


Dynamic Pattern Matching with Wildcards

Arshia Ataee Naeini  

University of Tehran, Iran

Amir-Parsa Mobed  

University of Tehran, Iran

Masoud Seddighin  

Tehran Institute for Advanced Studies (TeIAS), Iran

Saeed Seddighin  

Independent Researcher, Chicago, USA

Abstract

We study the fully dynamic pattern matching problem where the pattern may contain up to k wildcard symbols, each matching any symbol of the alphabet. Both the text and the pattern are subject to *updates* (insert, delete, change). We design an algorithm with $\mathcal{O}(n \log^2 n)$ preprocessing and update/query time $\tilde{\mathcal{O}}(kn^{k/k+1} + k^2 \log n)$. The bound is truly sublinear for a constant k , and sublinear when $k = o(\log n)$. We further complement our results with a conditional lower bound: assuming subquadratic preprocessing time, achieving truly sublinear update time for the case $k = \Omega(\log n)$ would contradict the Strong Exponential Time Hypothesis (SETH). Finally, we develop sublinear algorithms for two special cases:

- If the pattern contains w non-wildcard symbols, we give an algorithm with preprocessing time $\mathcal{O}(nw)$ and update time $\mathcal{O}(w + \log n)$, which is truly sublinear whenever w is truly sublinear.
- Using FFT technique combined with block decomposition, we design a deterministic truly sublinear algorithm with preprocessing time $\mathcal{O}(n^{1.8})$ and update time $\mathcal{O}(n^{0.8} \log n)$ for the case that there are at most two non-wildcards.

2012 ACM Subject Classification Theory of computation \rightarrow Problems, reductions and completeness

Keywords and phrases pattern matching, wildcards, dynamic algorithms, string algorithms, data structures

Digital Object Identifier 10.4230/LIPIcs.STACS.2026.68

Related Version *Full Version:* <https://arxiv.org/abs/2601.16182>

1 Introduction

Pattern matching is a fundamental problem in computer science with applications in text search, bioinformatics, log analysis, and data mining [37, 40, 20, 2, 25, 6, 36, 12, 15, 11, 28]. In its basic form, given a pattern P of length m and a text T of length n with symbols from alphabet Σ with size σ , the task is to decide whether P occurs in T as a *contiguous substring*; that is, whether there exists an index $i \in \{1, \dots, n - m + 1\}$ such that $T_{i:i+m-1} = P$. This problem has been studied for decades, and many efficient linear-time solutions such as the Knuth–Morris–Pratt and Boyer–Moore algorithms are well known [32, 10].

Traditionally, pattern matching aims to find exact occurrences of a pattern, yet in many practical applications such rigidity is unrealistic [41, 33, 35, 23]. The pattern or the text may contain uncertain or corrupted symbols due to noise, ambiguity, or small variations; for example when accounting for mutations in genomic data. This motivates the study of algorithms that search for substrings close to the pattern under standard distance measures such as Hamming distance [1, 13, 27]. Another common way to model uncertainty is to assume that the positions of the corrupted symbols are known. This is typically handled by introducing a *wildcard* symbol, which matches any single symbol of the alphabet Σ



© Arshia Ataee Naeini, Amir-Parsa Mobed, Masoud Seddighin, and Saeed Seddighin; licensed under Creative Commons License CC-BY 4.0

43rd International Symposium on Theoretical Aspects of Computer Science (STACS 2026).

Editors: Meena Mahajan, Florin Manea, Annabelle McIver, and Nguyễn Kim Thăng

Article No. 68; pp. 68:1–68:20



Leibniz International Proceedings in Informatics

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



[22, 26, 29]. This gives rise to the problem of *pattern matching with wildcards*, where across both the pattern and the text, the total number of wildcard positions is at most k . Since each wildcard can stand for any symbol of Σ , a pattern with k wildcards implicitly represents up to $|\Sigma|^k$ different strings.

Adding wildcards makes the pattern matching problem harder. With a fixed pattern, the search space is small. For example, let the text be $T = \text{CACCGGCT}$ and the pattern be $P = \text{CG}$. For this instance, there is only one match. If we instead use $P' = \text{C?}$, where “?” refers to the wildcard symbol, then every occurrence of C in T becomes a match. This simple change increases the number of matches from 1 to 4. Thus, even limited uncertainty can drastically increase the complexity of the problem.

