

Set Parameterized Matching via Multi-Layer Hashing

Moshe Lewenstein  

Bar-Ilan University, Ramat Gan, Israel

Ely Porat  

Bar-Ilan University, Ramat Gan, Israel

Abstract

We study the *set parameterized matching* problem, a generalization of the classical parameterized matching problem introduced by Baker [7, 8]. In set parameterized matching, both the pattern and text are sequences where each position contains a *set* of characters rather than a single character. Two set-strings parameterized match if there exists a bijection between their alphabets that maps one to the other set-wise. Boussidan [12] introduced this problem for the case of equal-length set-strings. We present a randomized algorithm running in $O(N + M)$ time with high probability, where N is the text size and M is the pattern size. Our approach employs a novel three-layer hashing scheme based on Karp-Rabin fingerprinting that addresses the challenges of (1) the size blowup in representations of the problem, (2) set-to-set matching, and (3) the dynamic nature of encodings of text substrings during pattern scanning.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Pattern matching; Mathematics of computing $\rightarrow$ Probabilistic algorithms

Keywords and phrases Set Parameterized Matching, Pattern Matching, Randomized Algorithms, Hashing, Parameterized Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.36

Funding *Moshe Lewenstein:* Work on this paper was supported by ISF grant # 2760/25 awarded by the Israel Science Foundation.

1 Introduction

The parameterized pattern matching problem, introduced by Baker [6, 7, 8], seeks all occurrences of a pattern $P[1..m]$ in a text $T[1..n]$ where an occurrence is defined not by exact character matches but by consistent character renaming. Formally, the pattern occurs at position i if there exists a bijection π from the pattern alphabet to the text alphabet such that $T[i + j - 1] = \pi(P[j])$ for all $j \in [1..m]$. This problem has found numerous applications in software maintenance [6], plagiarism detection, computational biology [25], and beyond [24].

1.1 Prior Work

The study of *parameterized matching* was pioneered by Baker [6, 7], who introduced the concept to identify duplicated code in large software systems. The core innovation was the “predecessor string” (or *prev*) representation, which converts a string into an integer sequence where each character is replaced by the offset to its previous occurrence. A key observation is that two strings parameterized match if and only if their *prev* transformations are identical. This insight allowed for efficient matching regardless of specific variable names, provided the usage patterns were consistent. This discovery enabled $O(n \log \min(m, |\Sigma_P|))$ -time algorithms [1] and led to the development of parameterized suffix trees [8] and suffix arrays [14]. Subsequent research has explored the theoretical boundaries and practical variations of the problem:



© Moshe Lewenstein and Ely Porat;

licensed under Creative Commons License CC-BY 4.0

37th Annual Symposium on Combinatorial Pattern Matching (CPM 2026).

Editors: Philip Bille and Nicola Prezza; Article No. 36; pp. 36:1–36:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- **Approximate matching:** Significant work has been done on matching with k mismatches [4, 18], edit distance [9, 5], lcs [20] and $\delta\gamma$ -distances [22].
- **Compressed matching:** Solutions have been developed for searching in compressed text [17], run-length encoded strings [3], and other compression schemes [10, 21].
- **Higher dimensions and Structures:** The problem has been extended beyond linear strings to matching on trees [2] and two-dimensional grids [13].
- **Succinct data structures:** Recent developments include space-efficient dictionaries [15, 16] and the parameterized Burrows-Wheeler Transform (pBWT) [16].

For a comprehensive overview of the field’s evolution, Lewenstein [23] and Mendivelso et al. [24] provide a detailed historical account of these problems and their various applications.

1.2 Set Parameterized Matching

While classical parameterized matching handles sequences of single characters, many real-world scenarios involve ambiguity or multiple valid interpretations at a single position. Boussidan [12] in his PhD thesis (see also [11]) was concerned about comparing 18th century French plays, where the characters of the play are important, but each scene has multiple characters within. To address this, Boussidan extended the model of parameterized matching to *set parameterized comparison*, where for two equal-length strings, each position in both contains a *set* of characters, and the question is whether they parameterize match. We call these strings containing sets, *set-strings*. Boussidan’s original algorithm relied on bipartite matching. This approach is computationally expensive, requiring $O(m \cdot |\Sigma|^{2.5})$ time for comparison alone. In this paper, we consider the full pattern matching version of the problem:

► **Problem 1 (Set Parameterized Matching).** Given a text $\mathcal{T} = \mathcal{T}[1..n]$ and a pattern $\mathcal{P} = \mathcal{P}[1..m]$ where each $\mathcal{T}[i], \mathcal{P}[j] \subseteq \Sigma$, report all positions $i \in [1..n - m + 1]$ such that $\mathcal{P}$ set-parameterized matches $\mathcal{T}[i..i + m - 1]$.

1.3 Our Contributions

We present a linear time randomized algorithm for set parameterized matching which is novel in its solution approach. This even dramatically improves the previous bipartite matching approaches for the special case of set parameterized comparison.

► **Theorem 1 (Main Result).** *Given a set-string text $\mathcal{T}[1..n]$ and set-string pattern $\mathcal{P}[1..m]$ over alphabet Σ , where $N = \sum_{i=1}^n |\mathcal{T}[i]|$ and $M = \sum_{j=1}^m |\mathcal{P}[j]|$, we can report all set-parameterized matches in time $O(N + M)$ with high probability using a Monte Carlo randomized algorithm with error probability at most $1/n$.*

Key to our result is the introduction of a new structural definition we term the *multiset offset representation*. Unlike classical predecessor pointers which capture only local history, this representation preserves the global distribution of every character, ensuring that the necessary bijection properties hold even when multiple characters share identical positions. To process this representation, which can be quadratically large in the worst case, we employ a novel three-layer hashing scheme:

1. **Layer 1:** Hash the offset set of each character to a single integer using Karp-Rabin fingerprinting, compressing the interaction history.
2. **Layer 2:** Hash the *multiset* of these values at each position to a single integer (“mashing”), enabling efficient set-to-set comparisons that respect multiplicities.
3. **Layer 3:** Apply a rolling Karp-Rabin hash to the sequence of mashed values to enable linear time processing of sliding windows.

As a warmup to the main result, we also demonstrate an efficient solution for the comparison problem:

► **Theorem 2** (Set-String Comparison). *Given two set-strings $S_1[1..m]$ and $S_2[1..m]$ over alphabet Σ , where $M_{S_1} = \sum_{i=1}^m |S_1[i]|$ and $M_{S_2} = \sum_{i=1}^m |S_2[i]|$, we can determine whether they set-parameterized match in time $O(M_{S_1} + M_{S_2})$ with error probability at most $1/m$.*

1.4 Organization

The remainder of the paper is organized as follows. Section 2 defines our model and the necessary notation. Section 3 serves as a warmup, proving Theorem 2 for comparing two set-strings. Section 4 details our main pattern matching algorithm, proving Theorem 1. Finally, we analyze the correctness and complexity in Section 5.

2 Model and Definitions

2.1 Notation and Basic Definitions

Throughout this paper, we denote by Σ a finite alphabet and by Σ^* the set of all finite strings over Σ . We assume that $\Sigma = \{1, 2, \dots, |\Sigma|\}$. If not, we sort Σ and rename the alphabet characters. For a string $S \in \Sigma^*$, we write $|S|$ for its size (length) and $S[i]$ for its i -th character, where $i \in [1..|S|]$. We use the notation $[n] := \{1, 2, \dots, n\}$ for positive integers n . For a prime p , we denote the field $\mathbb{Z}_p := \{0, 1, \dots, p-1\}$, the integers modulo p .

2.2 Classical Parameterized Matching

We begin by recalling the classical parameterized matching framework of Baker [7, 8], which forms the foundation of our work.

