# Combinatorial Sanitization of Strings

Solon P. Pissis

The Cyprus Institute

**CPM 2026 Summer School**

# Outline

# Outline

# Definitions and Motivation

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0\,..\,|W|-1]$ is a sequence of letters over $\Sigma$.

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0..|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0\,..\,|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0..|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc
- It fuels location-based, web analytics, or bioinformatics apps

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0\,..\,|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc
- It fuels location-based, web analytics, or bioinformatics apps
- Dissemination may expose patterns modeling private knowledge

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0\,..\,|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc
- It fuels location-based, web analytics, or bioinformatics apps
- Dissemination may expose patterns modeling private knowledge
- We call these patterns sensitive

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0..|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc
- It fuels location-based, web analytics, or bioinformatics apps
- Dissemination may expose patterns modeling private knowledge
- We call these patterns sensitive

$$W = \mathtt{aabaaaababbbbaab}$$

# Definitions and Motivation

An **alphabet** $\Sigma$ is a finite set whose elements are called letters.
A **string** $W = W[0\,..\,|W|-1]$ is a sequence of letters over $\Sigma$.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbbaab}$$

- $W$ can represent location or purchase history, a DNA sequence, etc
- It fuels location-based, web analytics, or bioinformatics apps
- Dissemination may expose patterns modeling private knowledge
- We call these patterns sensitive

$$W = \mathtt{aabaaaababbbbaab}$$

**The (informal) goal of String Sanitization**

Conceal **sensitive** patterns in $W$ while maintaining **data utility**.

# Our Model

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

- **C1** No string from $\mathcal{S}$ occurs in $X$

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

- **C1** No string from $\mathcal{S}$ occurs in $X$
- **C2** The order of length-$k$ non-sensitive patterns is preserved in $X$

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

- **C1** No string from $\mathcal{S}$ occurs in $X$
- **C2** The order of length-$k$ non-sensitive patterns is preserved in $X$
- **C3** The frequency of length-$k$ non-sensitive patterns is preserved in $X$

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

- **C1** No string from $\mathcal{S}$ occurs in $X$
- **C2** The order of length-$k$ non-sensitive patterns is preserved in $X$
- **C3** The frequency of length-$k$ non-sensitive patterns is preserved in $X$

  $\Rightarrow$ No utility loss for tasks based on sequentiality and frequency

# Our Model

## The General Model

Given a set of **constraints**, determine the "optimal" sequence of **edit operations** to be applied to $W$ subject to the constraints.

By $\mathcal{S}$ we denote the set of sensitive patterns and call it the **antidictionary**.

## Our Setting

Given $W$ and a set $\mathcal{S}$ of **length-$k$** sensitive patterns, construct string $X$:

- **C1** No string from $\mathcal{S}$ occurs in $X$
- **C2** The order of length-$k$ non-sensitive patterns is preserved in $X$
- **C3** The frequency of length-$k$ non-sensitive patterns is preserved in $X$

  $\Rightarrow$ No utility loss for tasks based on sequentiality and frequency

**C3** is implied by **C2**.

# Our Problems and Results

# Our Problems and Results

**TFS** (Total order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $X \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **C2**.

# Our Problems and Results

**TFS** (Total order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $X \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **C2**.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\mathcal{S} = \{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

# Our Problems and Results

**TFS** (Total order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $X \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **C2**.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\mathcal{S} = \{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

$$W = \mathtt{aabaaaababbbaab}$$

# Our Problems and Results

**TFS** (Total order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $X \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **C2**.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\mathcal{S} = \{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

$$W = \mathtt{aabaaaababbbaab}$$

$$X = \mathtt{aabaa\#aaababbba\#baab}$$

# Our Problems and Results

**TFS** (Total order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $X \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **C2**.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\mathcal{S} = \{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

$$W = \mathtt{aabaaaababbbaab}$$

$$X = \mathtt{aabaa\#aaababbba\#baab}$$

**Theorem**

**TFS** *can be solved in* $O(||\mathcal{S}|| + k|W|)$ *time. We have* $|X| = \Omega(k|W|)$.

# Our Problems and Results

# Our Problems and Results

Could we generally hope for a shorter string?

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

**PFS** (Partial order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $Y \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **Π2**.

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

**PFS** (Partial order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $Y \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **Π2**.

$$W = \mathtt{aabaaaababbbaab}$$

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

**PFS** (Partial order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $Y \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **Π2**.

$$W = \text{aabaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

**PFS** (Partial order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $Y \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **Π2**.

$$W = \mathtt{aabaaaababbbaab}$$

$$X = \mathtt{aabaa\#aaababbba\#baab}$$
$$Y = \mathtt{aaababbba\#aabaab}$$

# Our Problems and Results

Could we generally hope for a shorter string? Relax the total order (**C2**).
Employ **Π2**: order of length-$k$ patterns in-between #s remains unchanged.

**PFS** (Partial order, Frequency, Sanitization) problem

Given $W$ and $\mathcal{S}$, construct a **shortest** $Y \in (\Sigma \sqcup \{\#\})^*$ satisfying **C1**, **Π2**.

$$W = \mathtt{aabaaaababbbaab}$$

$$X = \mathtt{aabaa\#aaababbba\#baab}$$
$$Y = \mathtt{aaababbba\#aabaab}$$

**Theorem**

**PFS** *can be solved in* $O(||\mathcal{S}|| + |W| + |Y|)$ *time.*

# Our Problems and Results

# Our Problems and Results

Unfortunately, the # occurrences reveal the position of sensitive patterns.

# Our Problems and Results

Unfortunately, the # occurrences reveal the position of sensitive patterns.
Can we hide these occurrences without introducing sensitive patterns?

# Our Problems and Results

Unfortunately, the # occurrences reveal the position of sensitive patterns.
Can we hide these occurrences without introducing sensitive patterns?

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

# Our Problems and Results

Unfortunately, the # occurrences reveal the position of sensitive patterns.
Can we hide these occurrences without introducing sensitive patterns?

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

$$U = \mathtt{aab} \quad V = \mathtt{aba}$$
$$\mathcal{S} = \{\mathtt{bbbb}, \mathtt{abba}, \mathtt{aaba}\}$$

$$\mathtt{aab\#aba}$$

$$X = \mathtt{aabbbaba}$$

# Our Problems and Results

Unfortunately, the # occurrences reveal the position of sensitive patterns.
Can we hide these occurrences without introducing sensitive patterns?

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

$$U = \texttt{aab} \quad V = \texttt{aba}$$
$$\mathcal{S} = \{\texttt{bbbb}, \texttt{abba}, \texttt{aaba}\}$$

$$\texttt{aab\#aba}$$

$$X = \texttt{aabbbaba}$$

Theorem

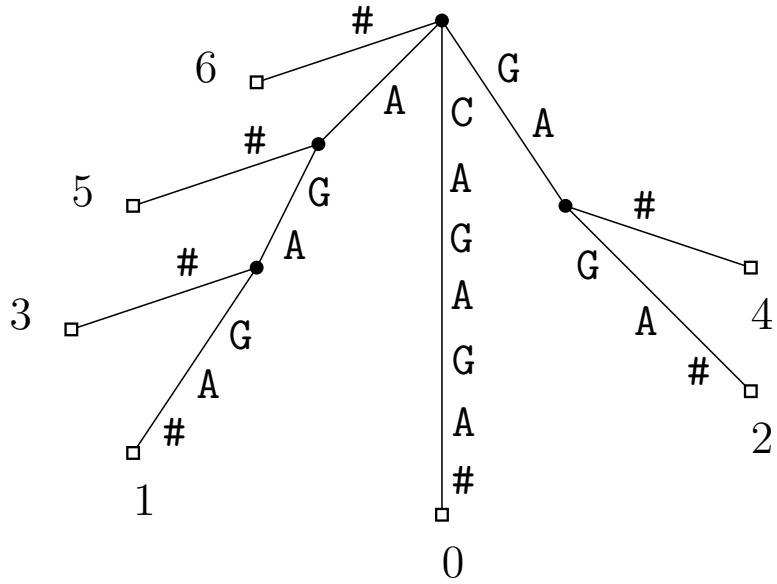
**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

# Suffix tree

The compacted trie of all the suffixes of the string.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

0 : CAGAGA#
1 : AGAGA#
2 : GAGA#
3 : AGA#
4 : GA#
5 : A#
6 : #




We assume that the terminating symbol # is lex-smallest.

# Suffix tree

The compacted trie of all the suffixes of the string.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #

5 : A#

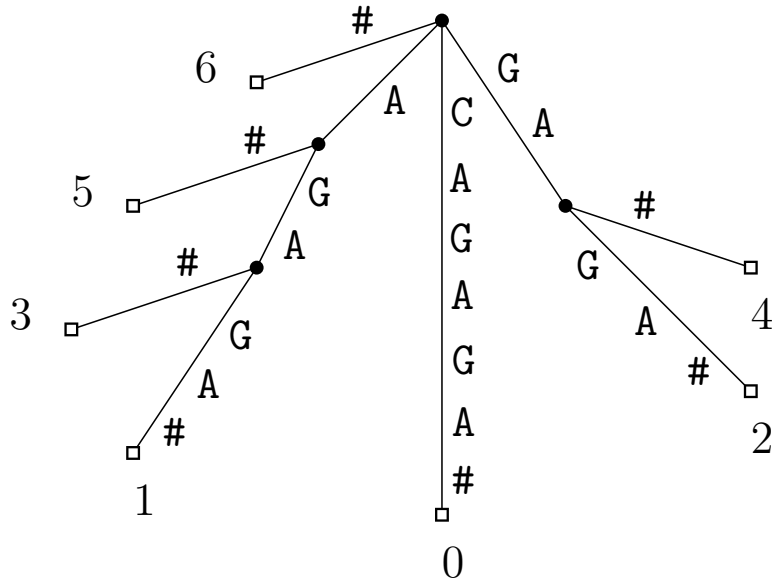
3 : AGA#

1 : AGAGA#

0 : CAGAGA#

2 : GA#

4 : GAGA#



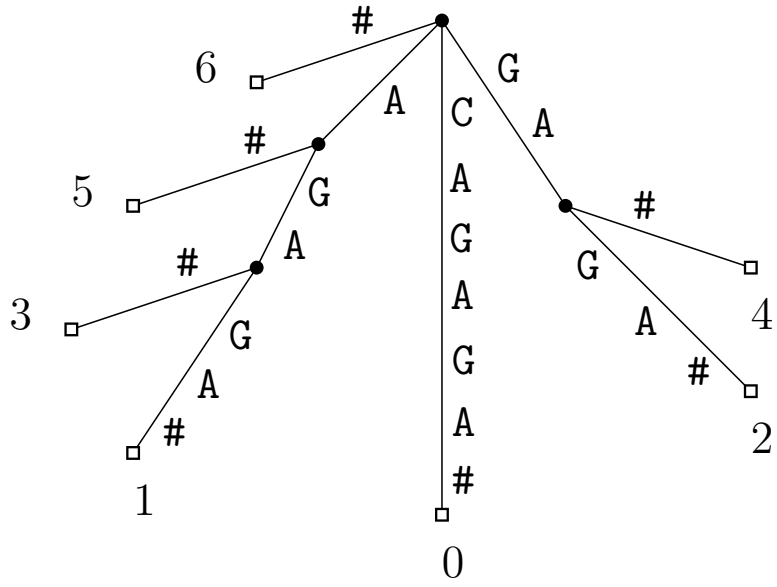

We assume that the terminating symbol # is lex-smallest.

# Suffix tree

The compacted trie of all the suffixes of the string.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #
5 : A#
3 : AGA#
1 : AGAGA#
0 : CAGAGA#
2 : GA#
4 : GAGA#




We assume that the terminating symbol # is lex-smallest.

**Theorem (Weiner, FOCS 1973)**

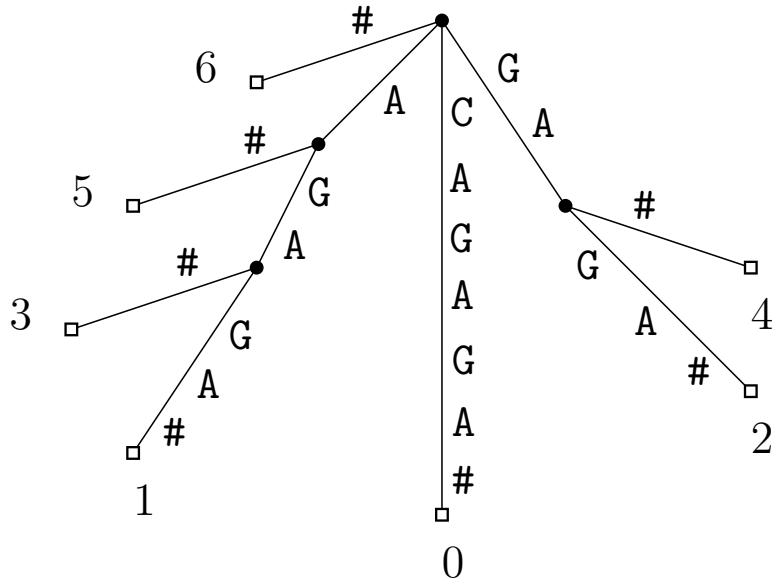
*The suffix tree of $T$ occupies $O(n)$ space.*

# Suffix tree

The compacted trie of all the suffixes of the string.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #
5 : A#
3 : AGA#
1 : AGAGA#
0 : CAGAGA#
2 : GA#
4 : GAGA#




We assume that the terminating symbol # is lex-smallest.