A substantial body of work has studied pattern matching with wildcards in either the pattern or the text [29, 8, 5]. For this problem, Fischer and Paterson [22] gave an $\mathcal{O}(n \log m \log \sigma)$ algorithm, and later Indyk [26] gave a randomized $\mathcal{O}(n \log n)$ algorithm. Kalai [29] subsequently obtained a randomized $\mathcal{O}(n \log m)$ algorithm, while the first deterministic $\mathcal{O}(n \log m)$ algorithm was given by Cole and Hariharan [17].

In this paper, we study pattern matching with wildcards in a fully dynamic setting. Both the text and the pattern may change through insertions, deletions, or substitutions, including updates to wildcard positions. Our goal is to maintain pattern-matching queries in sublinear time. For dynamic pattern matching, most prior work considers settings without wildcards. In these studies, dynamism may arise in several ways: the text may be updated dynamically, the pattern may be updated dynamically, or both may change over time. All three cases have been studied extensively. For dynamic text with a static pattern, Amir et al. [4] presented an algorithm that preprocesses a text of length n and a pattern of length m in $\mathcal{O}(n \log \log m + m\sqrt{\log m})$ time, reporting all new occurrences of the pattern after each text update in $\mathcal{O}(\log \log m)$ time.

For static text with a dynamic pattern, efficient suffix tree based solutions were introduced by Weiner [43] and later improved by McCreight [34], Ukkonen [42], Farach [19], and others [30]. For a fixed finite alphabet, these algorithms preprocess the text in $\mathcal{O}(n)$ time and answer pattern queries in $\mathcal{O}(m + occ)$ time, where occ is the number of occurrences of the pattern. Finally, in the fully dynamic setting where both text and pattern may change, several algorithms have been developed [24, 21, 38, 3]. For example, Sahinalp and Vishkin [38] support insertions and deletions in $\mathcal{O}(\log^3 n + m)$ time per update, with query times comparable to Weiner’s static algorithm. Alstrup, Brodal, and Rauhe [3] handle insertions, deletions, and substring moves in $\mathcal{O}(\log^2 n \log \log n \log^* n)$ time per update, and answer pattern searches in $\mathcal{O}(\log n \log \log n + m + occ)$ time.

Our work extends these dynamic pattern matching studies to consider wildcards. We provide a randomized algorithm that, after $\mathcal{O}(n \log^2 n)$ preprocessing, supports updates and queries in amortized time $\tilde{\mathcal{O}}\left(kn^{\frac{k}{k+1}} + k^2 \log n\right)$, which is sublinear for $k = o(\log n)$ and truly sublinear for constant k . We also show that this bound is almost tight: under SETH, no algorithm with subquadratic preprocessing time can achieve truly sublinear update time when $k = \Omega(\log n)$. In addition to the general result, we design specialized algorithms for two restricted settings, with the complexity expressed in terms of the number of non-wildcard symbols in the pattern.

1. When the wildcard positions are fixed, we obtain preprocessing in $\mathcal{O}(nw)$ time and support updates in time $\mathcal{O}(w + \log n)$ time, where w is the number of non-wildcard symbols.
2. When the pattern contains at most two non-wildcard symbols, we give a deterministic algorithm based on FFT and block decomposition techniques, with preprocessing time $\mathcal{O}(n^{1.8})$ and update time $\mathcal{O}(n^{0.8} \log n)$.

Organization of the paper

Section 2 gives a high-level overview of our techniques and main results. In Section 3, we introduce formal definitions and preliminaries. The general dynamic pattern matching with wildcards setting is studied in Section 4, followed by algorithms for sparse pattern matching regimes in Section 5. Finally, in Section 6 we establish a conditional lower bound on the update time of any algorithm for the dynamic pattern matching with wildcards problem.

2 Results and Techniques

We study the problem of dynamic pattern matching in the presence of wildcard symbols. Our contributions fall into two main lines.

