens(T, δ1, δ2, f, R, S, t0)
2     A1 := {t ∈ T, ∃St ∈ S1, δ1(S, t) = St}
3     A2 := T \ f(δ2(t0))
4     return {t ∈ A1 ∩ A2, (δ1(S, t), δ2(t)) ∈ R}
```

# 5 Implementation Details

In this section, we discuss some ways to efficiently implement the algorithms presented above.

## 5.1 Algorithm 1: `DirectMerge` for $\texttt{merge}_{a,b}$ with $a \neq b$

Beyond storing the transition function of the DFA, we should also create and maintain:

- The reverse transition function, to easily identify the previous state given a certain token (cf. lines 13 and 21 of Algorithm 1);
- Two dictionaries mapping each token to the states with an outgoing transition with this token or to the states with an incoming transition with this token, to easily identify the states affected by the algorithm (cf. line 3).

## 5.2 Algorithm 2: `DirectMerge` for $\texttt{merge}_{a,a}$

Same as with `DirectMerge` for $\texttt{merge}_{a,b}$ when $a \neq b$. Additionally and instead of removing irrelevant states at the end (cf. line 20 of Algorithm 2), we can perform the breadth-first graph exploration before creating the new states (cf. lines 4-7) to avoid creating states that will be removed right away.

## 5.3 Algorithm 3: Applying `DirectMerge` to `".*"`

On top of the list of states $\mathcal{S}$, the transition function restricted to $S_0$ and the lists of disallowed tokens per state, we should keep track of:

- The reverse transition function, to easily count the tokens leading to a certain state (cf. line 11 of Algorithm 3);

- The list of states that have a certain token in their list of disallowed tokens (to implement lines 18-19 without looping over all states).

Moreover, states with the same list of forbidden tokens can be fused at the end of `BuildTokenizerDFA` to save space and reduce the number of states to consider with `CartesianMerge`.

### 5.4 Algorithms 5 and 6: `CartesianMerge` (Preprocessing and Decoding)

Before performing a breadth-first exploration to identify the relevant states (cf. line 4 of Algorithm 5) during preprocessing, we can rearrange the transition function $\delta_1$ of the DFA recognizing $\texttt{decode}^{-1}(\mathcal{L})$ so that all $t$ transitions with the same $(\delta_1(S,t),\delta_2(t))$ are grouped together. In this way, for example when searching for reachable product states, if any $t$ transition of the $(\delta_1(S,t),\delta_2(t))$ group takes place, we can label the product state $(\delta_1(S,t),\delta_2(t))$ as reachable and move on to the next group, without testing the other transitions of the current group.

Furthermore, the condition $(\delta_1(S,t),\delta_2(t)) \in R$ at line 4 of Algorithm 6 is expensive to compute at decoding time. Since it only depends on $S$, the current state of $\texttt{DFA}_1$ and $t$, the potential next token, we can restrict $\delta_1$, the transition function of $\texttt{DFA}_1$ to the transitions pointing to relevant product states. In this way, Algorithm 5 does not need to return $R$ and Algorithm 6 can simply return $A_1 \cap A_2$.

## References


[1] Rico Sennrich, Barry Haddow, and Alexandra Birch. "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)*. Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162.

[2] Vivien Tran-Thien. *Fast, High-Fidelity LLM Decoding with Regex Constraints*. 2024. URL: https://vivien000.github.io/blog/journal/llm-decoding-with-regex-constraints.html.

[3] Brandon T. Willard and Rémi Louf. *Efficient Guided Generation for Large Language Models*. 2023. arXiv: 2307.09702 [cs.CL].