**Theorem (Farach, FOCS 1997)**

*The suffix tree of $T$ can be constructed in $O(n)$ time.*

# Suffix tree

Say we have a pattern $P = \texttt{AGA}$.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #

5 : A#

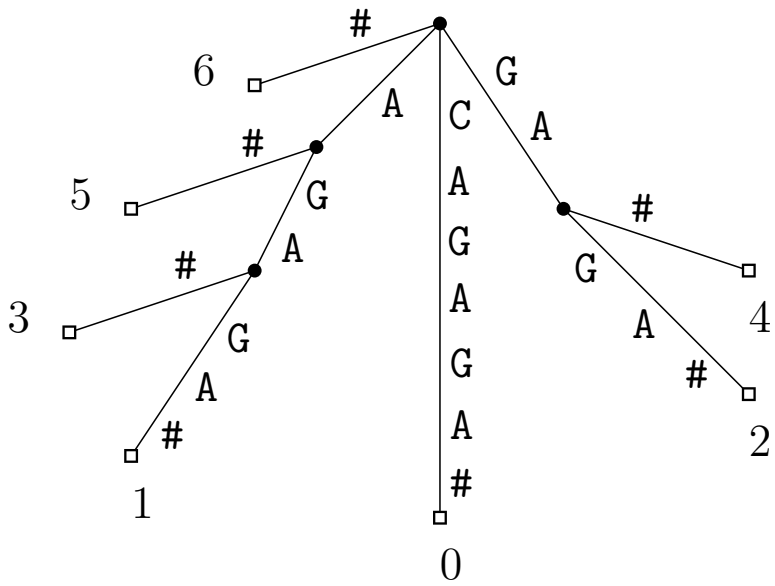
3 : AGA#

1 : AGAGA#

0 : CAGAGA#

2 : GA#

4 : GAGA#



# Suffix tree

Say we have a pattern $P = \texttt{AGA}$.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #

5 : A#

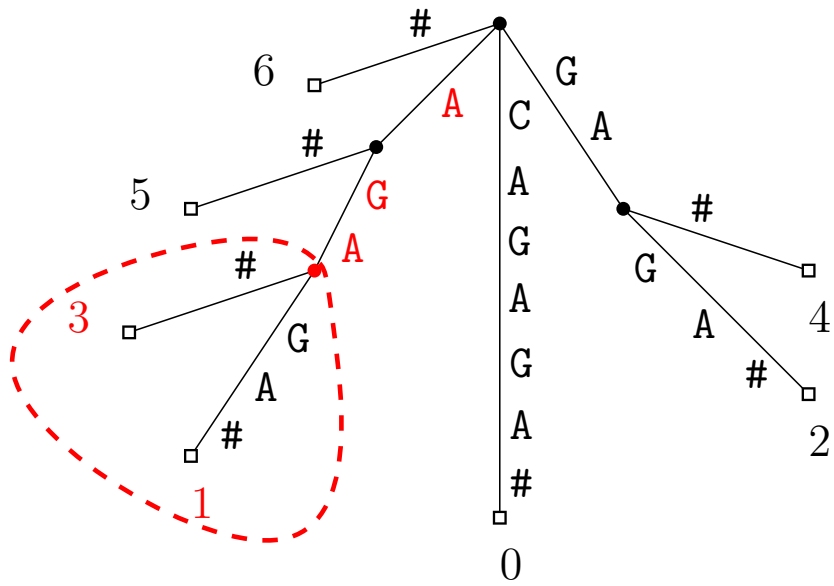
3 : AGA#

1 : AGAGA#

0 : CAGAGA#

2 : GA#

4 : GAGA#



# Suffix tree

Say we have a pattern $P = \texttt{AGA}$.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #

5 : A#

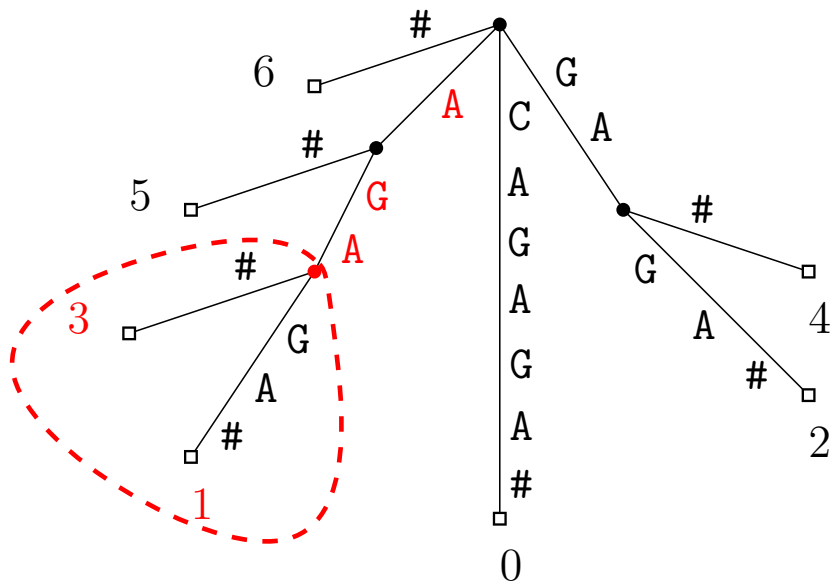
3 : AGA#

1 : AGAGA#

0 : CAGAGA#

2 : GA#

4 : GAGA#




Indeed $P = \texttt{AGA}$ occurs at positions 1 and 3 in $T$.

# Suffix tree

Say we have a pattern $P = \texttt{AGA}$.

$$T = \overset{0\,1\,2\,3\,4\,5\,6}{\texttt{CAGAGA\#}}$$

6 : #

5 : A#

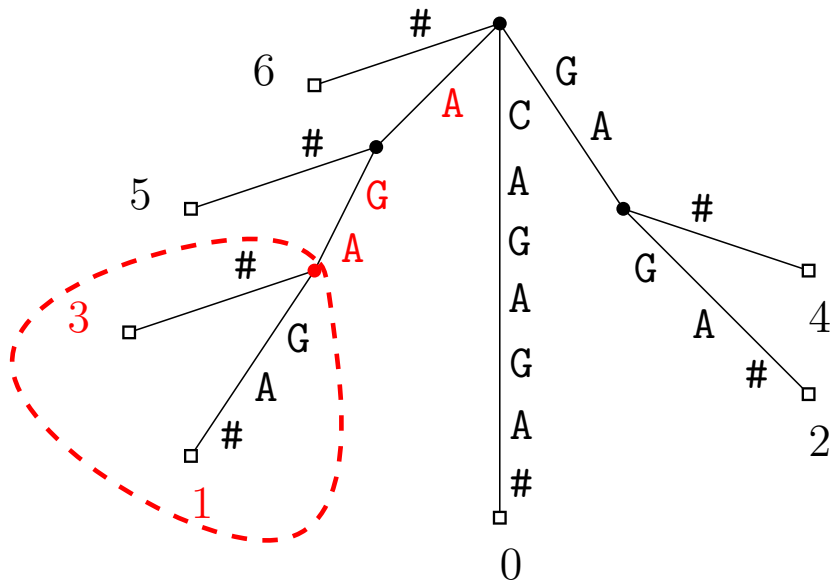
3 : AGA#

1 : AGAGA#

0 : CAGAGA#

2 : GA#

4 : GAGA#




Indeed $P = \texttt{AGA}$ occurs at positions 1 and 3 in $T$.

**Theorem (Weiner, FOCS 1973)**

*The occurrences of pattern $P$ in $T$ can be found in $O(|P| + \text{output})$ time.*

# Longest common prefix (LCP) queries

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$
**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

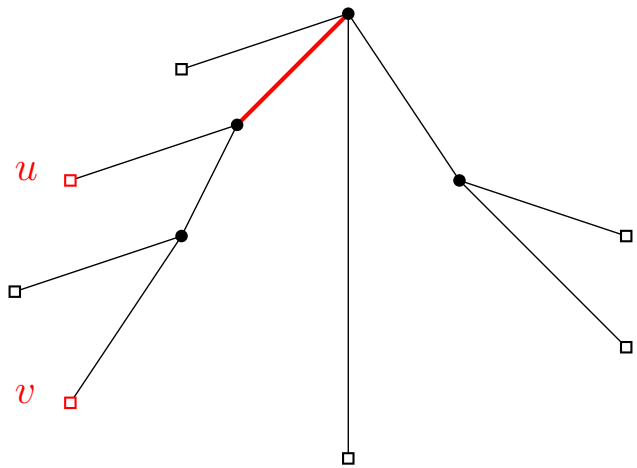
The lowest common ancestor (LCA) of two nodes $u$ and $v$ is the deepest node that is an ancestor of both $u$ and $v$.

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

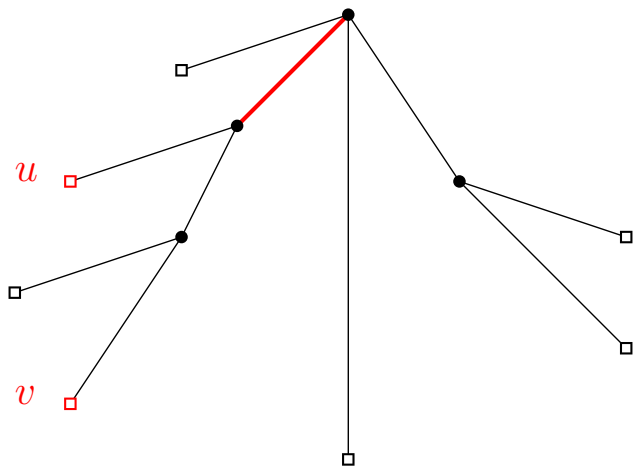
The lowest common ancestor (LCA) of two nodes $u$ and $v$ is the deepest node that is an ancestor of both $u$ and $v$.



# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$
**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

The lowest common ancestor (LCA) of two nodes $u$ and $v$ is the deepest node that is an ancestor of both $u$ and $v$.




**Theorem (Bender and Farach-Colton, LATIN 2000)**

*Any tree of size $O(N)$ can be preprocessed in $O(N)$ time and space so that the LCA of any two nodes can be computed in $O(1)$ time.*

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

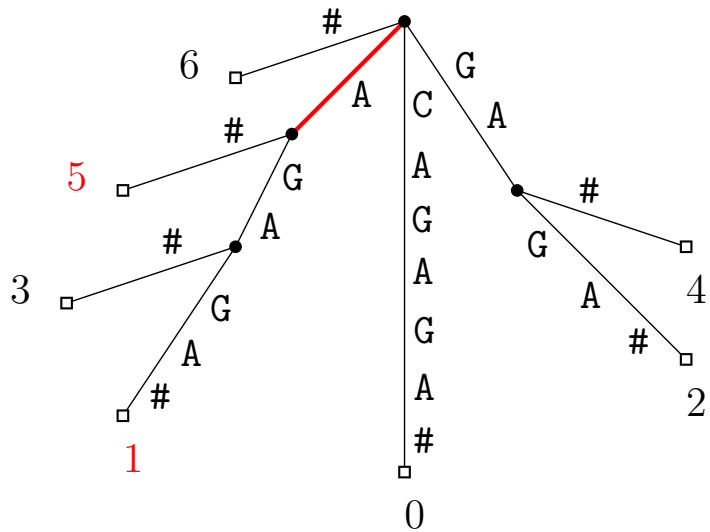
Let $T = \texttt{CAGAGA\#}$. Let $(1, 5)$ be the query. The answer is $1 = |\texttt{A}|$.

# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

Let $T = \texttt{CAGAGA\#}$. Let $(1, 5)$ be the query. The answer is $1 = |\texttt{A}|$.

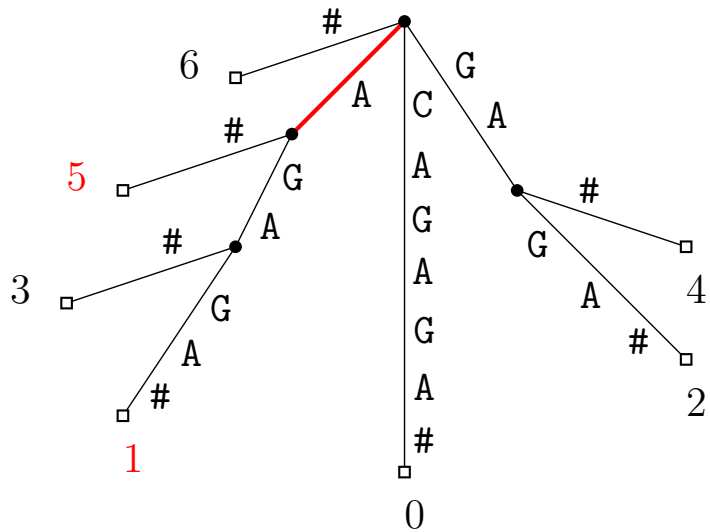


# Longest common prefix (LCP) queries

**Preprocess**: a string $T$ of length $n$

**Query**: a pair $(i, j)$; return the length of the LCP of $(T[i\,..], T[j\,..])$

Let $T = \texttt{CAGAGA\#}$. Let $(1, 5)$ be the query. The answer is $1 = |\texttt{A}|$.




**Theorem (Landau and Vishkin, TCS 1986)**

*LCP queries in $T$ can be answered in $O(1)$ time after $O(n)$ time preprocessing.*

# Solving TFS: Intuition

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa`

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`)

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: $\ldots$ `baaa` $\rightarrow$ $\ldots$ `baa#aaa` (because we must hide `baaa`)
  - **R2**: $\ldots$ `aaa#aaab` $\rightarrow$ $\ldots$ `aaab` (because a shorter string is possible)

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab (because a shorter string is possible)

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab (because a shorter string is possible)

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

- Implementing **R1** and **R2** carefully produces $X$ in $O(k|W|)$ time:

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab (because a shorter string is possible)

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

- Implementing **R1** and **R2** carefully produces $X$ in $O(k|W|)$ time:
  - If the last marked sens. pattern and the current non-sens. have an LCP $(k-1)$, append the last letter of the latter to $X$

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab (because a shorter string is possible)

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

- Implementing **R1** and **R2** carefully produces $X$ in $O(k|W|)$ time:
  - If the last marked sens. pattern and the current non-sens. have an LCP $(k-1)$, append the last letter of the latter to $X$
  - Else append # and the current non-sens. pattern to $X$

# Solving TFS: Intuition

- Read $W$ letter by letter from left to right
- If the length-$k$ substring read is non-sensitive append it to $X$
- Otherwise:
  - **R1**: ...baaa $\rightarrow$ ...baa#aaa (because we must hide baaa)
  - **R2**: ...aaa#aaab $\rightarrow$ ...aaab (because a shorter string is possible)

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

- Implementing **R1** and **R2** carefully produces $X$ in $O(k|W|)$ time:
  - If the last marked sens. pattern and the current non-sens. have an LCP $(k-1)$, append the last letter of the latter to $X$
  - Else append # and the current non-sens. pattern to $X$
  - The check is implemented in $O(1)$ time using LCP queries

# Algorithm Pseudocode

# Algorithm Pseudocode

**Preprocessing: $O(|W| + ||\mathcal{S}||)$ time**

We find the occurrences of the sensitive patterns using the suffix tree.

# Algorithm Pseudocode

**Preprocessing: $O(|W| + ||\mathcal{S}||)$ time**

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance
6: **if** the leftmost is sens. and the rightmost is not **then**

# Algorithm Pseudocode

**Preprocessing: $O(|W| + ||\mathcal{S}||)$ time**

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance
6: **if** the leftmost is sens. and the rightmost is not **then**
7: **if** the last marked sens. and the current non-sens. have LCP $(k-1)$ **then**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance
6: **if** the leftmost is sens. and the rightmost is not **then**
7: **if** the last marked sens. and the current non-sens. have LCP $(k-1)$ **then**
8: Append the last letter of the latter to $X$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance
6: **if** the leftmost is sens. and the rightmost is not **then**
7: **if** the last marked sens. and the current non-sens. have LCP $(k-1)$ **then**
8: Append the last letter of the latter to $X$
9: **else**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

We find the occurrences of the sensitive patterns using the suffix tree.