General setting. In the most flexible version of the problem, both the text and the pattern may contain wildcards, and updates may insert, delete, or substitute symbols (including wildcards). For this setting, we design randomized algorithms with preprocessing time $\mathcal{O}(n \log^2 n)$ and update/query time $\tilde{\mathcal{O}}(kn^{k/(k+1)} + k^2 \log n)$. The bound is truly sublinear for constant k , and remains sublinear whenever $k = o(\log n)$. This gives the first general framework for handling fully dynamic wildcards on both sides of the matching problem.

Sparse patterns. We also investigate regimes where the number of non-wildcards in the pattern is small. This additional structure enables more efficient algorithms:

- If the pattern contains at most two non-wildcard symbols, we obtain a deterministic algorithm based on FFT and block decomposition, with preprocessing time $\mathcal{O}(n^{1.8})$ and update time $\mathcal{O}(n^{0.8} \log n)$. Though the setting might seem simple, this is the most technical part of the paper, with the most challenging component captured by the *Range-Pair* problem (Problem 5.1), which we solve and use in this regime. The full description and proof of this algorithm are given in the full version.
- If all wildcards are confined to the pattern and their positions are fixed in advance, we design a randomized algorithm with preprocessing time $\mathcal{O}(nw)$ and update time $\mathcal{O}(w + \log n)$, where w is the number of non-wildcard positions.

Finally, we establish a conditional lower bound on the update and query time for any algorithm solving dynamic pattern matching with wildcards.

Together, these results give a unified picture of the problem's complexity. Figure 1 highlights the regimes where our upper and lower bounds apply. Also, for completeness and comparison, Table 1 summarizes the state-of-the-art results in pattern matching alongside our contributions. As shown in Table 1, when the pattern contains no wildcards, dynamic exact matching can be solved with $\mathcal{O}(n \log^2 n)$ preprocessing and polylogarithmic update time [14]. However, as noted, even a single wildcard can increase the number of matches.

Our approach builds on a simple but very useful observation. If the pattern P contains a symbol that appears only a few times in the text T , we can directly check those occurrences in the text and verify the aligned substrings (see Figure 2).

This leads us to a frequency-based classification of symbols: those that occur at most τ times in T are called *rare*, while the rest are considered *frequent*. When P contains a rare symbol, we verify all possible candidate matches that align that rare symbol.

Given this observation, we can assume without loss of generality that every symbol in the pattern is frequent. In this case, the number of candidate positions in the text that could match the pattern is large. The key insight, however, is that the number of distinct frequent symbols is limited to n/τ . This restriction allows us to design an alternative algorithm based on standard data structures.

■ **Table 1** Summary of Pattern Matching Results. For static problems, preprocessing refers to pattern preprocessing and query time refers to text scanning.

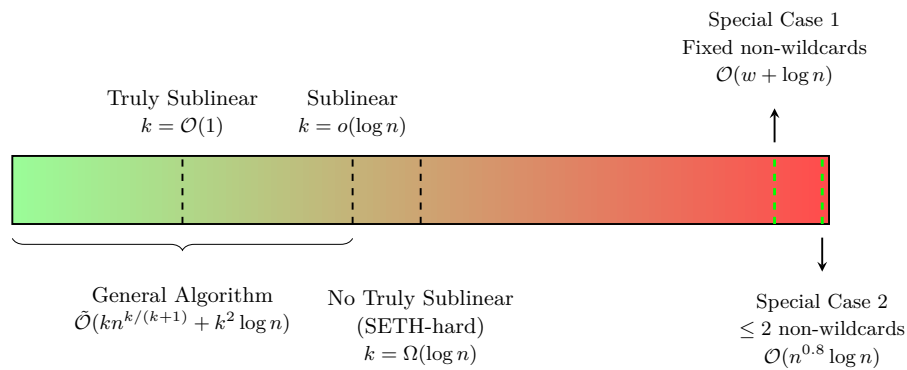
Problem Variant	Preprocessing Time	Algorithm/Query Time
Static Matching		
Exact Matching [32, 10]	$\mathcal{O}(m)$	$\mathcal{O}(n)$
Wildcard Matching [22, 26, 29]	$\mathcal{O}(m \log m)$	$\mathcal{O}(n \log m)$
k -mismatches(constant $ \Sigma $) [1, 16]	$\mathcal{O}(m \log m)$	$\mathcal{O}(n \log m)$
Dynamic Exact Matching		
Dynamic Text [4]	$\mathcal{O}(n \log \log m)$	$\mathcal{O}(\log \log m)$ per update
Fully Dynamic [3]	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\text{polylog } n)$ per query
Dynamic Matching with Wildcards (This Work)		
General (Randomized) [Theorem 4.1]	$\mathcal{O}(n \log^2 n)$	$\tilde{\mathcal{O}}(kn^{\frac{k}{k+1}} + k^2 \log n)$
Sparse Pattern (at most 2 non-wildcards) [Theorem 5.3]	$\mathcal{O}(n^{9/5})$	$\mathcal{O}(n^{4/5} \log n)$
Sparse Pattern (fixed wildcards) [Theorem 5.7]	$\mathcal{O}(wn)$	$\mathcal{O}(w + \log n)$
Conditional Lower Bound [Theorem 6.4]	–	No $\mathcal{O}(n^{1-\epsilon})$ for $k = \Omega(\log n)$