► **Definition 3** (Parameterized Match [7]). *Let $P[1..m], T[1..m] \in \Sigma^*$ be two strings of equal length over alphabet Σ . We say that P and T parameterized match, denoted $P \sim_p T$, if there exists a bijection $\pi : \Sigma \rightarrow \Sigma$ such that $T[i] = \pi(P[i])$ for all $i \in [1..m]$.*

► **Remark 4.** The bijection π in Definition 3 must be defined on the entire alphabet Σ , though it need only be consistent on the characters actually appearing in P and T . Some works distinguish between “constant” and “parameter” alphabets $\Sigma_C \cup \Sigma_P$ where π acts as the identity on Σ_C ; for simplicity, we consider the case where all characters are parameters (equivalently, $\Sigma_C = \emptyset$).

► **Problem 2** (Parameterized Pattern Matching).

Input: A text $T[1..n]$ and a pattern $P[1..m]$ over alphabet Σ .

Output: All positions $i \in [1..n - m + 1]$ such that $P \sim_p T[i..i + m - 1]$.

Baker’s key insight was to transform the string into a structural representation invariant under permutation. This is the *prev* transformation (or *predecessor encoding*):

► **Definition 5** (Prev Transformation [7]). *For a string $S[1..n]$ over alphabet Σ , the prev transformation $\text{prev}(S)$ is the integer sequence $\text{prev}(S)[1..n]$ where*

$$\text{prev}(S)[i] = \begin{cases} 0 & \text{if } S[i] \text{ first occurs at position } i \\ i - j & \text{otherwise, where } j = \max\{k < i : S[k] = S[i]\} \end{cases}$$

► **Lemma 6** (Correctness of Prev [7]). *For strings $P[1..m], T[1..m]$ over Σ , we have $P \sim_p T$ if and only if $\text{prev}(P) = \text{prev}(T)$ (equality of integer sequences).*

2.3 Set-Strings and Set Parameterized Matching

We now formalize the set parameterized matching model, which generalizes the concepts above to sequences of sets.

► **Definition 7 (Set-String).** A set-string $\mathcal{S}$ of length n over alphabet Σ is a sequence $\mathcal{S}[1..n]$ where each $\mathcal{S}[i] \subseteq \Sigma$ is a finite set of characters. The length of $\mathcal{S}$ is n and its size is $|\mathcal{S}| = \sum_{j=1}^n |\mathcal{S}[j]|$. We write $\mathcal{S}[i..j]$ for the subsequence $\mathcal{S}[i], \mathcal{S}[i+1], \dots, \mathcal{S}[j]$.

► **Definition 8 (Set Parameterized Match).** Let $\mathcal{S}_1[1..m]$ and $\mathcal{S}_2[1..m]$ be two set-strings of equal length over alphabet Σ . We say that $\mathcal{S}_1$ and $\mathcal{S}_2$ set-parameterized match, denoted $\mathcal{S}_1 \approx_p \mathcal{S}_2$, if there exists a bijection $\pi : \Sigma \rightarrow \Sigma$ such that for all $i \in [1..m]$: $\pi(\mathcal{S}_1[i]) = \mathcal{S}_2[i]$ where $\pi(A) := \{\pi(a) : a \in A\}$ for any set $A \subseteq \Sigma$.

► **Remark 9.** Definition 8 requires that the bijection π simultaneously rename all characters in all sets consistently. This generalizes Definition 3: if each set $\mathcal{P}[i]$ and $\mathcal{T}[i]$ is a singleton, we recover classical parameterized matching.

► **Problem 3 (Reformulation of Set Parameterized Matching).**

Input: A text $\mathcal{T}[1..n]$ and a pattern $\mathcal{P}[1..m]$, both set-strings over alphabet Σ .

Output: All positions $i \in [1..n - m + 1]$ such that $\mathcal{P} \approx_p \mathcal{T}[i..i + m - 1]$.

For the special case where $n = m$ we call the problem the *Set Parameterized Comparison* as all that is required is to verify if the strings parameterize match each other.

2.4 Offset Representation

A naive extension of the prev transformation to set-strings – where one replaces every character in every set with its predecessor offset – fails because it destroys the continuity of character chains. Consider the following counter-example: Let set-string $\mathcal{S}_1 = \{\emptyset, \{b\}, \emptyset, \emptyset, \{a\}, \emptyset, \emptyset, \{a, b\}, \emptyset, \emptyset, \emptyset, \{a\}\}$ and $\mathcal{S}_2 = \{\emptyset, \{c\}, \emptyset, \emptyset, \{d\}, \emptyset, \emptyset, \{c, d\}, \emptyset, \emptyset, \emptyset, \{c\}\}$. If we simply replace each character with its prev value, the resulting sets of prevs match perfectly at every position: $\text{prev}(\mathcal{S}_1) = \text{prev}(\mathcal{S}_2) =$

$$\{\emptyset, \{0\}, \emptyset, \emptyset, \{0\}, \emptyset, \emptyset, \{3, 6\}, \emptyset, \emptyset, \emptyset, \{5\}\}$$

The sets of prevs are identical, yet there is no valid parameterization. In $\mathcal{S}_1$, the character appearing at 13 (“a”) is the one that started at 5. In $\mathcal{S}_2$, the character at 13 (“c”) is the one that started at 2. The simple set of prevs $\{3, 6\}$ at position 8 obscures which prev belongs to which character chain. To overcome this, we represent all appearances of a specific character “together” as a structural unit. While a “prev set” (listing all predecessor offsets for a character) might capture this structure, it is not convenient for the rolling hash updates required by our algorithm. Instead, we chose to represent this “togetherness” using *offset sets*. Since a character may appear in multiple sets, and a set contains multiple characters, a single “previous occurrence” is ill-defined. We must instead track the *set* of previous offsets. This motivates the following definition.

► **Definition 10 (Offset Set).** Let $\mathcal{S}[1..n]$ be a set-string over Σ , and let $\sigma \in \Sigma$. Suppose σ appears in $\mathcal{S}$ at positions $i_1 < i_2 < \dots < i_k$ (i.e., $\sigma \in \mathcal{S}[i_j]$ for all $j \in [1..k]$). The offset set of σ in $\mathcal{S}$ is

$$\text{Offset}_{\mathcal{S}}(\sigma) := \{i_j - i_1 : j \in [1..k]\} \subseteq \{0, 1, 2, \dots, n - 1\}.$$

If σ does not appear in $\mathcal{S}$, we define $\text{Offset}_{\mathcal{S}}(\sigma) := \emptyset$.

Note that multisets are necessary as we may have two characters in one set with the same offset set. We need to keep both of them as the multiplicity of characters in the compared set-strings must be equal. With the offset set defined for each individual character, we now aggregate this information to represent the structure of the entire set-string. At any given position i in the string, multiple characters may be present. To fully capture the parameterization constraints at this position, we must collect the offset sets of *all* characters appearing there, forming the signature of that position, allowing us to compare the structural roles of characters across two different strings.