1: Append the first non-sens. pattern to $X$
2: Examine two consecutive patterns
3: **if** both are non-sens. **then** append the last letter of the rightmost to $X$
4: **if** only the rightmost is sens. **then** mark it and advance
5: **if** both are sens. **then** advance
6: **if** the leftmost is sens. and the rightmost is not **then**
7:  **if** the last marked sens. and the current non-sens. have LCP $(k-1)$ **then**
8:  Append the last letter of the latter to $X$
9:  **else**
10:  Append # and the current non-sens. pattern to $X$

# Full Example

# Full Example

**Given:**

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$
- Non-sensitive ($N$): All other length-4 substrings

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$
- Non-sensitive ($N$): All other length-4 substrings

**Initialization:**

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$
- Non-sensitive ($N$): All other length-4 substrings

**Initialization:**

- The first $N$ pattern is $P_1 = \texttt{aaba}$

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$
- Non-sensitive ($N$): All other length-4 substrings

**Initialization:**

- The first $N$ pattern is $P_1 = \texttt{aaba}$
- Append $P_1$ to $X$

# Full Example

**Given:**

- $W = \texttt{aabaaaababbbaab}$, $k = 4$
- Sensitive ($S$): $\mathcal{S} = \{\texttt{baaa}, \texttt{aaaa}, \texttt{bbaa}\}$
- Non-sensitive ($N$): All other length-4 substrings

**Initialization:**

- The first $N$ pattern is $P_1 = \texttt{aaba}$
- Append $P_1$ to $X$

**Initial Status**

$X = \texttt{aaba}$

# Full Example

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X = \texttt{aabaa}$

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X = \texttt{aabaa}$

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X = \texttt{aabaa}$

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X = \texttt{aabaa}$

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X =$ aabaa

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance
- $X =$ aabaa (unchanged)

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X =$ aabaa

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance
- $X =$ aabaa (unchanged)

**Step 3: Examine $P_3$ (baaa, $S$) and $P_4$ (aaaa, $S$)**

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X = \texttt{aabaa}$

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance
- $X = \texttt{aabaa}$ (unchanged)

**Step 3: Examine $P_3$ (baaa, $S$) and $P_4$ (aaaa, $S$)**

- **Condition**: Both are $S$

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X =$ aabaa

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance
- $X =$ aabaa (unchanged)

**Step 3: Examine $P_3$ (baaa, $S$) and $P_4$ (aaaa, $S$)**

- **Condition**: Both are $S$
- **Action**: Advance

# Full Example

**Step 1: Examine $P_1$ (aaba, $N$) and $P_2$ (abaa, $N$)**

- **Condition**: Both are $N$
- **Action**: Append last letter of rightmost (a) to $X$
- $X =$ aabaa

**Step 2: Examine $P_2$ (abaa, $N$) and $P_3$ (baaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_3$ as last $S$ pattern and advance
- $X =$ aabaa (unchanged)

**Step 3: Examine $P_3$ (baaa, $S$) and $P_4$ (aaaa, $S$)**

- **Condition**: Both are $S$
- **Action**: Advance
- $X =$ aabaa (unchanged)

# Full Example

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_3$: baaa) and current $N$ ($P_5$: aaab) have LCP $(k-1)$?

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_3$: baaa) and current $N$ ($P_5$: aaab) have LCP $(k-1)$?
  - No

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_3$: baaa) and current $N$ ($P_5$: aaab) have LCP $(k-1)$?
  - No
- **Action**: Append # and $P_5$ to $X$

# Full Example

**Step 4: Examine $P_4$ (aaaa, $S$) and $P_5$ (aaab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_3$: baaa) and current $N$ ($P_5$: aaab) have LCP $(k-1)$?
  - No
- **Action**: Append # and $P_5$ to $X$

**Current Status**

$X =$ aabaa#aaab

# Full Example

# Full Example

**Steps 5 to 9: Sequence of $N$ patterns**

# Full Example

**Steps 5 to 9: Sequence of $N$ patterns**

- $P_6$ (aaba), $P_7$ (abab), $P_8$ (babb), $P_9$ (abbb), $P_{10}$ (bbba)

# Full Example

**Steps 5 to 9: Sequence of $N$ patterns**

- $P_6$ (aaba), $P_7$ (abab), $P_8$ (babb), $P_9$ (abbb), $P_{10}$ (bbba)
- **Condition**: In all pairs, both are $N$

# Full Example

**Steps 5 to 9: Sequence of $N$ patterns**

- $P_6$ (aaba), $P_7$ (abab), $P_8$ (babb), $P_9$ (abbb), $P_{10}$ (bbba)
- **Condition**: In all pairs, both are $N$
- **Action**: Append the last letter of each to $X$

# Full Example

**Steps 5 to 9: Sequence of $N$ patterns**

- $P_6$ (aaba), $P_7$ (abab), $P_8$ (babb), $P_9$ (abbb), $P_{10}$ (bbba)
- **Condition**: In all pairs, both are $N$
- **Action**: Append the last letter of each to $X$

## Current Status

$X = \texttt{aabaa\#aaaba} \quad (P_6)$
$X = \texttt{aabaa\#aaabab} \quad (P_7)$
$X = \texttt{aabaa\#aaababb} \quad (P_8)$
$X = \texttt{aabaa\#aaababbb} \quad (P_9)$
$X = \texttt{aabaa\#aaababbba} \quad (P_{10})$

# Full Example

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_{11}$: bbaa) and current $N$ ($P_{12}$: baab) have an LCP of length $k-1$?

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_{11}$: bbaa) and current $N$ ($P_{12}$: baab) have an LCP of length $k-1$?
  - No

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_{11}$: bbaa) and current $N$ ($P_{12}$: baab) have an LCP of length $k-1$?
  - No
- **Action**: Append # and $P_{12}$ to $X$

# Full Example

**Step 10: Examine $P_{10}$ (bbba, $N$) and $P_{11}$ (bbaa, $S$)**

- **Condition**: Rightmost is $S$
- **Action**: Mark $P_{11}$ as last $S$ pattern and advance

**Step 11: Examine $P_{11}$ (bbaa, $S$) and $P_{12}$ (baab, $N$)**

- **Condition**: Leftmost is $S$, rightmost is not
- **Check**: Do the last marked $S$ ($P_{11}$: bbaa) and current $N$ ($P_{12}$: baab) have an LCP of length $k-1$?
  - No
- **Action**: Append # and $P_{12}$ to $X$

**Output**

$X =$ aabaa#aaababbba#baab

# Asymptotic Optimality: Lower Bound Construction

# Asymptotic Optimality: Lower Bound Construction

## Lemma

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

# Asymptotic Optimality: Lower Bound Construction

## Lemma

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

## Question

Can you think of such an instance?

# Asymptotic Optimality: Lower Bound Construction

## Lemma

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

## Question

Can you think of such an instance?

**Construction:**

# Asymptotic Optimality: Lower Bound Construction

**Lemma**

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

**Question**

Can you think of such an instance?

**Construction:**

- Let $W$ be a **de Bruijn sequence** of order $(k-1)$ over $\Sigma$

# Asymptotic Optimality: Lower Bound Construction

**Lemma**

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

**Question**

Can you think of such an instance?

**Construction:**

- Let $W$ be a **de Bruijn sequence** of order $(k-1)$ over $\Sigma$
- By definition: Every string in $\Sigma^{k-1}$ occurs **exactly once** in $W$

# Asymptotic Optimality: Lower Bound Construction

**Lemma**

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

**Question**

Can you think of such an instance?

**Construction:**

- Let $W$ be a **de Bruijn sequence** of order $(k-1)$ over $\Sigma$
- By definition: Every string in $\Sigma^{k-1}$ occurs **exactly once** in $W$
- Let $P_i = W[i \,..\, i+k-1]$ be the $i$-th length-$k$ pattern in $W$

# Asymptotic Optimality: Lower Bound Construction

**Lemma**

*We have* $|X| = \Omega(k|W|)$ *in the worst case.*

**Question**

Can you think of such an instance?

**Construction:**

- Let $W$ be a **de Bruijn sequence** of order $(k-1)$ over $\Sigma$
- By definition: Every string in $\Sigma^{k-1}$ occurs **exactly once** in $W$
- Let $P_i = W[i \,..\, i+k-1]$ be the $i$-th length-$k$ pattern in $W$
- Assign $P_i$ to be sensitive if $i$ is even, and non-sensitive if $i$ is odd

# Asymptotic Optimality: Lower Bound Construction

**Lemma**

*We have $|X| = \Omega(k|W|)$ in the worst case.*

**Question**

Can you think of such an instance?

**Construction:**

- Let $W$ be a **de Bruijn sequence** of order $(k-1)$ over $\Sigma$
- By definition: Every string in $\Sigma^{k-1}$ occurs **exactly once** in $W$
- Let $P_i = W[i \,..\, i+k-1]$ be the $i$-th length-$k$ pattern in $W$
- Assign $P_i$ to be sensitive if $i$ is even, and non-sensitive if $i$ is odd

As a consequence, the sequence of non-sensitive patterns $\langle P_1, P_3, P_5, \ldots \rangle$ contains roughly $|W|/2$ elements.

# No Overlap

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

- An overlap of $(k-1)$ requires the length-$(k-1)$ suffix of $P_i$ to exactly match the length-$(k-1)$ prefix of $P_{i+2}$

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

- An overlap of $(k-1)$ requires the length-$(k-1)$ suffix of $P_i$ to exactly match the length-$(k-1)$ prefix of $P_{i+2}$
- The suffix of $P_i$ is $W[i+1 \ldots i+k-1]$

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

- An overlap of $(k-1)$ requires the length-$(k-1)$ suffix of $P_i$ to exactly match the length-$(k-1)$ prefix of $P_{i+2}$
- The suffix of $P_i$ is $W[i+1 \ldots i+k-1]$
- The prefix of $P_{i+2}$ is $W[i+2 \ldots i+k]$

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

- An overlap of $(k-1)$ requires the length-$(k-1)$ suffix of $P_i$ to exactly match the length-$(k-1)$ prefix of $P_{i+2}$
- The suffix of $P_i$ is $W[i+1 \ldots i+k-1]$
- The prefix of $P_{i+2}$ is $W[i+2 \ldots i+k]$
- Because $W$ is a de Bruijn sequence of order $(k-1)$, all length-$(k-1)$ substrings are distinct

# No Overlap

To pack non-sens. patterns, $P_i$ and $P_{i+2}$ must overlap by $(k-1)$ letters.

**Lemma**

*$P_i$ and $P_{i+2}$ cannot overlap by $(k-1)$ letters.*

**Proof.**

- An overlap of $(k-1)$ requires the length-$(k-1)$ suffix of $P_i$ to exactly match the length-$(k-1)$ prefix of $P_{i+2}$
- The suffix of $P_i$ is $W[i+1 \ldots i+k-1]$
- The prefix of $P_{i+2}$ is $W[i+2 \ldots i+k]$
- Because $W$ is a de Bruijn sequence of order $(k-1)$, all length-$(k-1)$ substrings are distinct
- Therefore, $W[i+1 \ldots i+k-1] \neq W[i+2 \ldots i+k]$

# Bounding the Length of $X$

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$
- Since no two consecutive patterns in $\langle P_1, P_3, P_5, \ldots \rangle$ can overlap, their occurrences in $X$ are completely separated (using the # separator) to avoid forming sensitive or novel patterns

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$
- Since no two consecutive patterns in $\langle P_1, P_3, P_5, \ldots \rangle$ can overlap, their occurrences in $X$ are completely separated (using the # separator) to avoid forming sensitive or novel patterns
- Hence, each $P_i$ contributes $\Theta(k)$ letters to $X$

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$
- Since no two consecutive patterns in $\langle P_1, P_3, P_5, \ldots \rangle$ can overlap, their occurrences in $X$ are completely separated (using the # separator) to avoid forming sensitive or novel patterns
- Hence, each $P_i$ contributes $\Theta(k)$ letters to $X$
- Since there are $|W|/2$ such patterns:

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$
- Since no two consecutive patterns in $\langle P_1, P_3, P_5, \ldots \rangle$ can overlap, their occurrences in $X$ are completely separated (using the # separator) to avoid forming sensitive or novel patterns
- Hence, each $P_i$ contributes $\Theta(k)$ letters to $X$
- Since there are $|W|/2$ such patterns:

$$|X| \geq k \cdot \frac{|W|}{2} \implies |X| = \Omega(k|W|)$$

# Bounding the Length of $X$

Establishing the $\Omega(k|W|)$ bound:

- By **C2**, $X$ must contain the sequence $\langle P_1, P_3, P_5, \ldots \rangle$
- By **C1**, no sensitive patterns can be formed in $X$
- Since no two consecutive patterns in $\langle P_1, P_3, P_5, \ldots \rangle$ can overlap, their occurrences in $X$ are completely separated (using the # separator) to avoid forming sensitive or novel patterns
- Hence, each $P_i$ contributes $\Theta(k)$ letters to $X$
- Since there are $|W|/2$ such patterns:

$$|X| \geq k \cdot \frac{|W|}{2} \implies |X| = \Omega(k|W|)$$

**Theorem**

**TFS** *can be solved in* $O(||\mathcal{S}|| + k|W|)$ *time. We have* $|X| = \Omega(k|W|)$.

# Solving PFS: Intuition

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \texttt{aabaaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

$$Y = \texttt{aaababbba\#aabaab}$$

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \text{aabaaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \text{aabaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \text{aabaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$
- **Main Ideas**:

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

$$Y = \texttt{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$
- **Main Ideas**:
  - Construct a compact (implicit) representation of $X$

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \texttt{aabaaaababbbaab}$$

$$X = \texttt{aabaa\#aaababbba\#baab}$$

$$Y = \texttt{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$
- **Main Ideas**:
  - Construct a compact (implicit) representation of $X$
  - Assign **ID**s to blocks' prefixes and suffixes of length $(k-1)$ of $X$

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \text{aabaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$
- **Main Ideas**:
  - Construct a compact (implicit) representation of $X$
  - Assign **ID**s to blocks' prefixes and suffixes of length $(k-1)$ of $X$
  - Ignore the middle part of the blocks (as they play no role)

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Intuition

- If blocks in-between #s **overlap** by $(k-1)$ letters, then we can further apply **R2** while still satisfying **Π2**

$$W = \text{aabaaaababbbaab}$$

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete[1]
- **Observation**: Overlaps allowed are of fixed length $(k-1)$
- **Main Ideas**:
  - Construct a compact (implicit) representation of $X$
  - Assign **ID**s to blocks' prefixes and suffixes of length $(k-1)$ of $X$
  - Ignore the middle part of the blocks (as they play no role)
  - Solve SCS for a collection of two-letter strings.