The main data structure we use in our algorithm is the *polynomial rolling hash table*. At a high level, it provides a succinct summary of substrings that (i) updates efficiently after a single-symbol change and (ii) lets us handle wildcards. We treat wildcards by partitioning the string into contiguous non-wildcard segments: the hash is computed only over these segments, and two strings match under the wildcard semantics if and only if all corresponding non-wildcard parts have equal hashes. In this way, wildcards are naturally ignored, and correctness reduces to verifying equality of the fixed pieces. To support dynamic updates, we maintain these hashes in balanced BSTs such as treaps [39], which allow insertions, deletions, and substitutions in logarithmic time, and combine them with fully dynamic LCS to handle candidate completions with polylogarithmic overhead [14].

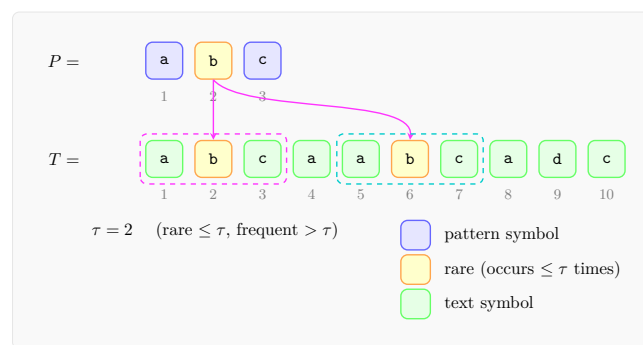
As we discussed earlier, our algorithm achieves truly sublinear update time when the number of wildcards is constant, and remains sublinear as long as the number of wildcards is $o(\log n)$. A natural question is whether these bounds can be improved further, perhaps matching the polylogarithmic update times known for the static case. Unfortunately, our results suggest otherwise. We show that once the number of wildcards reaches $\Omega(\log n)$, no algorithm with reasonable preprocessing can achieve truly sublinear update time with subquadratic preprocessing unless the Strong Exponential Time Hypothesis (SETH) fails. This hardness result follows from a reduction from the orthogonal vectors problem.

Given our results, several natural questions remain open:

- Can we design algorithms with truly sublinear update time when k is not constant but still $o(\log n)$? For constant k , can we further improve the update and query bounds?
- Can we obtain truly sublinear algorithms for cases where $k = \Omega(\log n)$, provided the wildcards have some special structure?



■ **Figure 1** The complexity landscape of fully dynamic pattern matching with k wildcards.



■ **Figure 2** Rare-symbol strategy: if the pattern contains a symbol that occurs at most τ times in the text (here, b), enumerate those occurrences and verify only the aligned substrings of length $|P|$.

The first direction appears particularly promising for future work. In line with the second, we provide algorithms achieving truly sublinear update times for certain restricted settings. We first consider the setting where the wildcard positions are fixed in advance and the number of non-wildcard symbols is sublinear. In this case, it suffices to maintain rolling hash values only for the $\mathcal{O}(k)$ non-wildcard positions, which enables sublinear update times for both the pattern and the text.