► **Definition 11** (Multiset Offset Representation). *For a set-string $\mathcal{S}$ of length m , the offset representation at position i , denoted $\widehat{\mathcal{S}}[i]$, is the multiset of offset sets for all characters appearing at $\mathcal{S}[i]$.*

$$\widehat{\mathcal{S}}[i] = \{ \text{Offset}_{\mathcal{S}}(\sigma) \mid \sigma \in \mathcal{S}[i] \}$$

and

$$\widehat{\mathcal{S}} = \widehat{\mathcal{S}}[1] \widehat{\mathcal{S}}[2] \cdots \widehat{\mathcal{S}}[n]$$

In continuation with the former example:

$$\widehat{\mathcal{S}}_1 = \{ \emptyset, \{ \{0, 6\} \}, \emptyset, \emptyset, \{ \{0, 3, 8\} \}, \emptyset, \emptyset, \{ \{0, 6\}, \{0, 3, 8\} \}, \emptyset, \emptyset, \emptyset, \{ \{0, 3, 8\} \} \} \text{ and}$$

$$\widehat{\mathcal{S}}_2 = \{ \emptyset, \{ \{0, 6, 11\} \}, \emptyset, \emptyset, \{ \{0, 3\} \}, \emptyset, \emptyset, \{ \{0, 6, 11\}, \{0, 3\} \}, \emptyset, \emptyset, \emptyset, \{ \{0, 6, 11\} \} \}$$

► **Observation 12.** *For a set-string $\mathcal{S}[1..n]$ over alphabet Σ , each offset set $\text{Offset}_{\mathcal{S}}(\sigma)$ has size at most n . Therefore, $\widehat{\mathcal{S}}[i]$ contains exactly $|\mathcal{S}[i]|$ offset sets, each of size at most n . Letting $N = \sum_{i=1}^n |\mathcal{S}[i]|$ denote the total number of character occurrences, in the worst case, $|\widehat{\mathcal{S}}|$ (the total number of integers in the offset representation) is $O(nN)$.*

The following lemma, proof in appendix, shows that the Multiset Offset Representation provides a necessary and sufficient condition for set parameterized matching.

► **Lemma 13.** *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be set-strings of length m . Then $\mathcal{S}_1 \approx_p \mathcal{S}_2$ if and only if $\widehat{\mathcal{S}}_1 = \widehat{\mathcal{S}}_2$ (where equality implies equality of the sequence of multisets).*

2.5 Challenges for Set Parameterized Comparison and Matching

While Lemma 13 successfully reduces set-parameterized comparison to checking equality of offset representations, it does not immediately yield an efficient algorithm for set parameterized comparison and, definitely, not for set parameterized matching, due to two distinct hurdles:

1. **Size blowup:** By Observation 12, the offset representation can be quadratically larger than the input. Comparing these representations directly is prohibitively expensive.
2. **Dynamic offset sets:** For set parameterized *matching*, we compare $\mathcal{P}$ against sliding windows $\mathcal{T}[i..i+m-1]$. The offset sets $\text{Offset}_{\mathcal{T}[i..i+m-1]}(\sigma)$ depend on the start index i and must be recomputed for each position. This contrasts with classical pattern matching, where the text “characters” remain static throughout the algorithm.

Our algorithms addresses these challenges via a multi-layer *hashing* scheme, detailed in Sections 3 and 4.

2.6 Karp-Rabin Fingerprinting

Our algorithms employ the Karp-Rabin fingerprinting technique [19]. We distinguish between two variants of the hash function, depending on whether the input is treated as a sequence (order matters) or a multiset (order irrelevant).

► **Definition 14** (Sequence Hash). *Let p be a prime and $r \in \mathbb{Z}_p$. For a sequence of integers $A = (a_0, a_1, \dots, a_{k-1})$, the sequence hash treats the elements as coefficients of a polynomial:*

$$h_{p,r}^{seq}(A) := \sum_{i=0}^{k-1} a_i \cdot r^i \bmod p.$$

► **Definition 15** (Set/Multiset Hash). *Let S be a multiset of integers drawn from a universe $\{0, \dots, U-1\}$. Let c_v denote the multiplicity of value v in S . The set/multiset hash treats the values as exponents:*

$$h_{p,r}^{set}(S) := \sum_{v=0}^{U-1} c_v \cdot r^v \bmod p.$$

► **Lemma 16** (Collision Probability [19]). *Let p be a prime and $r \in \mathbb{Z}_p$ be chosen uniformly at random.*

1. **Sequence Collision:** *Let A and B be two distinct sequences of length at most ℓ . Then:*

$$\Pr[h_{p,r}^{seq}(A) = h_{p,r}^{seq}(B)] \leq \frac{\ell-1}{p}$$

2. **Multiset Collision:** *Let S_1 and S_2 be two distinct multisets containing values from the universe $\{0, \dots, U-1\}$. Then:*

$$\Pr[h_{p,r}^{set}(S_1) = h_{p,r}^{set}(S_2)] \leq \frac{U}{p}$$

Proof. In both cases, the collision condition $h(X) = h(Y)$ is equivalent to r being a root of the non-zero difference polynomial $Q(x) = h(X) - h(Y)$ over the field $\mathbb{Z}_p$. By the fundamental theorem of algebra, the number of roots is bounded by the degree of $Q(x)$.

1. For sequences, $Q(r) = \sum (a_i - b_i)r^i$. The degree is bounded by the maximum index, which is $\ell - 1$.
2. For multisets, $Q(r) = \sum (c_{1,v} - c_{2,v})r^v$. The degree is bounded by the maximum value v such that counts differ, which is at most U .

Dividing the degree by the field size p yields the probability bound. ◀

2.7 Rolling Hashes

In the classical Rabin-Karp pattern matching algorithm [19], the rolling hash essentially computes a fingerprint for every window of length m in the text, and compares it to the fingerprint of the pattern.

► **Definition 17** (Rolling Hash). *For a sequence $S[1..n]$ where each $S[i] \in \mathbb{Z}_p$, and a window size $m \leq n$, define*

$$H_{p,r}^{(m)}(S, i) := \sum_{j=0}^{m-1} S[i+j] \cdot r^j \bmod p$$

for $i \in [1..n-m+1]$. This is the rolling hash of the window $S[i..i+m-1]$.

A key property of this scheme is the ability to efficiently update the hash value as the window slides from one position to the next.

► **Observation 18** (Efficient Rolling Hash Update). *Given $H_{p,r}^{(m)}(S, i)$, we can compute $H_{p,r}^{(m)}(S, i + 1)$ in $O(1)$ time via*

$$H_{p,r}^{(m)}(S, i + 1) = \frac{H_{p,r}^{(m)}(S, i) - S[i]}{r} + S[i + m] \cdot r^{m-1} \bmod p.$$

Our algorithm, to appear later, builds upon this rolling concept. However, adapting it to parameterized matching, and specifically set parameterized matching, introduces significant complexity: the effective value of a “character” (its offset) is relative to the window’s start position and thus changes dynamically as the window rolls.

3 Set Parameterized Comparison

In this section, we prove Theorem 2 by presenting an efficient algorithm for set parameterized comparison, i.e., determining whether two set-strings of equal length set-parameterized match. This algorithm serves as a conceptual warmup for our main pattern matching algorithm and introduces the first two layers of our hashing scheme.

3.1 Algorithm Overview

The algorithm proceeds in three logical steps to progressively compress the set-string structure:

1. First, we compute the explicit offset sets for all characters in both set-strings.
2. Next, we hash each offset set to a single integer fingerprint using Karp-Rabin hashing (**Layer 1**).
3. Finally, we hash the set of fingerprints at each position to obtain a single value, allowing for linear comparisons (**Layer 2**).

3.2 Layer 1: Hashing Offset Sets

We first address the computational cost of generating the offset sets. Recall that $\Sigma = \{1, 2, \dots, |\Sigma|\}$.