[1] Gallant, Maier, Storer, JCSS, 1980

# Solving PFS: Algorithm Overview

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.
2. Identify the length-$(k-1)$ prefix and suffix of each block $X_i$ in $X$

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.
2. Identify the length-$(k-1)$ prefix and suffix of each block $X_i$ in $X$
3. Map these prefixes and suffixes to their lexicographic integer ranks

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.
2. Identify the length-$(k-1)$ prefix and suffix of each block $X_i$ in $X$
3. Map these prefixes and suffixes to their lexicographic integer ranks
4. Represent each block $X_i$ as the **2-letter string**

$$(\text{prefix rank} \cdot \text{suffix rank})$$

to form a collection $B'$

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.
2. Identify the length-$(k-1)$ prefix and suffix of each block $X_i$ in $X$
3. Map these prefixes and suffixes to their lexicographic integer ranks
4. Represent each block $X_i$ as the **2-letter string**

$$(\text{prefix rank} \cdot \text{suffix rank})$$

   to form a collection $B'$
5. Solve the Shortest Common Superstring (SCS) problem over $B'$

# Solving PFS: Algorithm Overview

1. Compacted $X$: instead of $\Theta(k)$ letters, write an interval $[i, j]$.
2. Identify the length-$(k-1)$ prefix and suffix of each block $X_i$ in $X$
3. Map these prefixes and suffixes to their lexicographic integer ranks
4. Represent each block $X_i$ as the **2-letter string**

$$(\text{prefix rank} \cdot \text{suffix rank})$$

   to form a collection $B'$
5. Solve the Shortest Common Superstring (SCS) problem over $B'$
6. Construct $Y$ using the output superstring

# Algorithm Pseudocode

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**
6: Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**
6: Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
7: Add $p$ to $P$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**
6: Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
7: Add $p$ to $P$
8: **while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
Initialize trail set $P \leftarrow \emptyset$
**while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
    Start trail $p$ at $v$
    **while** $v$ has an outgoing edge $v \to w$ **do**
        Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
    Add $p$ to $P$
**while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
    Traverse remaining edges to form a cycle $c$; remove from $E$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**
6: Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
7: Add $p$ to $P$
8: **while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
9: Traverse remaining edges to form a cycle $c$; remove from $E$
10: **if** $c$ intersects some trail $p \in P$ **then**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

1: Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
2: Initialize trail set $P \leftarrow \emptyset$
3: **while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
4: Start trail $p$ at $v$
5: **while** $v$ has an outgoing edge $v \to w$ **do**
6: Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
7: Add $p$ to $P$
8: **while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
9: Traverse remaining edges to form a cycle $c$; remove from $E$
10: **if** $c$ intersects some trail $p \in P$ **then**
11: Splice $c$ into $p$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
Initialize trail set $P \leftarrow \emptyset$
**while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
    Start trail $p$ at $v$
    **while** $v$ has an outgoing edge $v \to w$ **do**
        Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
    Add $p$ to $P$
**while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
    Traverse remaining edges to form a cycle $c$; remove from $E$
    **if** $c$ intersects some trail $p \in P$ **then**
        Splice $c$ into $p$
    **else**

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
Initialize trail set $P \leftarrow \emptyset$
**while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
    Start trail $p$ at $v$
    **while** $v$ has an outgoing edge $v \to w$ **do**
        Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
    Add $p$ to $P$
**while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
    Traverse remaining edges to form a cycle $c$; remove from $E$
    **if** $c$ intersects some trail $p \in P$ **then**
        Splice $c$ into $p$
    **else**
        Break $c$ at an arbitrary node to form a new trail $p'$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
Initialize trail set $P \leftarrow \emptyset$
**while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
    Start trail $p$ at $v$
    **while** $v$ has an outgoing edge $v \to w$ **do**
        Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
    Add $p$ to $P$
**while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
    Traverse remaining edges to form a cycle $c$; remove from $E$
    **if** $c$ intersects some trail $p \in P$ **then**
        Splice $c$ into $p$
    **else**
        Break $c$ at an arbitrary node to form a new trail $p'$
        Add $p'$ to $P$

# Algorithm Pseudocode

## Preprocessing: $O(|W| + ||\mathcal{S}||)$ time

Construct (compacted) $X$ and then $B'$ using the suffix tree.

Build directed multigraph $G = (V, E)$ with an edge $u \to v$ for each $uv \in B'$
Initialize trail set $P \leftarrow \emptyset$
**while** $\exists$ node $v$ with $\text{IN}(v) < \text{OUT}(v)$ **do**
    Start trail $p$ at $v$
    **while** $v$ has an outgoing edge $v \to w$ **do**
        Traverse edge, remove $v \to w$ from $E$, and set $v \leftarrow w$
    Add $p$ to $P$
**while** $E \neq \emptyset$ **do** *// Only disjoint cycles remain*
    Traverse remaining edges to form a cycle $c$; remove from $E$
    **if** $c$ intersects some trail $p \in P$ **then**
        Splice $c$ into $p$
    **else**
        Break $c$ at an arbitrary node to form a new trail $p'$
        Add $p'$ to $P$
**return** $P$

# Optimality

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$
- A superstring of $B'$ corresponds to an **edge-disjoint trail cover** of $G$

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$
- A superstring of $B'$ corresponds to an **edge-disjoint trail cover** of $G$
- The length of a superstring generated by a set $P$ of trails is $|E| + |P|$:

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$
- A superstring of $B'$ corresponds to an **edge-disjoint trail cover** of $G$
- The length of a superstring generated by a set $P$ of trails is $|E| + |P|$:
  - Each traversed edge adds 1 letter

# Optimality

**Lemma**

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$
- A superstring of $B'$ corresponds to an **edge-disjoint trail cover** of $G$
- The length of a superstring generated by a set $P$ of trails is $|E| + |P|$:
  - Each traversed edge adds 1 letter
  - The starting node of each trail adds 1 letter

# Optimality

## Lemma

*The algorithm computes the optimal (shortest) common superstring of $B'$.*

**Proof Sketch:**

- Every 2-letter string of $B'$ corresponds to an edge in $G$
- A superstring of $B'$ corresponds to an **edge-disjoint trail cover** of $G$
- The length of a superstring generated by a set $P$ of trails is $|E| + |P|$:
  - Each traversed edge adds 1 letter
  - The starting node of each trail adds 1 letter

## Observation

Since $|E|$ is fixed, minimizing the length of the superstring is equivalent to **minimizing the number of trails** $|P|$.

# Optimality

# Optimality

**Handling Trails:**

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

**Handling Cycles (if $G$ is not empty):**

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

**Handling Cycles (if $G$ is not empty):**

- By construction, $\text{IN}(u) = \text{OUT}(u)$ for all remaining nodes

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

**Handling Cycles (if $G$ is not empty):**

- By construction, $\text{IN}(u) = \text{OUT}(u)$ for all remaining nodes
- Thus, the remaining graph consists **only of cycles**

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

**Handling Cycles (if $G$ is not empty):**

- By construction, $\text{IN}(u) = \text{OUT}(u)$ for all remaining nodes
- Thus, the remaining graph consists **only of cycles**
- Splicing a cycle into a trail in $P$ keeps $|P|$ the same

# Optimality

**Handling Trails:**

- While $\exists v$ with $\text{IN}(v) < \text{OUT}(v)$, start a trail from $v$
- Traverse consecutive edges and delete them, updating degrees
- Stop when a node $v'$ with $\text{OUT}(v') = 0$ is reached
- Add trail $p = v \ldots v'$ to the set $P$

**Handling Cycles (if $G$ is not empty):**

- By construction, $\text{IN}(u) = \text{OUT}(u)$ for all remaining nodes
- Thus, the remaining graph consists **only of cycles**
- Splicing a cycle into a trail in $P$ keeps $|P|$ the same
- Isolated cycles cannot be spliced and require 1 new trail

# Optimality

# Optimality

## Why is this decomposition minimal?

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge
- Therefore, at most $\text{IN}(v)$ outgoing edges can belong to trails that originated at some node other than $v$

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge
- Therefore, at most $\text{IN}(v)$ outgoing edges can belong to trails that originated at some node other than $v$
- The remaining $(\text{OUT}(v) - \text{IN}(v)) > 0$ outgoing edges must be the first edge of a new trail. Thus the lower bound:

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge
- Therefore, at most $\text{IN}(v)$ outgoing edges can belong to trails that originated at some node other than $v$
- The remaining $(\text{OUT}(v) - \text{IN}(v)) > 0$ outgoing edges must be the first edge of a new trail. Thus the lower bound:

$$|P^*| \geq \sum_{v \in V} \max(0, \text{OUT}(v) - \text{IN}(v)).$$

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge
- Therefore, at most $\text{IN}(v)$ outgoing edges can belong to trails that originated at some node other than $v$
- The remaining $(\text{OUT}(v) - \text{IN}(v)) > 0$ outgoing edges must be the first edge of a new trail. Thus the lower bound:

$$|P^*| \geq \sum_{v \in V} \max(0, \text{OUT}(v) - \text{IN}(v)).$$

The algorithm starts a new trail precisely when $\text{OUT}(v) > \text{IN}(v)$.

# Optimality

## Why is this decomposition minimal?

Let $P^*$ be **any** edge-disjoint trail cover of $G$

- For any node $v$, a trail passing through $v$ consumes exactly one incoming edge and one outgoing edge
- Therefore, at most $\text{IN}(v)$ outgoing edges can belong to trails that originated at some node other than $v$
- The remaining $(\text{OUT}(v) - \text{IN}(v)) > 0$ outgoing edges must be the first edge of a new trail. Thus the lower bound:

$$|P^*| \geq \sum_{v \in V} \max(0, \text{OUT}(v) - \text{IN}(v)).$$

The algorithm starts a new trail precisely when $\text{OUT}(v) > \text{IN}(v)$.

## Theorem

**PFS** *can be solved in* $O(||\mathcal{S}|| + |W| + |Y|)$ *time.*

# Full Example: Constructing $B'$

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.
From the TFS theorem, we obtain $X = X_1 \# X_2 \# X_3$:

- $X_1 = \texttt{aabaa}$
- $X_2 = \texttt{aaababbba}$
- $X_3 = \texttt{baab}$

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.
From the TFS theorem, we obtain $X = X_1 \# X_2 \# X_3$:

- $X_1 = \texttt{aabaa}$
- $X_2 = \texttt{aaababbba}$
- $X_3 = \texttt{baab}$

Extract and rank length-3 prefixes and suffixes:

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.
From the TFS theorem, we obtain $X = X_1 \# X_2 \# X_3$:

- $X_1 = \texttt{aabaa}$
- $X_2 = \texttt{aaababbba}$
- $X_3 = \texttt{baab}$

Extract and rank length-3 prefixes and suffixes:

- $X_1$: prefix aab, suffix baa
- $X_2$: prefix aaa, suffix bba
- $X_3$: prefix baa, suffix aab

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.
From the TFS theorem, we obtain $X = X_1 \# X_2 \# X_3$:

- $X_1 = \texttt{aabaa}$
- $X_2 = \texttt{aaababbba}$
- $X_3 = \texttt{baab}$

Extract and rank length-3 prefixes and suffixes:

- $X_1$: prefix aab, suffix baa
- $X_2$: prefix aaa, suffix bba
- $X_3$: prefix baa, suffix aab

Lexicographic ranks: $1 \to \texttt{aaa}$, $2 \to \texttt{aab}$, $3 \to \texttt{baa}$, $4 \to \texttt{bba}$.

# Full Example: Constructing $B'$

**Setup:** $W = \texttt{aabaaaababbbaab}$, $k = 4 \implies k - 1 = 3$.
From the TFS theorem, we obtain $X = X_1 \# X_2 \# X_3$:

- $X_1 = \texttt{aabaa}$
- $X_2 = \texttt{aaababbba}$
- $X_3 = \texttt{baab}$

Extract and rank length-3 prefixes and suffixes:

- $X_1$: prefix aab, suffix baa
- $X_2$: prefix aaa, suffix bba
- $X_3$: prefix baa, suffix aab

Lexicographic ranks: $1 \to \texttt{aaa}$, $2 \to \texttt{aab}$, $3 \to \texttt{baa}$, $4 \to \texttt{bba}$.

**Collection $B'$ of 2-letter strings**

$X_1 \implies 2 \cdot 3 \qquad X_2 \implies 1 \cdot 4 \qquad X_3 \implies 3 \cdot 2$

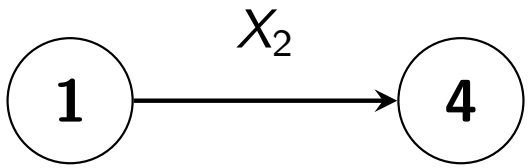
# Full Example: Graph Algorithm

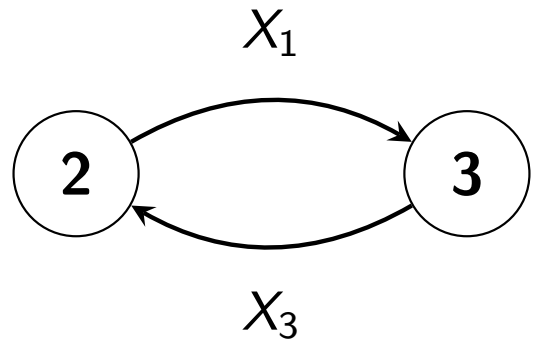
# Full Example: Graph Algorithm

**The Graph:**

# Full Example: Graph Algorithm

**The Graph:**



# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\mathrm{IN}(1) = 0, \mathrm{OUT}(1) = 1$

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\text{IN}(1) = 0, \text{OUT}(1) = 1$
  - Traverse $1 \to 4$. Add trail $p_1 = 14$ to $P$

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\text{IN}(1) = 0, \text{OUT}(1) = 1$
  - Traverse $1 \to 4$. Add trail $p_1 = 14$ to $P$
- **Second While Loop (Cycles):** Remaining edges $2 \to 3$ and $3 \to 2$

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\text{IN}(1) = 0, \text{OUT}(1) = 1$
  - Traverse $1 \to 4$. Add trail $p_1 = 14$ to $P$
- **Second While Loop (Cycles):** Remaining edges $2 \to 3$ and $3 \to 2$
  - Cycle $c = 2 \leftrightarrow 3$ does **not** intersect trail $p_1$

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\mathrm{IN}(1) = 0, \mathrm{OUT}(1) = 1$
  - Traverse $1 \to 4$. Add trail $p_1 = 14$ to $P$
- **Second While Loop (Cycles):** Remaining edges $2 \to 3$ and $3 \to 2$
  - Cycle $c = 2 \leftrightarrow 3$ does **not** intersect trail $p_1$
  - Break $c$ arbitrarily into new trail $p_2 = 232$ and add to $P$

# Full Example: Graph Algorithm

**The Graph:**


**Applying the Algorithm:**

- **First While Loop (Trails):** Node 1 has $\text{IN}(1) = 0, \text{OUT}(1) = 1$
  - Traverse $1 \to 4$. Add trail $p_1 = 14$ to $P$
- **Second While Loop (Cycles):** Remaining edges $2 \to 3$ and $3 \to 2$
  - Cycle $c = 2 \leftrightarrow 3$ does **not** intersect trail $p_1$
  - Break $c$ arbitrarily into new trail $p_2 = 232$ and add to $P$

$P = \{p_1, p_2\}$, where $p_1 = 14$ and $p_2 = 232$.