For the special case where the pattern contains at most two non-wildcard symbols, the static version of the problem can already be solved efficiently by applying FFT: by encoding blocks of the text as polynomials and using convolution, one can align the two characters across the text. In the dynamic setting, however, a single update can change many convolution values, so the FFT method alone is insufficient. To address this, we combine block decomposition with the frequent/rare symbol technique: the text is divided into blocks, only the affected blocks are recomputed after an update, rare symbols are treated explicitly, and frequent ones are handled in the convolution structures. This approach ensures that each update can be processed within sublinear time, achieving preprocessing complexity $\mathcal{O}(n^{\frac{9}{5}})$ and query complexity $\mathcal{O}(n^{\frac{4}{5}})$.

3 Preliminaries

We consider a finite alphabet Σ of size $|\Sigma| = \sigma$. Our input consists of two strings: a *text* $T \in (\Sigma \cup \{?\})^*$ of length n , and a *pattern* $P \in (\Sigma \cup \{?\})^*$ of length m . Both the text and the pattern together may contain at most k wildcard symbols, denoted by $?$. A wildcard symbol can match any symbol from the alphabet.

68:6 Dynamic Pattern Matching with Wildcards

For a string S and $1 \leq \ell \leq r \leq |S|$, we use $S_{\ell:r}$ to denote the substring $S_\ell S_{\ell+1} \dots S_r$. For example, if $S = abcde$, then $S_{2:4} = bcd$. We say that P *matches* a substring $T_{i:i+m-1}$ ($1 \leq i \leq n - m + 1$) if for every $1 \leq j \leq m$, either $P_j = T_{i+j-1}$, $P_j = ?$ or $T_{i+j-1} = ?$. The set of all matching positions is denoted by

$$\Gamma(P, T) = \{i \mid 1 \leq i \leq n - m + 1, P \text{ matches } T_{i:i+m-1}\}.$$

The concatenation of two strings S, S' , denoted $S \cdot S'$, is defined by appending S' to the end of S . For instance, if $S = ab$ and $S' = cd$, then $S \cdot S' = abcd$.

We classify symbols in Σ as *frequent* or *rare* according to a frequency threshold. Let the threshold value be

$$\tau = (n^k \log^7 n)^{\frac{1}{k+1}}.$$

A symbol $c \in \Sigma$ is called *rare* if it occurs fewer than τ times in T , and *frequent* otherwise.

For each symbol $c \in (\Sigma \cup ?)$, we maintain the set $\mathfrak{R}(c) = \{i \mid 1 \leq i \leq n, T_i = c\}$, which stores all positions where c appears in T . In addition to membership queries for $\mathfrak{R}(c)$, we also require efficient support for *lower bound queries*, i.e., finding the smallest index in $\mathfrak{R}(c)$ that is greater than or equal to a given value. For handling the updates and queries efficiently, we use balanced binary search trees.

► **Fact 3.1** (Balanced BST support). *Each set $\mathfrak{R}(c)$ can be maintained under dynamic text updates such that membership queries, lower bound queries, and updates are all supported in $\mathcal{O}(\log n)$ time, using a balanced binary search tree [18].*

Many of our algorithms are randomized and succeed with high probability. Throughout this paper, we say an event occurs **with high probability** if it holds with probability at least $1 - 1/n$.

The foundation for our analysis, based on polynomial hashing, is detailed in **Handling Matching Query** section in the full version. We use a polynomial rolling hash with a prime modulus p [31] to efficiently support matching queries. The hash function is formally defined as follows.