► **Lemma 19** (Offset Set Computation). *Given a set-string $\mathcal{S}[1..m]$ over alphabet Σ of size $M = \sum_{i=1}^m |\mathcal{S}[i]|$, we can compute $\text{Offset}_{\mathcal{S}}(\sigma)$ for all $\sigma \in \Sigma$ in time $O(M)$ and space $O(M)$.*

Proof. We perform a single pass over the set-string. We maintain a hash table $\text{FIRSTOCC} : \Sigma \rightarrow [1..m] \cup \{\perp\}$ to track the first occurrence of each character, and for each $\sigma \in \Sigma$, a list $\text{OFFSETS}[\sigma]$ initially empty.

```

1: for  $\sigma \in \Sigma$  do
2:    $\text{FIRSTOCC}[\sigma] \leftarrow \perp$ 
3:    $\text{OFFSETS}[\sigma] \leftarrow \emptyset$ 
4: end for
5: for  $i = 1$  to  $m$  do
6:   for  $\sigma \in \mathcal{S}[i]$  do
7:     if  $\text{FIRSTOCC}[\sigma] = \perp$  then
8:        $\text{FIRSTOCC}[\sigma] \leftarrow i$ 
9:        $\text{OFFSETS}[\sigma] \leftarrow \text{OFFSETS}[\sigma] \cup \{0\}$ 
10:    else
```


36:8 Set Parameterized Matching via Multi-Layer Hashing

```

11:     OFFSETS[σ] ← OFFSETS[σ] ∪ {i − FIRSTOCC[σ]}
12:   end if
13: end for
14: end for

```

The initialization loop takes $O(|\Sigma|)$ time, which is $O(M)$. The outer loop runs m times, and the inner loop processes each character in each set. In total, we process exactly $M = \sum_{i=1}^m |\mathcal{S}[i]|$ character-position pairs. Each operation takes $O(1)$ expected time with hashing, giving a total time complexity of $O(M)$. ◀

Once the offset sets are computed, we compress them using the set hash function from Def. 15.

► **Definition 20** (Layer 1 Hash). *Let p_1 be a prime and let $r_1 \in \mathbb{Z}_{p_1}$ be chosen uniformly at random. For an offset set $O \subseteq \{0, 1, \dots, m-1\}$, the Layer 1 hash function is:*

$$\phi_1(O) := h_{p_1, r_1}^{set}(O) = \sum_{x \in O} r_1^x \bmod p_1.$$

Collision Analysis. By the definition, the Layer 1 hash is exactly the multiset hash $\phi_1(O) = h_{p_1, r_1}^{set}(O)$. The elements of the offset sets are integers drawn from the universe $\{0, \dots, m-1\}$. We apply Lemma 16 (Multiset Collision case) with universe bound $U = m$, yielding for $O_1 \neq O_2$:

$$\Pr[\phi_1(O_1) = \phi_1(O_2)] \leq \frac{m}{p_1}.$$

3.3 Layer 2: Hashing Sets of Fingerprints

After applying Layer 1, each position i in set-string $\mathcal{S}$ is represented by a multiset of fingerprints:

$$\Phi_1(\mathcal{S}[i]) := \{\phi_1(\text{Offset}_{\mathcal{S}}(\sigma)) : \sigma \in \mathcal{S}[i]\}.$$

To compare positions efficiently, we must compare these multisets. We therefore apply a second layer of hashing to “mash” these multisets into single values. Specifically, we use the multiset hash function from Definition 15.

► **Definition 21** (Layer 2 Hash). *Let p_2 be a prime and let $r_2 \in \mathbb{Z}_{p_2}$ be chosen uniformly at random. For a multiset F of fingerprints from $\mathbb{Z}_{p_1}$, the Layer 2 hash function is:*

$$\phi_2(F) := h_{p_2, r_2}^{set}(F) = \sum_{f \in F} r_2^f \bmod p_2.$$

We call the resulting hash value the *mashed* value of the position and the *mashed representation* of a set-string $\mathcal{S}[1..m]$ is the sequence $\mathcal{M}(\mathcal{S}) = (M_1, M_2, \dots, M_m)$ where $M_i := \phi_2(\Phi_1(\mathcal{S}[i]))$.

Collision Analysis. By the definition, Layer 2 is also the multiset hash $\phi_2(F) = h_{p_2, r_2}^{set}(F)$. The elements of the multisets are Layer 1 fingerprints, which are integers drawn from the universe $\{0, \dots, p_1-1\}$. Therefore, we can apply Lemma 16 (Multiset Collision case) with universe bound $U = p_1$, yielding:

$$\Pr_{r_2}[\phi_2(F_1) = \phi_2(F_2)] \leq \frac{p_1}{p_2}$$

3.4 Set Parameterized Comparison Algorithm

The full procedure for set parameterized comparison, comparing two set-strings, consisting of applying the 2-level hash to each, is formalized in Algorithm 1 and appears in the appendix.

► **Theorem 22** (Correctness of Comparison Algorithm). *Algorithm 1 correctly determines whether $\mathcal{S}_1 \approx_p \mathcal{S}_2$ with probability at least $1 - m^{-1}$.*

Proof.

Completeness. Assume $\mathcal{S}_1 \approx_p \mathcal{S}_2$. By Lemma 13, we know $\widehat{\mathcal{S}}_1 = \widehat{\mathcal{S}}_2$. This implies that for every position i and every $\sigma \in \mathcal{S}_1[i]$, there exists a corresponding $\tau \in \mathcal{S}_2[i]$ such that $\text{OffSet}_{\mathcal{S}_1}(\sigma) = \text{OffSet}_{\mathcal{S}_2}(\tau)$, and vice versa. It follows deterministically that:

- $\Phi_1(\mathcal{S}_1[i]) = \Phi_1(\mathcal{S}_2[i])$ for all i (as sets), and consequently,
- $M_i^{\mathcal{S}_1} = M_i^{\mathcal{S}_2}$ for all i .

Thus, the algorithm correctly returns **true**.

Soundness. Assume $\mathcal{S}_1 \not\approx_p \mathcal{S}_2$. By Lemma 13, $\widehat{\mathcal{S}}_1 \neq \widehat{\mathcal{S}}_2$. Thus, there must exist some position i^* where the sets of offset sets differ: $\widehat{\mathcal{S}}_1[i^*] \neq \widehat{\mathcal{S}}_2[i^*]$. Let $\mathcal{E}_1$ be the event that “there exist distinct offset sets O_1, O_2 with $\phi_1(O_1) = \phi_1(O_2)$.” By choosing $p_1 = \Theta(m^2|\Sigma|^2)$ and taking the union bound over all $\binom{|\Sigma|}{2} \leq |\Sigma|^2$ pairs of distinct characters, we bound the probability of this collision:

$$\Pr[\mathcal{E}_1] \leq |\Sigma|^2 \cdot \frac{m}{p_1} = O\left(\frac{m|\Sigma|^2}{m^2|\Sigma|^2}\right) = O\left(\frac{1}{m}\right).$$

Conditioned on $\neg\mathcal{E}_1$, the hash function ϕ_1 is injective on the offset sets appearing in $\mathcal{S}_1$ and $\mathcal{S}_2$. Since $\widehat{\mathcal{S}}_1[i^*] \neq \widehat{\mathcal{S}}_2[i^*]$, it must be that $\Phi_1(\mathcal{S}_1[i^*]) \neq \Phi_1(\mathcal{S}_2[i^*])$ as sets of fingerprints. Next, let $\mathcal{E}_2$ be the event that “there exist distinct sets $F_1, F_2 \subseteq \mathbb{Z}_{p_1}$ with $\phi_2(F_1) = \phi_2(F_2)$.” Choose $p_2 = \Theta(m^4|\Sigma|^2)$. We are comparing m multisets (one for each position $i \in [1..m]$). For any specific position, the collision probability is bounded by p_1/p_2 (by Lemma 16 with universe p_1). Applying the union bound over all m positions:

$$\Pr[\mathcal{E}_2 \mid \neg\mathcal{E}_1] \leq m \cdot \frac{p_1}{p_2} = O\left(\frac{m^3|\Sigma|^2}{m^4|\Sigma|^2}\right) = O\left(\frac{1}{m}\right).$$

Conditioned on $\neg\mathcal{E}_1 \wedge \neg\mathcal{E}_2$, we are guaranteed that $M_{i^*}^{\mathcal{S}_1} \neq M_{i^*}^{\mathcal{S}_2}$, and so the algorithm returns **false**. The overall error probability is bounded by the sum of probabilities of these collision events:

$$\Pr[\mathcal{E}_1 \vee \mathcal{E}_2] \leq \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2 \mid \neg\mathcal{E}_1] = O\left(\frac{1}{m}\right). \quad \blacktriangleleft$$

This concludes the proof of Theorem 2, with a summary appearing in the index.