# Full Example: Constructing $Y$

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)
  - $\texttt{aabaa} \oplus \texttt{baab} = \texttt{aabaab}$

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)
  - $\texttt{aabaa} \oplus \texttt{baab} = \texttt{aabaab}$

**Final concatenation:**

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)
  - $\texttt{aabaa} \oplus \texttt{baab} = \texttt{aabaab}$

**Final concatenation:**

- Concatenate disjoint trails in $P$ using #.

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)
  - $\texttt{aabaa} \oplus \texttt{baab} = \texttt{aabaab}$

**Final concatenation:**

- Concatenate disjoint trails in $P$ using #.
- $Y = X_2 \,\#\, (X_1 \oplus X_3)$

# Full Example: Constructing $Y$

**Mapping trails back to string blocks:**

- Trail 14 $\implies$ $X_2 = \texttt{aaababbba}$
- Trail 232 $\implies$ $X_1 \oplus X_3$ (Overlapped)
  - $X_1$ (aabaa) and $X_3$ (baab) overlap by rank 3 (baa)
  - $\texttt{aabaa} \oplus \texttt{baab} = \texttt{aabaab}$

**Final concatenation:**

- Concatenate disjoint trails in $P$ using #.
- $Y = X_2 \,\#\, (X_1 \oplus X_3)$

**Output**

$Y = \texttt{aaababbba\#aabaab}$

# Reference

G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone, M. Sweering:
Combinatorial Algorithms for String Sanitization.
ACM Trans. Knowl. Discov. Data 15(1): 8:1-8:34 (2021)

# Solving MVR: Intuition

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

# Solving MVR: Intuition

## **MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

## Question

Can you think of a (simple) polynomial-time algorithm?

# Solving MVR: Intuition

## **MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

## Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$

# Solving MVR: Intuition

## **MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

## Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings
  - Edges are labeled from $\Sigma$

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings
  - Edges are labeled from $\Sigma$
- Spell $U$ in $G = (D, E)$ to find the single source node

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings
  - Edges are labeled from $\Sigma$
- Spell $U$ in $G = (D, E)$ to find the single source node
- Find valid sink nodes:

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings
  - Edges are labeled from $\Sigma$
- Spell $U$ in $G = (D, E)$ to find the single source node
- Find valid sink nodes:
  - Nodes $W$ such that $W \cdot V$ does not contain a string in $\mathcal{S}$

# Solving MVR: Intuition

**MVR** (Missing Value Replacement) problem

Given strings $U, V \in \Sigma^*$ and a finite set $\mathcal{S} \subset \Sigma^*$, compute a shortest string $X \in \Sigma^*$: $U$ is a prefix of $X$, $V$ is a suffix of $X$, and no $S \in \mathcal{S}$ occurs in $X$.

Question

Can you think of a (simple) polynomial-time algorithm?

- A string is called **dangerous** if it is a prefix of a string in $\mathcal{S}$
- Construct graph $G = (D, E)$, where $D$ is the set of dangerous strings
- Add labeled edges telling us how to *safely* extend dangerous strings
  - Edges are labeled from $\Sigma$
- Spell $U$ in $G = (D, E)$ to find the single source node
- Find valid sink nodes:
  - Nodes $W$ such that $W \cdot V$ does not contain a string in $\mathcal{S}$
- BFS (from source to sinks) to find a shortest path

# The Graph

# The Graph

- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$

# The Graph

- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:

# The Graph

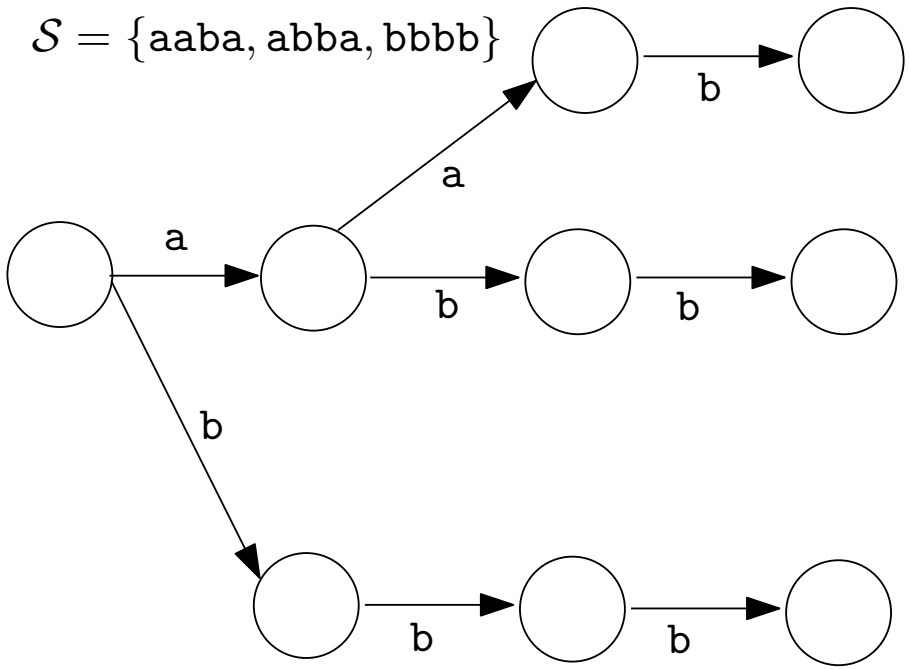
- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:
  - $v_1\alpha$ is not in $\mathcal{S}$

# The Graph

- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:
  - $v_1\alpha$ is not in $\mathcal{S}$
  - $v_2$ is the longest dangerous suffix of $v_1\alpha$

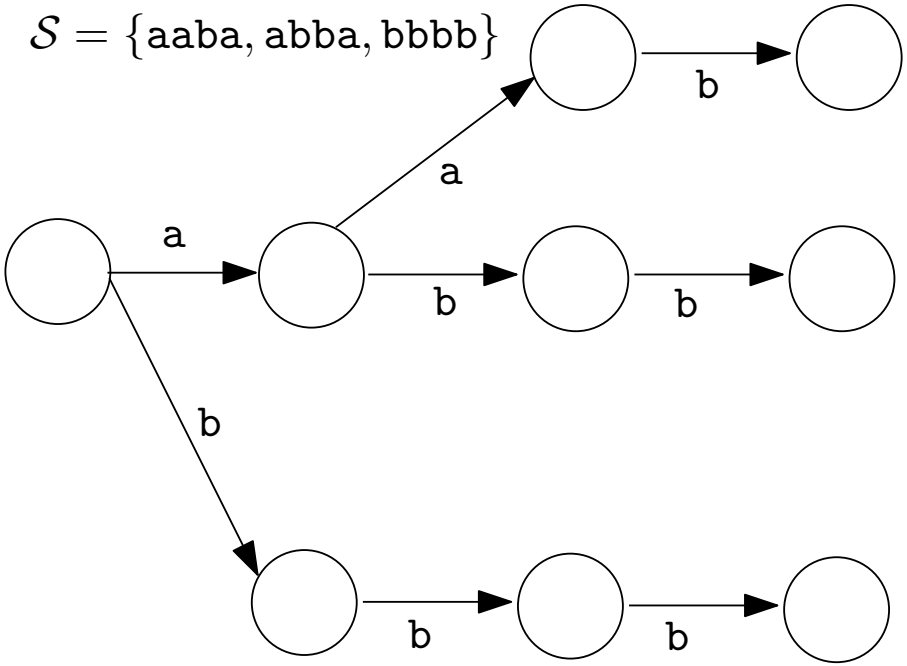
# The Graph

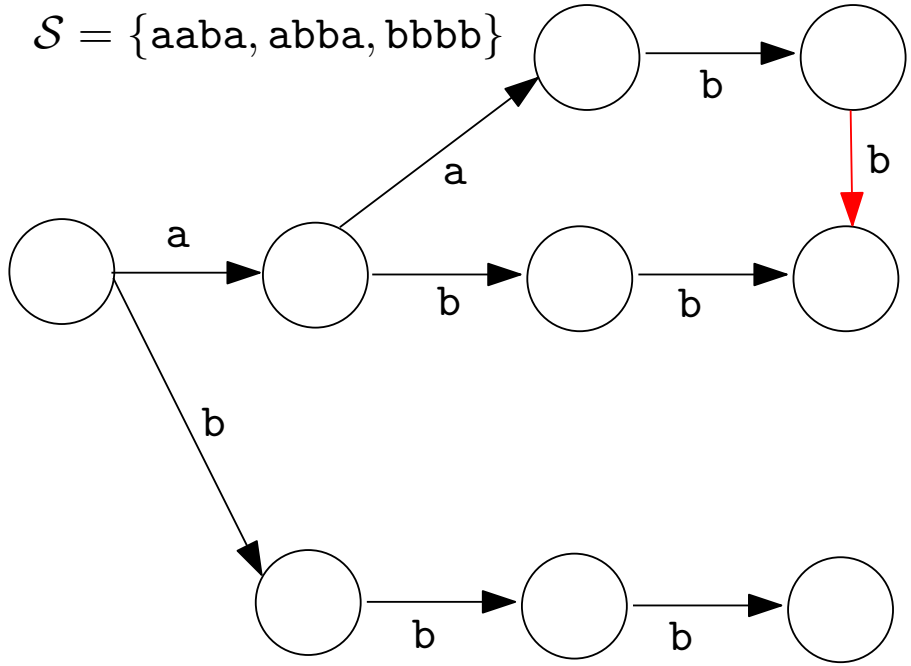
- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:
  - $v_1\alpha$ is not in $\mathcal{S}$
  - $v_2$ is the longest dangerous suffix of $v_1\alpha$



# The Graph

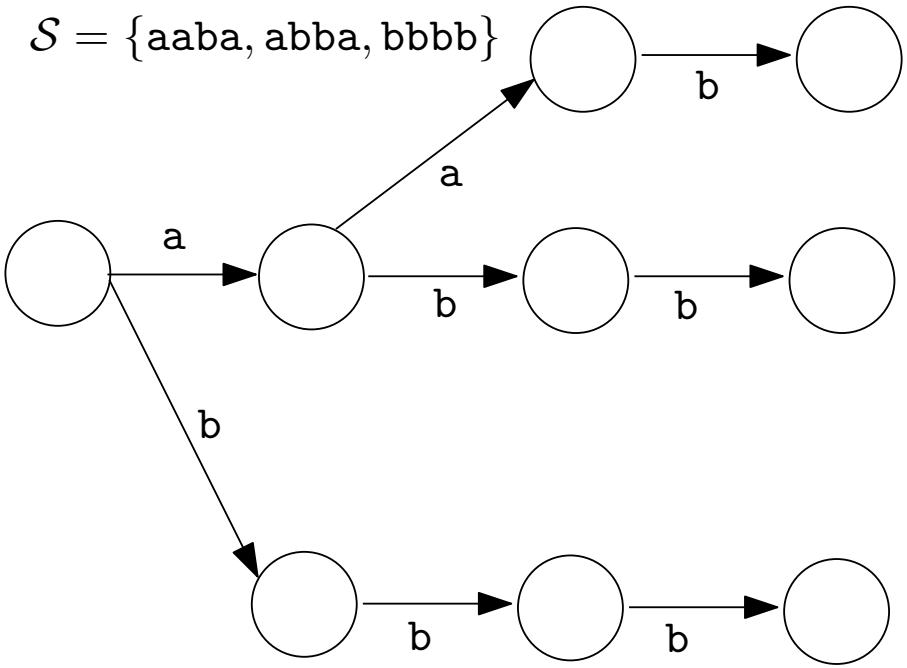
- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:
  - $v_1\alpha$ is not in $\mathcal{S}$
  - $v_2$ is the longest dangerous suffix of $v_1\alpha$

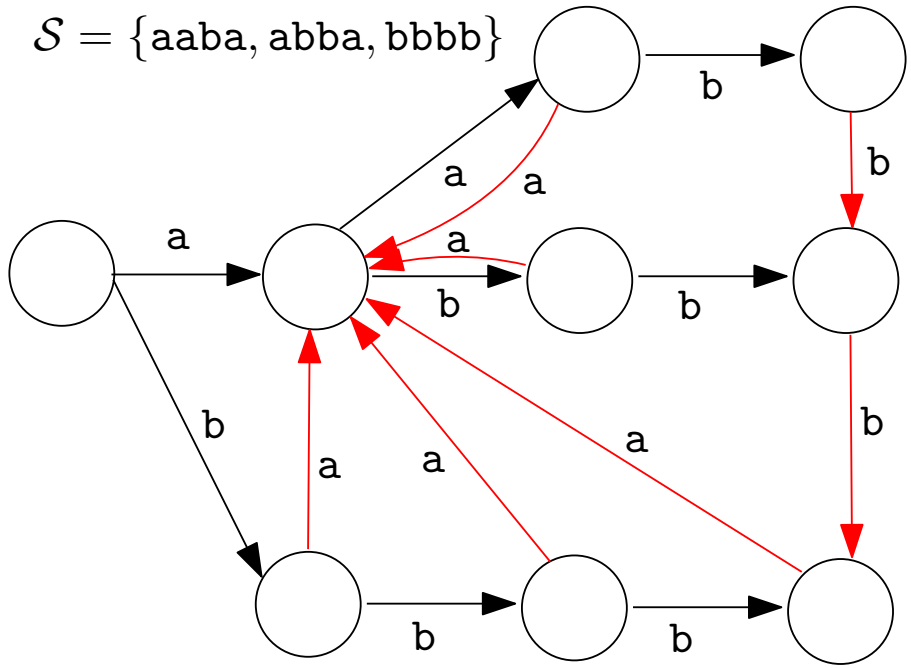


# The Graph

- Construct the trie of the longest proper prefixes of the strings in $\mathcal{S}$
- Add the failure labeled edge $v_1 \xrightarrow{\alpha} v_2$ if and only if:
  - $v_1\alpha$ is not in $\mathcal{S}$
  - $v_2$ is the longest dangerous suffix of $v_1\alpha$




**Lemma**

*$G = (D, E)$ can be constructed in $O(||\mathcal{S}|| \cdot |\Sigma|)$ time.*

# Finding the sinks and the source

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$

# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$
  - For every such suffix $P$ such that $QP \in \mathcal{S}$, set $Q$ as a non-sink

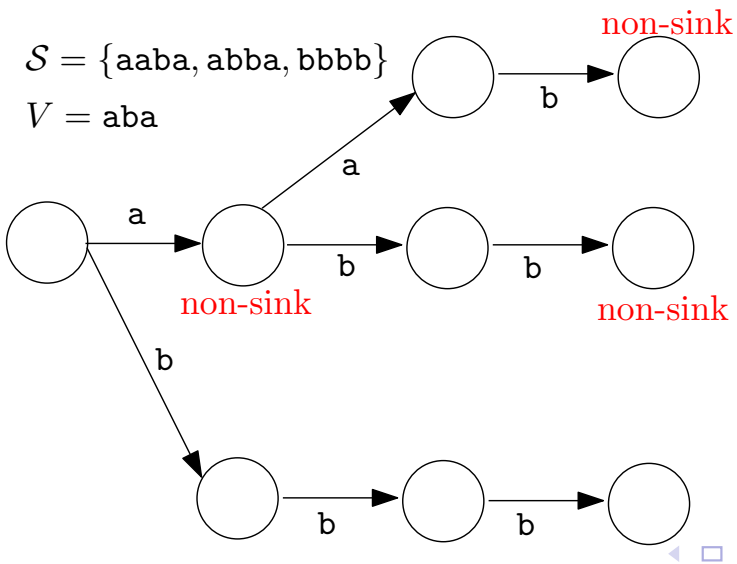
# Finding the sinks and the source

A **sink** is a node $W$ such that $WV$ does not contain a string in $\mathcal{S}$.