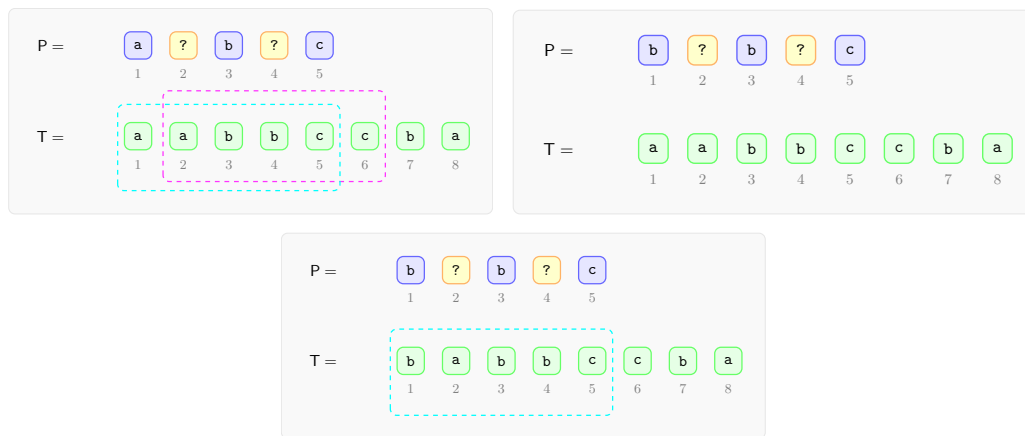
► **Definition 3.2** (Polynomial Rolling Hash). Let S be a string over an ordered alphabet $\Sigma = \{c_1, c_2, \dots, c_{|\Sigma|}\}$. We define a mapping $\pi : \Sigma \rightarrow \mathbb{N}$ such that $\pi(c_j) = j$ for $1 \leq j \leq |\Sigma|$. The *polynomial rolling hash* of S is

$$H_{b,p}(S) = \left(\sum_{i=1}^{|S|} \pi(S_i) \cdot b^{|S|-i} \right) \bmod p,$$

where b is chosen uniformly at random from $\{1, 2, \dots, p-1\}$, and p is a large prime modulus.

We leverage the following lemma from **Handling Matching Query** section to prove our probabilistic claims in the paper.

► **Lemma 3.3.** *Let $\{(S_1, S'_1), \dots, (S_N, S'_N)\}$ be a collection of N pairs of distinct strings, each string drawn from the alphabet Σ and of length at most x . Then the probability that there exists at least one pair (S_i, S'_i) such that $H_{b,p}(S_i) = H_{b,p}(S'_i)$ is bounded by $\frac{Nx}{p}$.*



■ **Figure 3** Pattern matching with $P = a?b?c$ and $T = aabbccba$. Yellow boxes represent wildcards. The red dashed region shows the match starting at position $i = 2$.

Dynamic Updates

As described earlier, both the pattern and the text may change dynamically. We support the following operations:

- **Pattern Update:** Insert, delete, or replace a symbol at position i in the pattern, where the new symbol $c \in \Sigma \cup \{?\}$ for $1 \leq i \leq m$.
- **Text Update:** Insert, delete, or replace a symbol at position i in the text, where the new symbol $c \in \Sigma \cup \{?\}$ for $1 \leq i \leq n$.

Updates are subject to the constraint that the total number of wildcards in P and T does not exceed k .

After each update, the query asks whether there exists an index i such that P matches the substring $T_{i:i+m-1}$, i.e., whether $\Gamma(P, T) \neq \emptyset$.

► **Example 3.4.** Let the alphabet be $\Sigma = \{a, b, c, d\}$ with $\sigma = 4$. Initially, the pattern is $P = a?b?c$ of length $m = 5$, and the text is $T = aabbccba$ of length $n = 8$. The pattern contains two wildcards ($P_2 = P_4 = ?$). The symbol occurrence sets in the text are as follows: $\mathfrak{R}(a) = \{1, 2, 8\}$, $\mathfrak{R}(b) = \{3, 4, 7\}$, $\mathfrak{R}(c) = \{5, 6\}$, $\mathfrak{R}(d) = \emptyset$. By definition, a match occurs at position i if $P_1 = T_i$, $P_3 = T_{i+2}$, and $P_5 = T_{i+4}$ (wildcards match any symbol). Checking $i = 1, \dots, 4$ gives $\Gamma(P, T) = \{1, 2\}$. Now, we modify the first symbol of the pattern from a to b . This produces the new pattern $P = b?b?c$, which still contains exactly two wildcard symbols. If we check the updated pattern against the text, we find that no substring of the text matches, so the set of occurrences becomes empty. Next, we change the first symbol of the text from a to b , resulting in $T = babbccba$. This modification updates the occurrence sets as follows: $\mathfrak{R}(a) = \{2, 8\}$, $\mathfrak{R}(b) = \{1, 3, 4, 7\}$, $\mathfrak{R}(c) = \{5, 6\}$, $\mathfrak{R}(d) = \emptyset$. Checking the updated text against the current pattern $b?b?c$ shows that a match now exists at position $i = 1$. See Figure 3.