4 Set Parameterized Matching Algorithm

We now present our main algorithm for Problem 3, the problem of Set Parameterized Matching, proving Theorem 1.

4.1 The Rolling Hash Challenge

Recall from Section 2 (Definition 17) that the Rabin-Karp rolling hash allows for efficient $O(1)$ updates when sliding a window over a fixed sequence of integers. In our context, the natural candidate for such a sequence is the sequence of Layer 2 mashed values, $M_1^T, M_2^T, \dots, M_n^T$. A direct application of the standard rolling hash for a window of size m starting at text position i would be:

$$H = \sum_{j=0}^{m-1} M_{i+j}^T \cdot r_3^j \bmod p_3$$

where p_3 is some prime, $r_3 \in [p_3]$, and M_k^T represents the Layer 2 hash of the offset sets at text position k .

The Dynamic Offset Problem

However, this standard approach fails because the values M_k^T are *not static*. In set parameterized matching, the offset set $\text{Offset}_{\mathcal{T}[i..i+m-1]}(\sigma)$ is defined relative to the *first occurrence* of σ inside the current window. When the window slides from i to $i+1$, if a character σ is dropped from position i , the “first occurrence” of σ changes to its next appearance in the window (say, at relative distance δ). Consequently, the offsets of *all* remaining occurrences of σ in the window must be decreased by δ . This changes the Layer 1 hashes (ψ) and, crucially, changes the Layer 2 mashed values (M^T) for every position where σ appears. Updating the rolling hash H would thus require recomputing up to $O(m)$ terms in the summation, which is prohibitively slow.

4.2 Character-Centric Rolling Hash

To handle these dynamic updates efficiently, we restructure the hashing scheme. Instead of viewing H as a sum over window positions (which is sensitive to shifting offsets), we swap the order of summation to view it as a sum over alphabet characters. Recall that for the window starting at location i , $M_{i+j}^T = \sum_{\sigma \in \mathcal{T}[i+j]} r_2^{\psi^{(i)}[\sigma]}$. Substituting this into the standard rolling hash definition:

$$H = \sum_{j=0}^{m-1} r_3^j \left(\sum_{\sigma \in \mathcal{T}[i+j]} r_2^{\psi^{(i)}[\sigma]} \bmod p_2 \right) \bmod p_3$$

By swapping the summation order, we group terms by character rather than by position:

$$H = \sum_{\sigma \in \Sigma} \left(r_2^{\psi^{(i)}[\sigma]} \bmod p_2 \right) \left(\sum_{j: \sigma \in \mathcal{T}[i+j]} r_3^j \right) \bmod p_3$$

This rearrangement separates the *structural shape* of the encoding of σ (encoded by $r_2^{\psi^{(i)}[\sigma]}$) from the *geometric positions* of the encoding of σ (encoded by the inner sum over r_3). This separation allows us to update the two components independently.

This motivates our formal definition of the components for the character-centric hash. We first isolate the “geometric” component – the inner sum representing the positions of a character.

► **Definition 23** (Character Position Fingerprint). *Let p_2 be a prime and let $r_2 \in \mathbb{Z}_{p_2}$ be chosen uniformly at random. For a character σ in the window starting at text position i , let its position fingerprint be the sum of the position powers for all its occurrences:*

$$P_i(\sigma) := \sum_{k \in \text{Occurrences of } \sigma \text{ in window}} r_3^k \bmod p_3$$

where $k \in \{0, \dots, m-1\}$ is the relative index of the occurrence within the window.

With the geometric component isolated, the total window hash can be expressed as a weighted sum of these fingerprints, where the weights are determined by the structural shape (offset sets) of each character.

► **Definition 24** (Layer 3 Hash). *The character-centric Layer 3 hash for the window at text position i is:*

$$H_i^T := \sum_{\sigma \in \Sigma} \left(r_2^{\psi^{(i)}[\sigma]} \bmod p_2 \right) \cdot P_i(\sigma) \bmod p_3$$

where $\psi^{(i)}[\sigma]$ is the Layer 1 hash of the offset set of σ in window i .

4.3 Efficient Transition Logic

To achieve our time complexity goals, we must compute H_{i+1}^T from H_i^T in time proportional only to the number of characters entering or leaving the window, independent of the alphabet size $|\Sigma|$. When sliding the window from i to $i+1$, two distinct types of updates occur:

1. **Geometric Shift:** Every character remaining in the window shifts left by one position. This implies that for every $\sigma \in \Sigma$, the position fingerprint changes: $P_{i+1}(\sigma) = P_i(\sigma) \cdot r_3^{-1}$.
2. **Structural Change:** For characters in $\mathcal{T}[i]$ (leaving) and $\mathcal{T}[i+m]$ (entering), the offset structure $\psi^{(i)}[\sigma]$ may change to a new value $\psi^{(i+1)}[\sigma]$.

A naive update is impossible because we cannot iterate over all $\sigma \in \Sigma$ to apply the geometric shift r_3^{-1} , nor can we simply multiply the total hash H_i^T by r_3^{-1} (since the structural weights $r_2^{\psi[\sigma]}$ for active characters would be incorrectly shifted). Instead, we decompose the total hash into a *Stable Component* (characters with unchanged structure) and an *Active Component* (characters appearing at the window boundaries). We process the transition in three steps: *Isolate*, *Shift*, and *Reintegrate*. Let $S_{\text{active}} = \mathcal{T}[i] \cup \mathcal{T}[i+m]$ be the set of characters undergoing structural updates.

Step 1: Isolate (Remove Active Terms)

First, we remove the contributions of all “active” characters from the current hash to isolate the “stable” portion. For any stable character $\sigma \notin S_{\text{active}}$, the structural hash ψ remains invariant ($\psi^{(i)}[\sigma] = \psi^{(i+1)}[\sigma]$).

$$H_{\text{stable}} = H_i^T - \sum_{\sigma \in S_{\text{active}}} \left(r_2^{\psi^{(i)}[\sigma]} \bmod p_2 \right) \cdot P_i(\sigma) \bmod p_3$$

This step requires $O(|S_{\text{active}}|)$ operations.

Step 2: Global Shift (Update Stable Terms)

The remaining hash H_{stable} consists essentially of a sum of terms $C_\sigma \cdot P_i(\sigma)$ where C_σ is constant. The only required update is the geometric shift of positions ($k \rightarrow k - 1$). By the linearity of the sum, we can apply this shift globally:

$$H'_{stable} = H_{stable} \cdot r_3^{-1} \bmod p_3$$

This single multiplication updates the hash for all non-active characters in $O(1)$ time. **Lazy Fingerprint Maintenance:** To support this global shift without updating individual $P(\sigma)$ values in memory, we maintain a global rolling multiplier R_{glob} . The stored value for a fingerprint is related to the actual value by $P_{actual}(\sigma) = P_{stored}(\sigma) \cdot R_{glob}$. In this step, we simply update $R_{glob} \leftarrow R_{glob} \cdot r_3^{-1}$.