## Key Idea

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$
  - For every such suffix $P$ such that $QP \in \mathcal{S}$, set $Q$ as a non-sink



# Finding the sinks and the source

A **sink** is a node $W$: $WV$ does not contain a string in $\mathcal{S}$.

**Key Idea**

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$
  - For every such suffix $P$ such that $QP \in \mathcal{S}$, set $Q$ as a non-sink

# Finding the sinks and the source

A **sink** is a node $W$: $WV$ does not contain a string in $\mathcal{S}$.

**Key Idea**

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$
  - For every such suffix $P$ such that $QP \in \mathcal{S}$, set $Q$ as a non-sink

Finding the source is trivial: spell $U$ in $G$.

# Finding the sinks and the source

A **sink** is a node $W$: $WV$ does not contain a string in $\mathcal{S}$.

**Key Idea**

Compute the non-sink nodes instead!

- Construct the suffix tree of $\mathcal{S} \cup \{V\}$
- For each prefix $P = V[0..i]$ of $V$:
  - Find all strings in $\mathcal{S}$ with suffix $P$
  - For every such suffix $P$ such that $QP \in \mathcal{S}$, set $Q$ as a non-sink

Finding the source is trivial: spell $U$ in $G$.

**Lemma**

*Finding the source and all sinks takes $O(|U| + |V| + ||\mathcal{S}||)$ time.*

# Constructing $X$

# Constructing $X$

We have two cases:

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink
  - $\implies X = U[0 \,..\, i-1]V$

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink
  - $\implies X = U[0 \,..\, i-1]V$

Non-Overlap Case:

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink
  - $\implies X = U[0 \,..\, i-1]V$

Non-Overlap Case:

- BFS from the source; let $h$ be a shortest string to a sink

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink
  - $\implies X = U[0 \,..\, i-1]V$

Non-Overlap Case:

- BFS from the source; let $h$ be a shortest string to a sink
- $\implies X = UhV$

# Constructing $X$

We have two cases:

- $|X| < |U| + |V|$ ($U$ and $V$ overlap)
- $|X| \geq |U| + |V|$ ($U$ and $V$ do not overlap)

Overlap Case:

- Find the length of every possible overlap (using the suffix tree)
- Spell $U$ letter by letter in $G$; if after following $i > 0$ edges:
  - $|U| - i$ is an overlap **and** we have arrived at a sink
  - $\implies X = U[0 \,..\, i-1]V$

Non-Overlap Case:

- BFS from the source; let $h$ be a shortest string to a sink
- $\implies X = UhV$

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

# Reference

G. Bernardini, C. Liu, G. Loukides, A. Marchetti-Spaccamela, S. P. Pissis, L. Stougie, M. Sweering:
Missing value replacement in strings and applications.
Data Min. Knowl. Discov. 39(2): 12 (2025)

# Outline

# Introduction to $z$-Anonymity

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

- **Generalization:** Replace values with ranges (e.g., Age 24 $\rightarrow$ 20s).

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

- **Generalization:** Replace values with ranges (e.g., Age 24 $\rightarrow$ 20s).
- **Suppression:** Mask values (e.g., Zip 10024 $\rightarrow$ 1002*).

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

- **Generalization:** Replace values with ranges (e.g., Age 24 $\rightarrow$ 20s).
- **Suppression:** Mask values (e.g., Zip 10024 $\rightarrow$ 1002*).

**Example: 2-Anonymized Medical Table ($z = 2$)**

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

- **Generalization:** Replace values with ranges (e.g., Age 24 $\rightarrow$ 20s).
- **Suppression:** Mask values (e.g., Zip 10024 $\rightarrow$ 1002*).

**Example: 2-Anonymized Medical Table ($z = 2$)**

| Name | Age | Zip Code | Condition (Sensitive) |
|---|---|---|---|
| Paul | 25 | 1002A | Heart Disease |
| Helen | 23 | 1002B | Viral Infection |
| Harry | 42 | 1003B | Cancer |
| Anna | 49 | 1003A | Heart Disease |

[2]Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

# Introduction to $z$-Anonymity

A dataset is said to have $z$-**anonymity** if the information for any given individual cannot be distinguished from at least $(z-1)$ other individuals.[2]

**Mechanism:** It prevents re-identification by altering *quasi-identifiers*:

- **Generalization:** Replace values with ranges (e.g., Age 24 $\rightarrow$ 20s).
- **Suppression:** Mask values (e.g., Zip 10024 $\rightarrow$ 1002*).

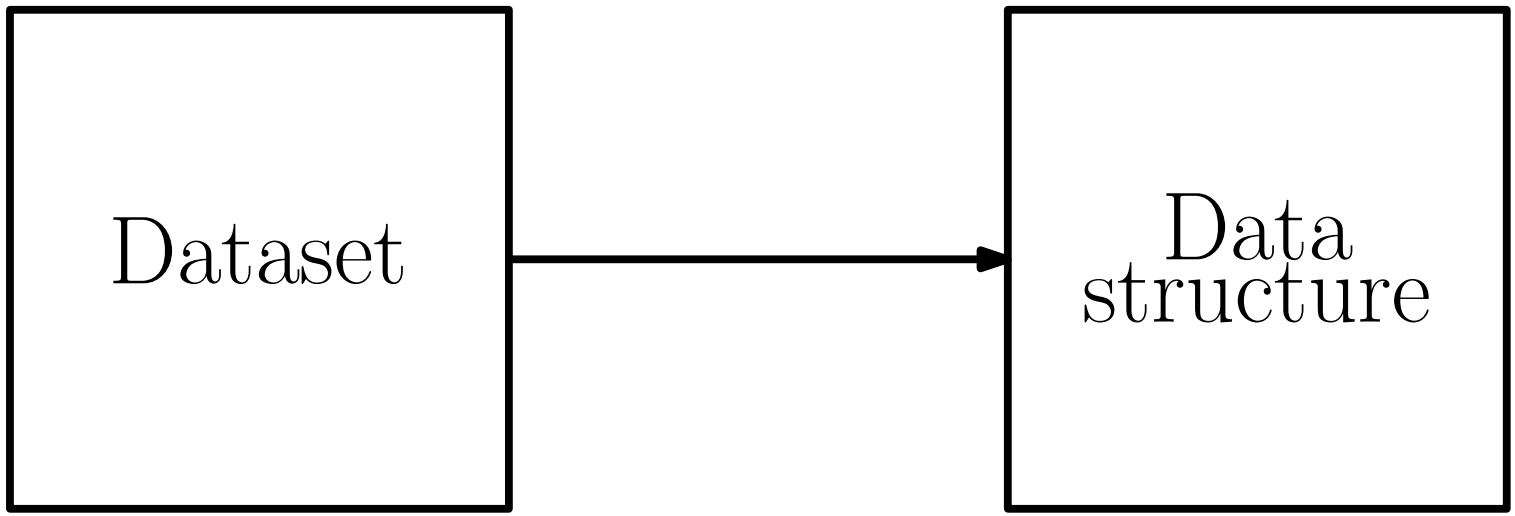
**Example: 2-Anonymized Medical Table ($z = 2$)**

| Name | Age | Zip Code | Condition (Sensitive) |
|---|---|---|---|
| Paul | 25 | 1002A | Heart Disease |
| Helen | 23 | 1002B | Viral Infection |
| Harry | 42 | 1003B | Cancer |
| Anna | 49 | 1003A | Heart Disease |

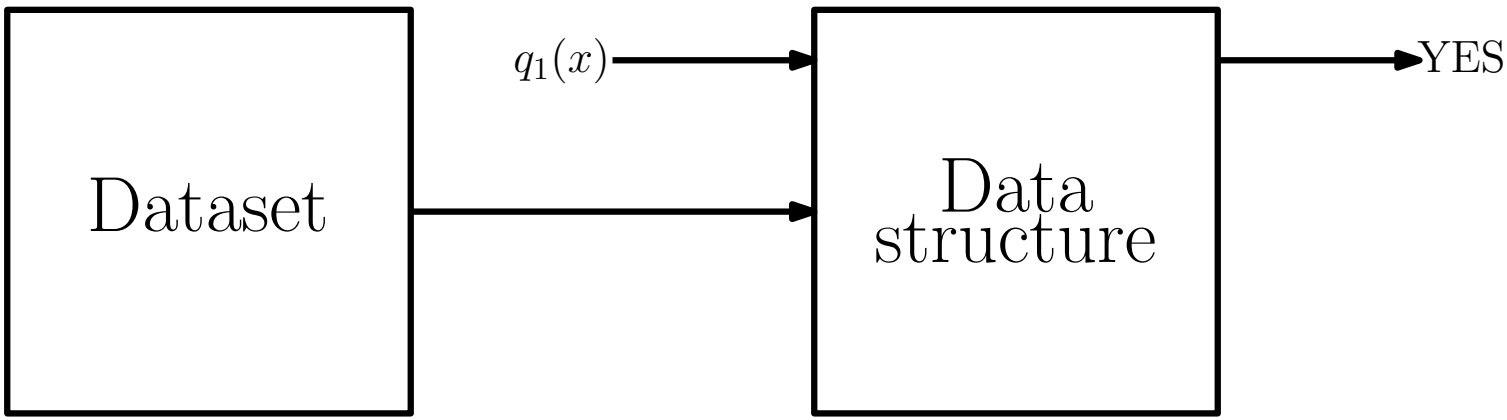
| Name | Age | Zip Code | Condition (Sensitive) |
|---|---|---|---|
| * | 20s | 1002* | Heart Disease |
| * | 20s | 1002* | Viral Infection |
| * | 40s | 1003* | Cancer |
| * | 40s | 1003* | Heart Disease |

[2] Samarati and Sweeney, Harvard Data Privacy Lab, 1998; Dalenius, Journal of Official Statistics, 1986

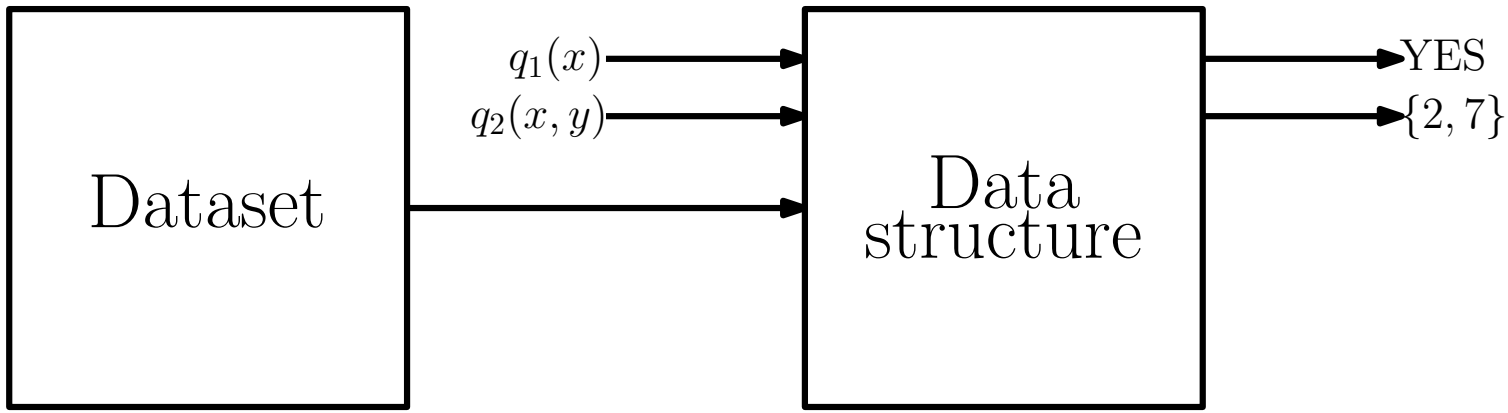
# $z$-Anonymous Data Structures



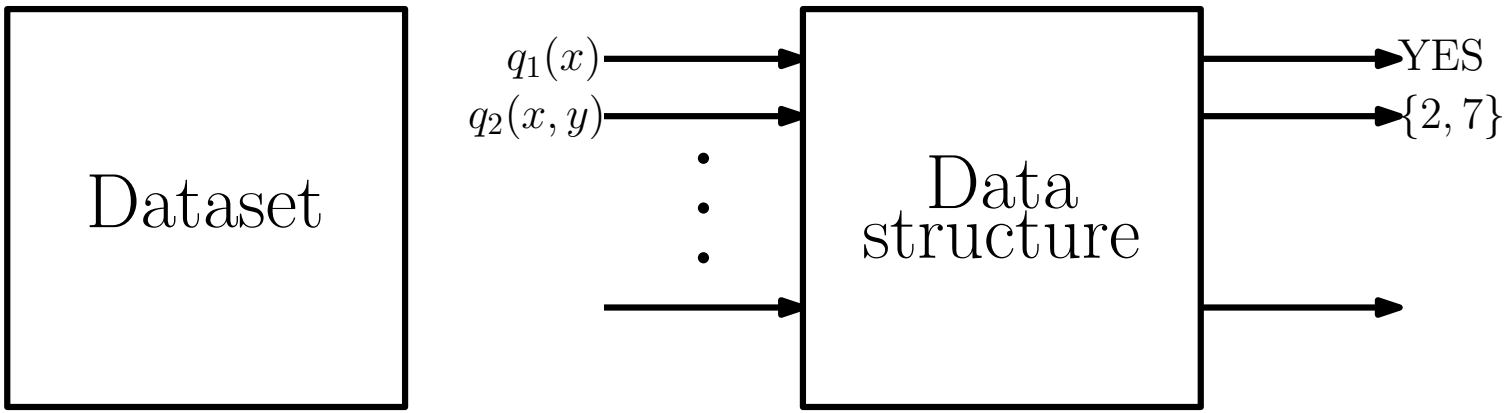
# $z$-Anonymous Data Structures



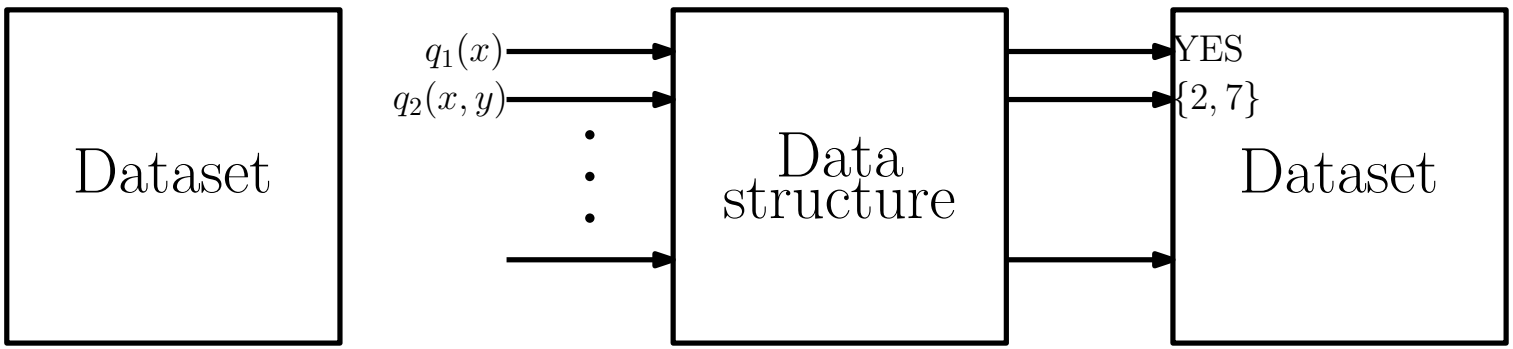
# $z$-Anonymous Data Structures



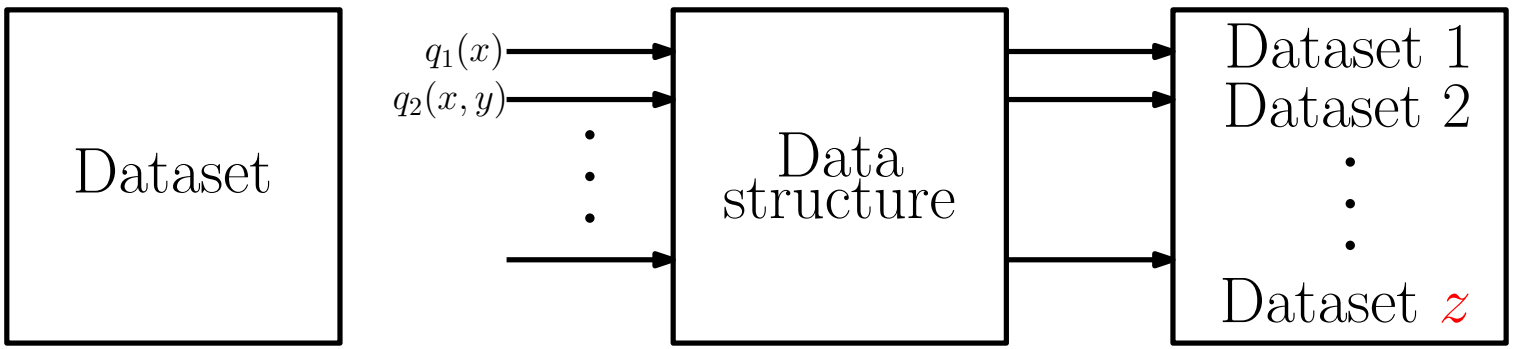
# $z$-Anonymous Data Structures



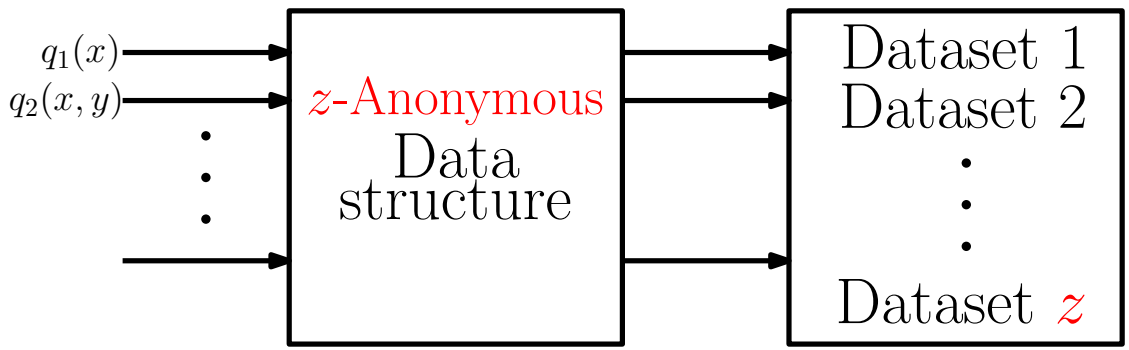
# $z$-Anonymous Data Structures



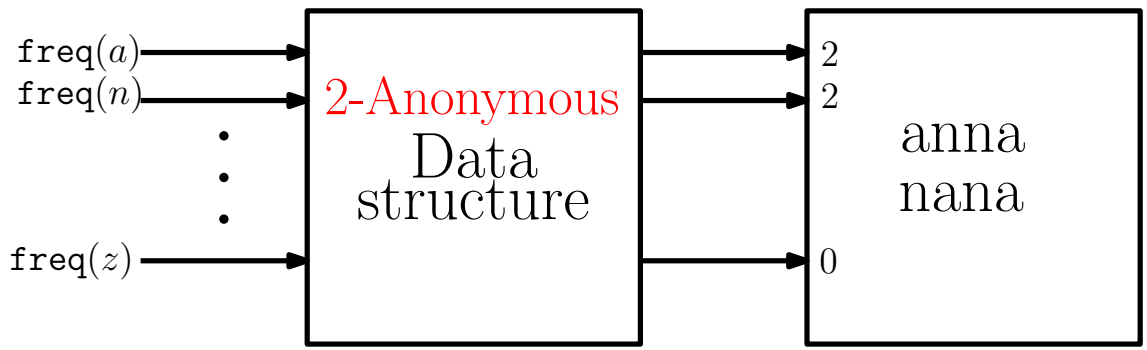
# $z$-Anonymous Data Structures



# $z$-Anonymous Data Structures



# $z$-Anonymous Data Structures



# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

# Strings to Graphs, and Back

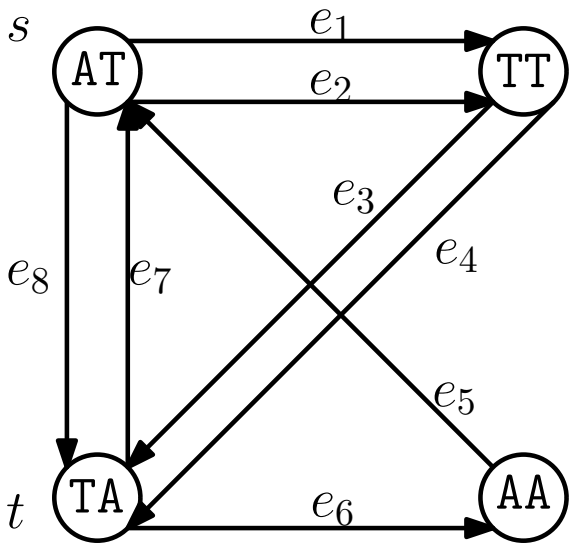
Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

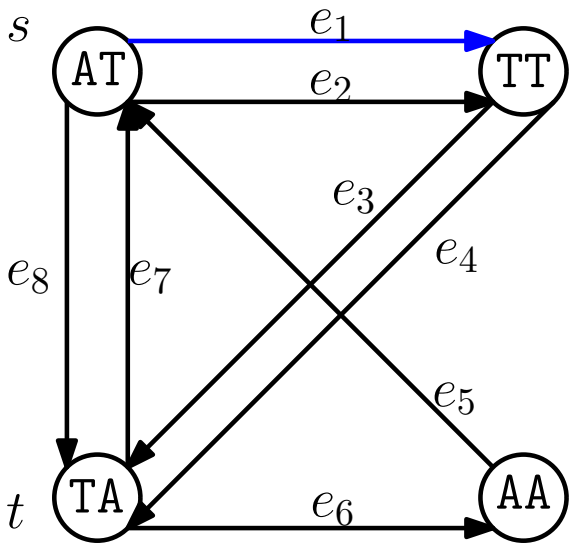
For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.



# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

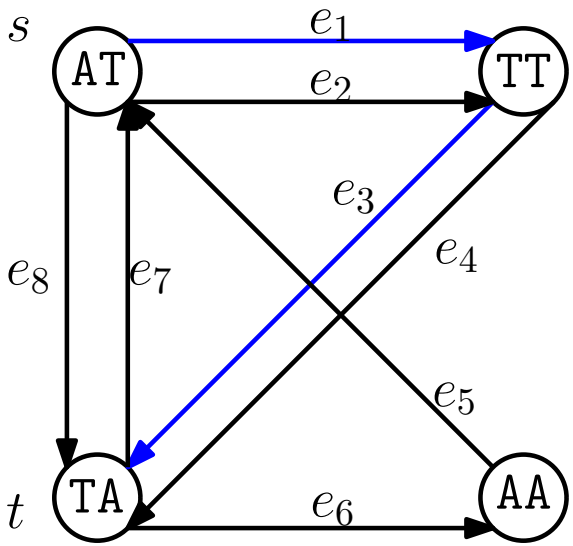



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

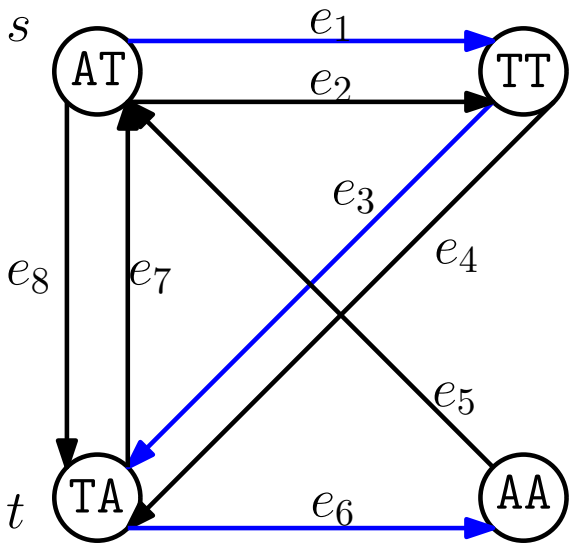



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

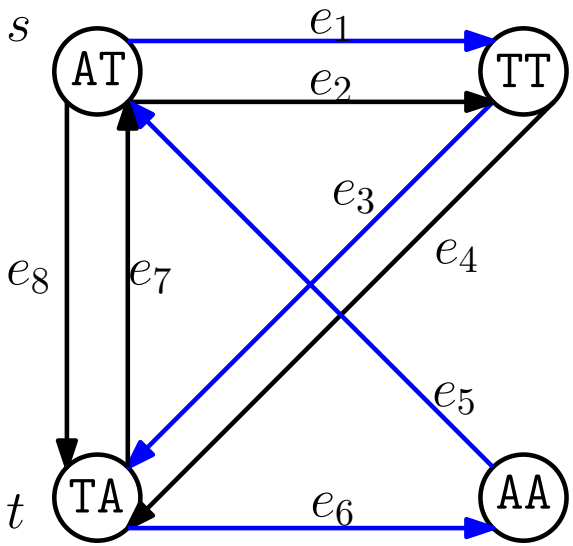



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

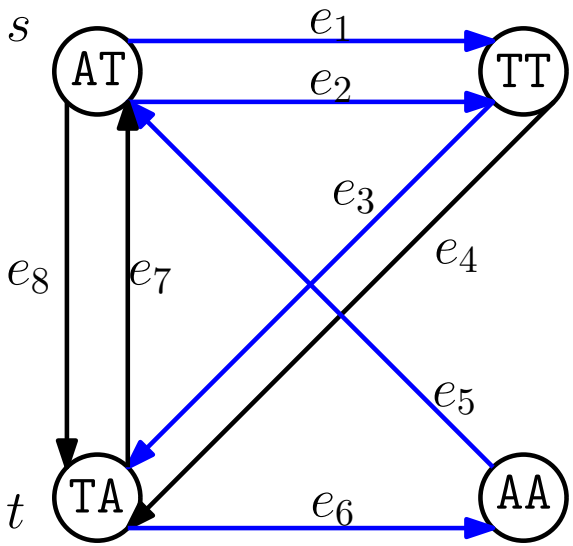



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

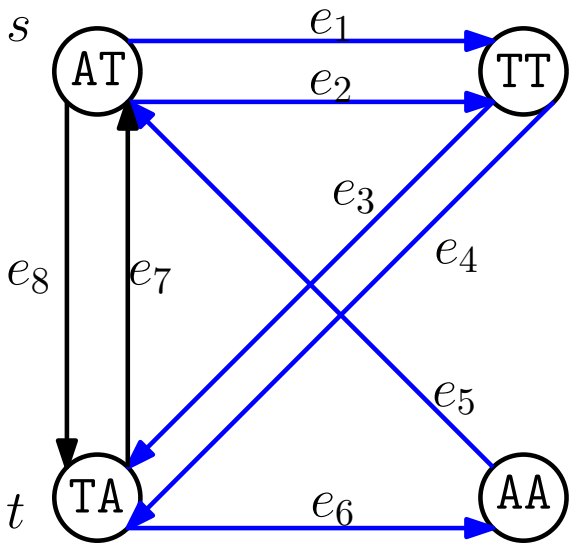



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

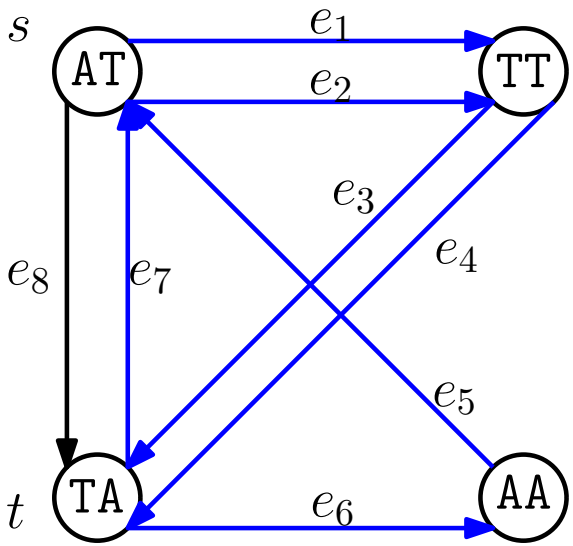



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

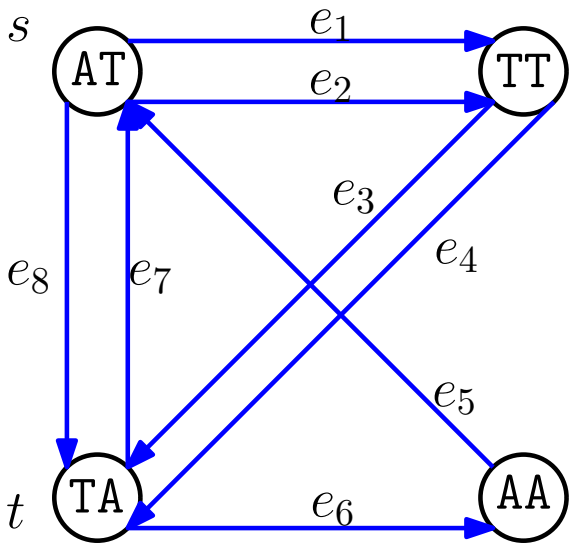



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

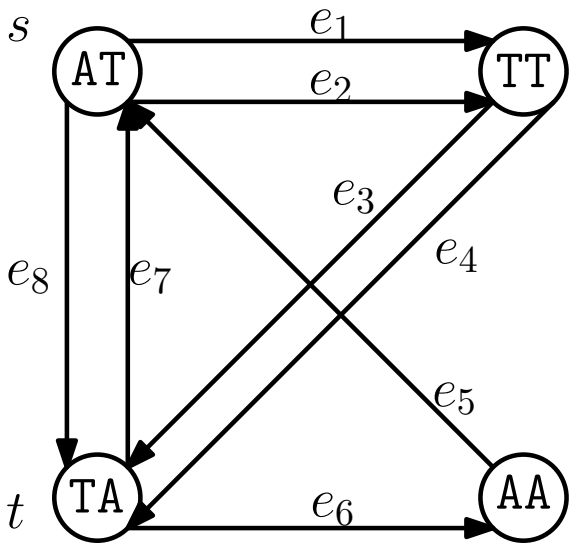



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.