Step 3: Reintegrate (Add Updated Active Terms)

Finally, we compute the new states for the active characters and add them back to the shifted stable hash. For each $\sigma \in S_{active}$:

- **Update Position:** We update $P_{stored}(\sigma)$ to reflect any removals (at relative index 0) or insertions (at relative index $m - 1$). Note that we must multiply any added terms by R_{glob}^{-1} to align with the current global frame.
- **Update Offset Structure:**
 - *Leaving* ($\sigma \in \mathcal{T}[i]$): We must remove the first occurrence. Let δ be the distance to the *next* occurrence of σ in the window. The structure updates by shifting the exponents:

$$\psi^{(i+1)}[\sigma] \leftarrow (\psi^{(i)}[\sigma] - r_1^0) \cdot r_1^{-\delta} \bmod p_1$$

(If σ does not appear again, $\psi^{(i+1)}[\sigma]$ resets to 0).

- *Entering* ($\sigma \in \mathcal{T}[i + m]$): We must add the new occurrence at the end of the window. Let k be the index of the *first occurrence* of σ in the new window (if σ is new to the window, $k = m - 1$). The new relative offset to add is $\delta_{new} = (m - 1) - k$.

$$\psi^{(i+1)}[\sigma] \leftarrow \psi^{(i+1)}[\sigma] + r_1^{\delta_{new}} \bmod p_1$$

We then add the new contribution to the total hash:

$$H_{i+1}^T = H'_{stable} + \sum_{\sigma \in S_{active}} \left(r_2^{\psi^{(i+1)}[\sigma]} \bmod p_2 \right) \cdot (P_{stored}(\sigma) \cdot R_{glob}) \bmod p_3$$

This step also takes time $O(|S_{active}|)$.

Note:

- (1) The definition of offset sets comes into its main play here, that is, the update of $\psi^{(i+1)}[\sigma]$ is computable in $O(1)$ time because of the structure of the offset set.
- (2) In the computation of the entering characters, we assume knowledge of the *first occurrence* of σ . We do this by saving linked lists of the locations of each σ and traverse them while shifting, changing pointers to current *first occurrence* in each list when characters leave the window. For the sake of brevity, we keep the details to a minimum.

► **Theorem 25** (Transition Complexity). *The transition from H_i^T to H_{i+1}^T can be computed in time $O(1 + |\mathcal{T}[i]| + |\mathcal{T}[i + m]|)$.*

Proof. Step 2 takes $O(1)$ time. Steps 1 and 3 involve iterating only over the characters present at the leaving position i and the entering position $i + m$. All internal lookups (offset lists, hash maps) take $O(1)$ time. Thus, the total time is proportional to the number of character occurrences added or removed. ◀

The full algorithm, depicting the above, is detailed in Algorithm 2 in the appendix.

5 Analysis

5.1 Correctness Analysis

We now prove that the hashing scheme correctly identifies matches with high probability.

► **Theorem 26 (Correctness).** *Algorithm 2 reports all valid set-parameterized matches and reports false positives with probability at most n^{-1} .*

Proof. The matching algorithm effectively performs a set-string comparison at each of the $n - m + 1$ text windows. We analyze the error probability by extending the proof of Theorem 22 by choosing primes: $p_1 = \Theta(n^2|\Sigma|^2m)$, $p_2 = \Theta(n^4|\Sigma|^2m^2)$, and $p_3 = \Theta(n^2m)$.

1. **Layers 1 & 2 (Structural Collision):** By the analysis in Theorem 22, the probability of a collision in the structural hashing (Layers 1 and 2) for any single window is bounded by:

$$\Pr[\mathcal{E}_{struct}] \leq \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2] \leq \frac{m|\Sigma|^2}{p_1} + \frac{mp_1}{p_2} = O\left(\frac{1}{n^2}\right) + O\left(\frac{1}{n^2}\right) = O\left(\frac{1}{n^2}\right).$$

2. **Layer 3 (Rolling Hash Collision):** The rolling hash H is a polynomial in r_3 of degree at most m . For any single window, using Lemma 16, the collision probability is:

$$\Pr[\mathcal{E}_{roll}] \leq \frac{m}{p_3} = O\left(\frac{m}{n^2m}\right) = O\left(\frac{1}{n^2}\right).$$

Total Error. Applying the union bound over all $O(n)$ sliding windows, the total probability of a false positive is:

$$\Pr[\text{Error}] \leq n \cdot (\Pr[\mathcal{E}_{struct}] + \Pr[\mathcal{E}_{roll}]) = n \cdot O\left(\frac{1}{n^2}\right) = O\left(\frac{1}{n}\right). \quad \blacktriangleleft$$

5.2 Complexity Analysis

Let $N = \sum_{i=1}^n |\mathcal{T}[i]|$ and $M = \sum_{j=1}^m |\mathcal{P}[j]|$. The following theorem summarizes the result. Details appear in the appendix.

► **Theorem 27 (Complexity).** *Algorithm 2 runs in time $O(N + M)$ and uses space $O(N + M)$.*

References

- 1 Amihod Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994. doi:10.1016/0020-0190(94)90086-8.
- 2 Amihod Amir and Gonzalo Navarro. Parameterized matching on non-linear structures. *Information Processing Letters*, 109(15):864–867, 2009. doi:10.1016/J.IPL.2009.04.012.
- 3 Alberto Apostolico, Péter L. Erdős, and Alpár Jüttner. Parameterized searching with mismatches for run-length encoded strings. *Theoretical Computer Science*, 454:23–29, 2012. doi:10.1016/J.TCS.2012.03.018.
- 4 Alberto Apostolico, Péter L. Erdős, and Moshe Lewenstein. Parameterized matching with mismatches. *Journal of Discrete Algorithms*, 5(1):135–140, 2007. doi:10.1016/J.JDA.2006.03.014.
- 5 Alberto Apostolico and Raffaele Giancarlo. Periodicity and repetitions in parameterized strings. *Discrete Applied Mathematics*, 156(9):1389–1398, 2008. doi:10.1016/J.DAM.2006.11.017.
- 6 Brenda S. Baker. A program for identifying duplicated code. In *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, 1992.

- 7 Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 71–80, 1993. doi:10.1145/167088.167115.
- 8 Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997. doi:10.1137/S0097539793246707.
- 9 Brenda S. Baker. Parameterized diff. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 854–855. SIAM, 1999. URL: <http://dl.acm.org/citation.cfm?id=314500.314968>.
- 10 Richard Beal and Donald A. Adjeroh. Compressed parameterized pattern matching. *Theor. Comput. Sci.*, 609:129–142, 2016. doi:10.1016/J.TCS.2015.09.015.
- 11 Pierre Bourhis, Aaron Boussidan, and Philippe Gambette. On distances between words with parameters. In *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023*, volume 259 of *LIPIcs*, pages 6:1–6:23, 2023. doi:10.4230/LIPIcs.CPM.2023.6.
- 12 Aaron Boussidan. *Comparaison automatique de pièces de théâtre à l’aide de distances d’édition et d’alignement*. Phd dissertation, Gustave Eiffel University, 2025.
- 13 Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms*, 11(2):12, 2014.
- 14 S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda. Parameterized suffix arrays for binary strings. In *Proceedings of the Prague Stringology Conference*, pages 84–94, 2008.
- 15 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Y. Yang. Space-efficient dictionaries for parameterized and order-preserving pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.2.
- 16 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 397–407. SIAM, 2017. doi:10.1137/1.9781611974782.25.
- 17 R. Garg, R. Prasad, and S. Agarwal. Fast parameterized word matching on compressed text. In *International Conference on Computer and Communication Technology (ICCCCT)*, pages 317–321. IEEE, 2014.
- 18 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms*, 3(3):29, 2007. doi:10.1145/1273340.1273345.
- 19 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/RD.312.0249.
- 20 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. On the longest common parameterized subsequence. *Theoretical Computer Science*, 410(51):5347–5353, 2009. doi:10.1016/J.TCS.2009.09.011.
- 21 R. Khetan, S. Agarwal, and R. Prasad. An efficient approach towards compressed parameterized word matching using wavelet tree. *Journal of Information and Optimization Sciences*, 37(2):285–301, 2016.
- 22 I. Lee, J. Mendivelso, and Yoan J. Pinzón. $\delta\gamma$ -parameterized matching. In *String Processing and Information Retrieval (SPIRE)*, volume 5280 of *LNCS*, pages 236–248, 2008.
- 23 Moshe Lewenstein. Parameterized matching. In *Encyclopedia of Algorithms*, pages 635–638. Springer, 2008. doi:10.1007/978-0-387-30162-4_282.
- 24 Juan Mendivelso, Sharma V. Thankachan, and Yoan J. Pinzón. A brief history of parameterized matching problems. *Discrete Applied Mathematics*, 274:103–115, 2020. doi:10.1016/J.DAM.2018.07.017.
- 25 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, 39(1):1–19, 2004. doi:10.1007/S00453-003-1067-9.

A

 Proof of Correctness for the Offset Sets

The following is the proof of Lemma 13.

Proof.

Direction ($\Rightarrow$). Suppose $\mathcal{S}_1 \approx_p \mathcal{S}_2$. There exists a bijection $\pi : \Sigma \rightarrow \Sigma$ such that $\pi(\mathcal{S}_1[i]) = \mathcal{S}_2[i]$ for all i . This implies that for every σ , the absolute positions of σ in $\mathcal{S}_1$ are identical to the absolute positions of $\pi(\sigma)$ in $\mathcal{S}_2$. Consequently, they share the same start position and the same offset set. Thus, the multiset of offset sets at every position remains invariant, and $\widehat{\mathcal{S}}_1 = \widehat{\mathcal{S}}_2$.

Direction ($\Leftarrow$). Suppose $\widehat{\mathcal{S}}_1 = \widehat{\mathcal{S}}_2$. We must construct a valid bijection π .

A valid bijection must map a character $\sigma \in \mathcal{S}_1$ to a character $\tau \in \mathcal{S}_2$ if and only if they are structurally identical. Structural identity requires two conditions:

1. They have the same **Offset Set** O (relative spacing).
2. They have the same **Start Position** s (absolute placement).

Let $N^{\mathcal{S}}(s, O)$ denote the number of distinct characters in string $\mathcal{S}$ that start at position s and have offset set O . To prove a bijection exists, it is sufficient to show that for all positions $s \in [1..m]$ and all offset sets O :

$$N^{\mathcal{S}_1}(s, O) = N^{\mathcal{S}_2}(s, O)$$

If this equality holds, we can map the k characters of type (s, O) in $\mathcal{S}_1$ arbitrarily to the k characters of type (s, O) in $\mathcal{S}_2$. **Computing $N(s, O)$ from the Representation:** We show that $N^{\mathcal{S}}(s, O)$ is uniquely determined by the sequence of multisets $\widehat{\mathcal{S}}$. For any position i and offset set O , let $Count^{\mathcal{S}}(i, O)$ be the multiplicity of O in the multiset $\widehat{\mathcal{S}}[i]$. A character σ contributes to the count of O at position i if:

1. $Offset(\sigma) = O$.
2. σ appears at i . This happens if σ started at some position $s \leq i$ such that $(i - s) \in O$.

We can express the total count at position i as the sum of characters starting *now* ($s = i$) and characters that started *earlier* ($s < i$) and are re-appearing:

$$Count^{\mathcal{S}}(i, O) = \underbrace{N^{\mathcal{S}}(i, O)}_{\text{Starts at } i} + \sum_{\delta \in O, \delta > 0} \underbrace{N^{\mathcal{S}}(i - \delta, O)}_{\text{Started at } i - \delta}$$

Rearranging to solve for the current starts:

$$N^{\mathcal{S}}(i, O) = Count^{\mathcal{S}}(i, O) - \sum_{\delta \in O, \delta > 0} N^{\mathcal{S}}(i - \delta, O)$$

Inductive Argument. We prove $N^{\mathcal{S}_1}(i, O) = N^{\mathcal{S}_2}(i, O)$ for all i by strong induction.

- **Base Case ($i = 1$):** There are no offsets $\delta > 0$ relative to 1. Thus $N^{\mathcal{S}}(1, O) = Count^{\mathcal{S}}(1, O)$. Since $\widehat{\mathcal{S}}_1 = \widehat{\mathcal{S}}_2$, the counts are identical, so $N^{\mathcal{S}_1}(1, O) = N^{\mathcal{S}_2}(1, O)$.
- **Inductive Step:** Assume $N^{\mathcal{S}_1}(j, O) = N^{\mathcal{S}_2}(j, O)$ for all $j < i$. In the equation for $N(i, O)$:
 - $Count(i, O)$ is identical because $\widehat{\mathcal{S}}_1[i] = \widehat{\mathcal{S}}_2[i]$.
 - The summation terms $N(i - \delta, O)$ involve indices strictly less than i . By the induction hypothesis, these terms are identical for $\mathcal{S}_1$ and $\mathcal{S}_2$.

Therefore, $N^{\mathcal{S}_1}(i, O) = N^{\mathcal{S}_2}(i, O)$.

Since the number of characters for every specific (Start Position, Offset Set) pair is identical, we can construct a perfect bijection π . ◀

B

 Proof of Theorem 2

► **Theorem 28** (Complexity of Comparison Algorithm). *Let $M_{S_1} = \sum_{i=1}^m |\mathcal{S}_1[i]|$ and $M_{S_2} = \sum_{i=1}^m |\mathcal{S}_2[i]|$ denote the total size (number of character occurrences) of the two set-strings. Algorithm 1 runs in time $O(M_{S_1} + M_{S_2})$ and uses space $O(M_{S_1} + M_{S_2})$.*

Proof.

Time Analysis.

- **Initialization:** Selecting primes and initializing tables takes $O(|\Sigma|)$ time.
- **Offset Set Computation:** By Lemma 19, computing the offset sets requires iterating through every character occurrence in both strings exactly once. The time required is $O(M_{S_1} + M_{S_2})$.
- **Layer 1 Hashing:** We calculate $\phi_1(O)$ for every character $\sigma \in \Sigma$. The cost of computing the hash for a specific offset set O is proportional to its size $|O|$. Since the sum of the sizes of all offset sets is exactly the total number of character occurrences, the total time is $\sum_{\sigma} |O_{\sigma}| = O(M_{S_1} + M_{S_2})$.
- **Layer 2 Hashing:** We calculate ϕ_2 for every position $i \in [1..m]$. The cost at position i is proportional to the number of characters in the set at that position, $|\mathcal{S}_1[i]|$ (or $|\mathcal{S}_2[i]|$). Summing over all positions i , the total time is $\sum_{i=1}^m |\mathcal{S}_1[i]| + \sum_{i=1}^m |\mathcal{S}_2[i]| = O(M_{S_1} + M_{S_2})$.
- **Comparison:** Comparing the two mashed sequences takes $O(m)$ time, which is dominated by $O(M_{S_1} + M_{S_2})$ since $m \leq M$.

Since $|\Sigma| = O(M_{S_1} + M_{S_2})$ the total running time is $O(M_{S_1} + M_{S_2})$.

Space Analysis. Storing the offset sets and the intermediate hash values requires space proportional to the total number of elements in the sets, which is $O(M_{S_1} + M_{S_2})$. ◀

C

 Proof of Complexity Analysis

Proof.

Time Complexity.