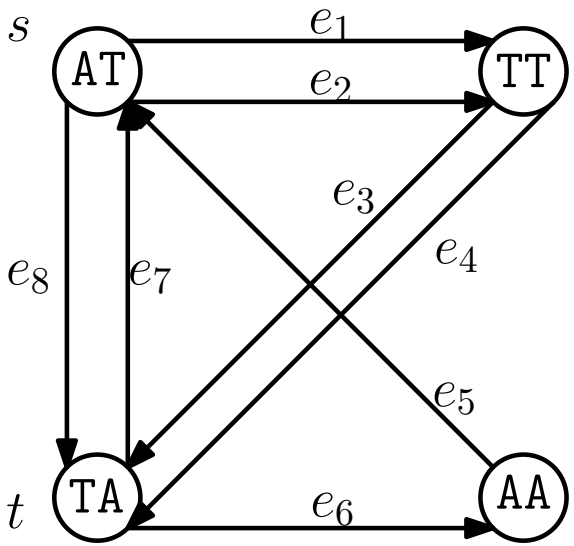



| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |
| $e_1e_3e_7e_2e_4e_6e_5e_8$ | ATTATTAATA |
| $e_1e_3e_7e_8e_6e_5e_2e_4$ | ATTATAATTA |
| $e_8e_6e_5e_1e_3e_7e_2e_4$ | ATAATTATTA |
| $e_8e_7e_1e_3e_6e_5e_2e_4$ | ATATTAATTA |
| $e_1e_3e_6e_5e_8e_7e_2e_4$ | ATTAATATTA |

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.



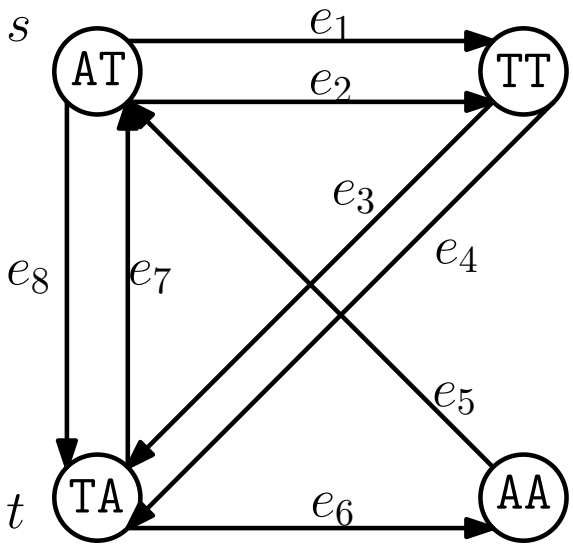

| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |
| $e_1e_3e_7e_2e_4e_6e_5e_8$ | ATTATTAATA |
| $e_1e_3e_7e_8e_6e_5e_2e_4$ | ATTATAATTA |
| $e_8e_6e_5e_1e_3e_7e_2e_4$ | ATAATTATTA |
| $e_8e_7e_1e_3e_6e_5e_2e_4$ | ATATTAATTA |
| $e_1e_3e_6e_5e_8e_7e_2e_4$ | ATTAATATTA |

The above de Bruijn graph is 6-anonymous for substrings of length $d = 3$.

# Strings to Graphs, and Back

Let $S = \texttt{ATTAATTATA}$ be a private string.

For $d = 3$, we get $\texttt{ATT}, \texttt{TTA}, \texttt{TAA}, \texttt{AAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAT}, \texttt{ATA}$.




| Eulerian trails | String reconstructions |
|---|---|
| $e_1e_3e_6e_5e_2e_4e_7e_8$ | ATTAATTATA |
| $e_1e_3e_7e_2e_4e_6e_5e_8$ | ATTATTAATA |
| $e_1e_3e_7e_8e_6e_5e_2e_4$ | ATTATAATTA |
| $e_8e_6e_5e_1e_3e_7e_2e_4$ | ATAATTATTA |
| $e_8e_7e_1e_3e_6e_5e_2e_4$ | ATATTAATTA |
| $e_1e_3e_6e_5e_8e_7e_2e_4$ | ATTAATATTA |

The above de Bruijn graph is 6-anonymous for substrings of length $d = 3$.

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$ such that the graph is $z$-anonymous in $O(|S|^\omega \log d)$ or $O((|S| + z) \log d)$ time.*

# Combinatorial Insights

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

**Lemma**

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

**Lemma**

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

**Question**

Can you think of a proof?

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

## Lemma

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

## Question

Can you think of a proof?

## Proof.

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

**Lemma**

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

**Question**

Can you think of a proof?

**Proof.**

Let $E_d(S)$ be the set of strings $d$-equivalent to $S$, where $|E_d(S)| = \alpha_d(S)$.

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

## Lemma

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

## Question

Can you think of a proof?

## Proof.

Let $E_d(S)$ be the set of strings $d$-equivalent to $S$, where $|E_d(S)| = \alpha_d(S)$. By definition, $(d+1)$-equivalence requires equal multisets of length-$k$ substrings for all $k \in [1, d+1]$.

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

## Lemma

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

## Question

Can you think of a proof?

## Proof.

Let $E_d(S)$ be the set of strings $d$-equivalent to $S$, where $|E_d(S)| = \alpha_d(S)$. By definition, $(d+1)$-equivalence requires equal multisets of length-$k$ substrings for all $k \in [1, d+1]$. This trivially includes all $k \in [1, d]$, meaning $(d+1)$-equivalence strictly implies $d$-equivalence.

# Combinatorial Insights

Two strings $X$ and $Y$ are called $d$-**equivalent**, for an integer $d \geq 1$, if and only if for every $k \in [1, d]$, we have the following **multiset** equality:

$$\{ X[i \,..\, i+k-1] \mid 0 \leq i \leq |X|-k \} = \{ Y[i \,..\, i+k-1] \mid 0 \leq i \leq |Y|-k \}.$$

**Lemma**

*For every string $S$, the number $\alpha_d(S)$ of strings that are $d$-equivalent to $S$ is non-increasing as $d$ increases.*

**Question**

Can you think of a proof?

**Proof.**

Let $E_d(S)$ be the set of strings $d$-equivalent to $S$, where $|E_d(S)| = \alpha_d(S)$. By definition, $(d+1)$-equivalence requires equal multisets of length-$k$ substrings for all $k \in [1, d+1]$. This trivially includes all $k \in [1, d]$, meaning $(d+1)$-equivalence strictly implies $d$-equivalence. Therefore, $E_{d+1}(S) \subseteq E_d(S)$, which directly yields $\alpha_{d+1}(S) \leq \alpha_d(S)$. □

# The Algorithm

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:
  - $O(|S|^\omega)$ time using the BEST theorem, $\omega$ is the matrix mult. exp.[3]

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:
  - $O(|S|^\omega)$ time using the BEST theorem, $\omega$ is the matrix mult. exp.[3]
  - $O(|S| + z)$ time using enumeration[4]

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:
  - $O(|S|^\omega)$ time using the BEST theorem, $\omega$ is the matrix mult. exp.[3]
  - $O(|S| + z)$ time using enumeration[4]
- Check if $\alpha_d(S) \geq z$

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:
  - $O(|S|^\omega)$ time using the BEST theorem, $\omega$ is the matrix mult. exp.[3]
  - $O(|S| + z)$ time using enumeration[4]
- Check if $\alpha_d(S) \geq z$

For the largest $d$ with $\alpha_d(S) \geq z$, release the graph $G_d$.

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# The Algorithm

We can binary search, counting the Eulerian trails at every iteration

For every $d$ implied by exponential search:

- Construct the graph $G_d$ of order $d$ (using the suffix tree)
- Count $\alpha_d(S)$ by counting the Eulerian trails in $G_d$:
  - $O(|S|^\omega)$ time using the BEST theorem, $\omega$ is the matrix mult. exp.[3]
  - $O(|S| + z)$ time using enumeration[4]
- Check if $\alpha_d(S) \geq z$

For the largest $d$ with $\alpha_d(S) \geq z$, release the graph $G_d$.

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$ such that the graph is $z$-anonymous in $O(|S|^\omega \log d)$ or $O((|S| + z) \log d)$ time.*

[3] van Aardenne-Ehrenfest, de Bruijn, Classic Papers in Combinatorics, 1951

[4] Bals, Tinca, Pissis, arXiv, 2026

# Application

# Application

Since $G_d$ is $z$-anonymous:

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$
- This tree answers counting queries **exactly** for any length $\leq d$

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$
- This tree answers counting queries **exactly** for any length $\leq d$

**Theorem**

*For any string $S$ of length $n$ and any privacy threshold $z > 0$, there exists a maximal $d \in [0, n)$ and a $z$-anonymous data structure $D$ such that:*

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$
- This tree answers counting queries **exactly** for any length $\leq d$

**Theorem**

*For any string $S$ of length $n$ and any privacy threshold $z > 0$, there exists a maximal $d \in [0, n)$ and a $z$-anonymous data structure $D$ such that:*

- *the size of $D$ is $O(n)$;*

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$
- This tree answers counting queries **exactly** for any length $\leq d$

**Theorem**

*For any string $S$ of length $n$ and any privacy threshold $z > 0$, there exists a maximal $d \in [0, n)$ and a $z$-anonymous data structure $D$ such that:*

- *the size of $D$ is $O(n)$;*
- *$D$ can be constructed in $O(n^\omega \log d)$ or $O((n+z)\log d)$ time;*

# Application

Since $G_d$ is $z$-anonymous:

- Sample a single Eulerian trail
- This corresponds to a string $S'$
- The suffix tree of $S'$ truncated at string depth $d$ is $z$-anonymous
- Cut the suffix tree at string depth $d$
- This tree answers counting queries **exactly** for any length $\leq d$

**Theorem**

*For any string $S$ of length $n$ and any privacy threshold $z > 0$, there exists a maximal $d \in [0, n)$ and a $z$-anonymous data structure $D$ such that:*

- *the size of $D$ is $O(n)$;*
- *$D$ can be constructed in $O(n^\omega \log d)$ or $O((n+z)\log d)$ time;*
- *$D$ answers counting queries exactly, for any pattern of length $m \leq d$, in the optimal $O(m)$ time.*

# Reference

G. Bernardini, H. Chen, G. Fici, G. Loukides, S. P. Pissis:
Reverse-Safe Text Indexing.
ACM J. Exp. Algorithmics 26: 1.10:1-1.10:26 (2021)

# Outline

# Open Problem 1

[5] Bernardini, Marchetti-Spaccamela, Pissis, Stougie, Sweering, CPM 2021

# Open Problem 1

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

[5] Bernardini, Marchetti-Spaccamela, Pissis, Stougie, Sweering, CPM 2021

# Open Problem 1

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we improve this theorem?

[5] Bernardini, Marchetti-Spaccamela, Pissis, Stougie, Sweering, CPM 2021

# Open Problem 1

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we improve this theorem?

- The size of $G = (D, E)$ is in $\Omega(||\mathcal{S}|| \cdot |\Sigma|)$ in the worst case[5]

[5] Bernardini, Marchetti-Spaccamela, Pissis, Stougie, Sweering, CPM 2021

# Open Problem 1

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we improve this theorem?

- The size of $G = (D, E)$ is in $\Omega(||\mathcal{S}|| \cdot |\Sigma|)$ in the worst case[5]
- We might need a completely different approach!

[5] Bernardini, Marchetti-Spaccamela, Pissis, Stougie, Sweering, CPM 2021

# Open Problem 2

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

# Open Problem 2

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we design a data structure for processing different $U, V$?

# Open Problem 2

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we design a data structure for processing different $U, V$?

- Motivation: The above theorem replaces a single #

# Open Problem 2

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we design a data structure for processing different $U, V$?

- Motivation: The above theorem replaces a single #
- The antidictionary can be static for a single string!

# Open Problem 2

**Theorem**

**MVR** *can be solved in* $O(|U| + |V| + ||\mathcal{S}|| \cdot |\Sigma|)$ *time.*

Can we design a data structure for processing different $U, V$?

- Motivation: The above theorem replaces a single #
- The antidictionary can be static for a single string!
- For every #, we may have different $U, V$

# Open Problem 3

# Open Problem 3

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

# Open Problem 3

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

Can we truncate the suffix tree in a non-uniform way?

# Open Problem 3

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

Can we truncate the suffix tree in a non-uniform way?

- For maximizing $d$, it suffices to cut everything at depth $d$

# Open Problem 3

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

Can we truncate the suffix tree in a non-uniform way?

- For maximizing $d$, it suffices to cut everything at depth $d$
- Can we really maximize the number of stored answers?

# Open Problem 4

# Open Problem 4

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

# Open Problem 4

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

Can we assess $G_d = (V, E)$ faster than $O(|V|^{\omega})$ or $O(|E| + z)$?

# Open Problem 4

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^\omega \log d)$ or $O((|S| + z) \log d)$ time.*

Can we assess $G_d = (V, E)$ faster than $O(|V|^\omega)$ or $O(|E| + z)$?

- $O(|V|^\omega)$ counts

# Open Problem 4

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^{\omega} \log d)$ or $O((|S| + z) \log d)$ time.*

Can we assess $G_d = (V, E)$ faster than $O(|V|^{\omega})$ or $O(|E| + z)$?

- $O(|V|^{\omega})$ counts
- $O(|E| + z)$ enumerates

# Open Problem 4

**Theorem**

*Given $S$ and a privacy threshold $z > 0$, we can find the maximal $d$, such that the graph is $z$-anonymous in $O(|S|^\omega \log d)$ or $O((|S| + z) \log d)$ time.*

Can we assess $G_d = (V, E)$ faster than $O(|V|^\omega)$ or $O(|E| + z)$?

- $O(|V|^\omega)$ counts
- $O(|E| + z)$ enumerates
- However, what we need is to assess: $\alpha_d(S) \geq z$?