1. **Preprocessing:** Building the occurrence lists $\mathcal{L}$ takes $O(N)$. Computing the initial pattern hash takes $O(M)$.
2. **Sliding Window Scan:** The loop runs for $n - m$ steps. Inside the loop, the “Global Shift” (Step 2) takes $O(1)$ time. The “Isolate” (Step 1) and “Reintegrate” (Step 3) phases iterate only over the set of active characters $S_{active} = \mathcal{T}[i] \cup \mathcal{T}[i + m]$. Let $Cost(i)$ be the number of operations at step i . We have $Cost(i) = O(1 + |\mathcal{T}[i]| + |\mathcal{T}[i + m]|)$. Summing over all windows:

$$\text{Total Time} = \sum_{i=1}^{n-m} Cost(i) = \sum_{i=1}^{n-m} (O(1) + |\mathcal{T}[i]| + |\mathcal{T}[i + m]|)$$

Observe that each character occurrence in the text at position k appears in the set $\mathcal{T}[i]$ exactly once (when it leaves the window) and in the set $\mathcal{T}[i + m]$ exactly once (when it enters the window). Therefore:

$$\sum_{i=1}^{n-m} (|\mathcal{T}[i]| + |\mathcal{T}[i + m]|) \leq 2 \sum_{k=1}^n |\mathcal{T}[k]| = 2N$$

3. **Hash Updates:** All internal hash updates (multiplications, additions) and list lookups (finding next occurrence δ) take $O(1)$ time.
- Thus, the total time complexity is $O(N + M)$.

Space Complexity. The occurrence lists $\mathcal{L}$ store every character position, requiring $O(N)$ space. The hash maps for P_{stored} and ψ store one entry per unique character currently in the window. Since the number of unique characters cannot exceed the number of character occurrences, this is bounded by $O(N)$. The pattern storage takes $O(M)$. Thus, the total space is $O(N + M)$. ◀

D Algorithms

Algorithm 1 COMPARESETSTRINGS($\mathcal{S}_1, \mathcal{S}_2$).

Require: Set-strings $\mathcal{S}_1[1..m]$ and $\mathcal{S}_2[1..m]$ over alphabet Σ
Ensure: **true** if $\mathcal{S}_1 \approx_p \mathcal{S}_2$, **false** otherwise (with high probability)

- 1: Choose prime $p_1 = \Theta((m^2|\Sigma|)^2)$ and random $r_1 \in \mathbb{Z}_{p_1}$
- 2: Choose prime $p_2 = \Theta((m^4|\Sigma|)^2)$ and random $r_2 \in \mathbb{Z}_{p_2}$
- 3: Compute $\text{Offset}_{\mathcal{S}_1}(\sigma)$ for all $\sigma \in \Sigma$ {Lemma 19}
- 4: Compute $\text{Offset}_{\mathcal{S}_2}(\sigma)$ for all $\sigma \in \Sigma$
- 5: **for** $\sigma \in \Sigma$ **do**
- 6: $\psi_{\mathcal{S}_1}[\sigma] \leftarrow \phi_1(\text{Offset}_{\mathcal{S}_1}(\sigma))$ {Layer 1 hash}
- 7: $\psi_{\mathcal{S}_2}[\sigma] \leftarrow \phi_1(\text{Offset}_{\mathcal{S}_2}(\sigma))$
- 8: **end for**
- 9: **for** $i = 1$ to m **do**
- 10: $M_i^{\mathcal{S}_1} \leftarrow \phi_2(\{\psi_{\mathcal{S}_1}[\sigma] : \sigma \in \mathcal{S}_1[i]\})$ {Layer 2 hash}
- 11: $M_i^{\mathcal{S}_2} \leftarrow \phi_2(\{\psi_{\mathcal{S}_2}[\sigma] : \sigma \in \mathcal{S}_2[i]\})$
- 12: **end for**
- 13: **return** $(M_1^{\mathcal{S}_1}, \dots, M_m^{\mathcal{S}_1}) = (M_1^{\mathcal{S}_2}, \dots, M_m^{\mathcal{S}_2})$

Algorithm 2 SETPARAMETERIZEDMATCHING($\mathcal{P}, \mathcal{T}$).

```

1: Preprocessing:
2: Build occurrence lists  $\mathcal{L}$  for text.
3: Compute pattern hash  $H^{\mathcal{P}}$ .
4: Compute  $H_1^{\mathcal{T}}$  explicitly. Initialize  $P_{stored}(\sigma)$  and  $\psi[\sigma]$ .
5: Set global multiplier  $R_{glob} \leftarrow 1$ .
6:  $Candidates \leftarrow \{1\}$  if  $H_1^{\mathcal{T}} = H^{\mathcal{P}}$ .
7: Sliding Window:
8: for  $i = 1$  to  $n - m$  do
9:    $H_{stable} \leftarrow H_i^{\mathcal{T}}$ 
10:   $S_{active} \leftarrow \mathcal{T}[i] \cup \mathcal{T}[i + m]$ 
11:  1. Isolate (Remove Old Contributions):
12:  for  $\sigma \in S_{active}$  do
13:     $P_{curr} \leftarrow P_{stored}(\sigma) \cdot R_{glob} \bmod p_3$ 
14:     $H_{stable} \leftarrow H_{stable} - (r_2^{\psi(i)[\sigma]} \bmod p_2) \cdot P_{curr} \bmod p_3$ 
15:  end for
16:  2. Global Shift:
17:   $H_{stable} \leftarrow H_{stable} \cdot r_3^{-1} \bmod p_3$ 
18:   $R_{glob} \leftarrow R_{glob} \cdot r_3^{-1} \bmod p_3$ 
19:  3. Reintegrate (Update & Add New Contributions):
20:  for  $\sigma \in S_{active}$  do
21:    // Update Position Fingerprint
22:    if  $\sigma \in \mathcal{T}[i]$  then
23:       $P_{stored}(\sigma) \leftarrow P_{stored}(\sigma) - (r_3^0 \cdot R_{glob}^{-1})$  {Remove leaving position}
24:    end if
25:    if  $\sigma \in \mathcal{T}[i + m]$  then
26:       $P_{stored}(\sigma) \leftarrow P_{stored}(\sigma) + (r_3^{m-1} \cdot R_{glob}^{-1})$  {Add entering position}
27:    end if
28:    // Update Offset Structure (Layer 1)
29:    if  $\sigma \in \mathcal{T}[i]$  then
30:      Find next occurrence distance  $\delta$  using  $\mathcal{L}$ .
31:      if  $\delta \leq m$  then
32:         $\psi[\sigma] \leftarrow (\psi[\sigma] - r_1^0) \cdot r_1^{-\delta} \bmod p_1$ 
33:      else
34:         $\psi[\sigma] \leftarrow 0$  {Reset if no more occurrences}
35:      end if
36:    end if
37:    if  $\sigma \in \mathcal{T}[i + m]$  then
38:      Determine relative offset  $\delta_{new}$  of new occurrence.
39:       $\psi[\sigma] \leftarrow \psi[\sigma] + r_1^{\delta_{new}} \bmod p_1$ 
40:    end if
41:     $P_{curr} \leftarrow P_{stored}(\sigma) \cdot R_{glob} \bmod p_3$ 
42:     $H_{stable} \leftarrow H_{stable} + (r_2^{\psi[\sigma]} \bmod p_2) \cdot P_{curr} \bmod p_3$ 
43:  end for
44:   $H_{i+1}^{\mathcal{T}} \leftarrow H_{stable}$ 
45:  if  $H_{i+1}^{\mathcal{T}} = H^{\mathcal{P}}$  then
46:     $Candidates \leftarrow Candidates \cup \{i + 1\}$ 
47:  end if
48: end for
49: return  $Candidates$ 

```