3.1 Queries

Throughout our algorithm, we frequently use two types of queries: the *matching query* and the *longest common substring (LCS) query*. Below, we define each query and state the time complexity required to answer it.

Matching Query. The matching query is defined as $\text{is_match}(P_{\ell_1:r_1}, T_{\ell_2:r_2})$, which returns true if the pattern substring $P_{\ell_1:r_1}$ matches the text substring $T_{\ell_2:r_2}$, allowing wildcards in both P and T .

As we show in **Handling Matching Query** section in the full version, this query can be answered in time $\mathcal{O}(k' \log n)$ using hash techniques, where k' denotes the total number of wildcard symbols appearing in segments $P_{\ell_1:r_1}$ and $T_{\ell_2:r_2}$. Here we assume that dynamic updates consist only of substitutions, and do not include insertions or deletions.

► **Lemma 3.5.** *The query $\text{is_match}(P_{\ell_1:r_1}, T_{\ell_2:r_2})$ can be answered with $\mathcal{O}(n)$ preprocessing time, $\mathcal{O}(\log n)$ time per update operation, and $\mathcal{O}(k' \log n)$ time per query, where k' is the total number of wildcards in $P_{\ell_1:r_1}$ and $T_{\ell_2:r_2}$. The algorithm reports a match with probability 1 when one exists, and reports no match with probability $1 - \frac{1}{n^2}$ when no match exists.*

LCS Query. The LCS query is defined as $\text{LCS}(A, B)$, which returns the length of the longest common substring of two dynamic strings A and B . By [14], we know that this query can be answered with $\mathcal{O}(n \log^2 n)$ preprocessing time and $\mathcal{O}(\log^8 n)$ time per update.

► **Theorem 3.6** ([14]). *For strings A and B , each of length at most N , the query $\text{LCS}(A, B)$ can be supported with $\mathcal{O}(N \log^2 N)$ preprocessing time and $\mathcal{O}(\log^8 N)$ time per update operation. Here an update operation may be a substitution, insertion, or deletion applied to either A or B .*

4 Fully Dynamic Pattern Matching with Wildcards

In this section, we consider the general case of *dynamic pattern matching with wildcards problem*. Note that here both the text T and the pattern P may contain wildcard symbols. We assume that updates consist only of single-character substitutions, and do not include insertions or deletions. This simplifies the presentation of the main ideas. We will later explain in the full version how the algorithm extends to handle insertions and deletions as well.

Our solution is based on maintaining a modified version of the text, denoted by T' , where every rare symbol is replaced with a special placeholder “#”. Therefore, T' is defined as

$$T'_i = \begin{cases} \# & \text{if } T_i \text{ is rare,} \\ T_i & \text{if } T_i \text{ is frequent.} \end{cases}$$

Note that the frequency of symbols may change during the dynamic updates. Whenever the frequency of a symbol crosses the threshold τ , we update T' by replacing its occurrences with “#” (if it becomes rare) or restoring them to the original symbol (if it becomes frequent). To implement this, we maintain for each symbol $c \in \Sigma$ the occurrence set $\mathfrak{R}(c)$. When the text is updated at position i from c_{old} to c_{new} , we remove i from $\mathfrak{R}(c_{\text{old}})$ and insert it into $\mathfrak{R}(c_{\text{new}})$. If the frequency of a frequent symbol falls below τ , we mark its τ occurrences in T' with the placeholder “#”; conversely, if a rare symbol becomes frequent, we restore its actual symbol at those positions. Each such change involves only $\mathcal{O}(\tau)$ positions, giving an update cost of $\mathcal{O}(\tau)$. The full procedure is formalized in the full version.

The preprocessing phase (Algorithm 1) initializes the data structures, including building T' , maintaining occurrence sets $\mathfrak{R}(\cdot)$ for each symbol, and computing the frequent set F . The algorithm that, after each update, maintains these structures efficiently is explained in the full version.

■ **Algorithm 3** Case 2: No rare symbols in P (joint completions).

Input: Pattern P , modified text T' ; frequent set F ; total wildcards k across P and T'
Output: Whether P matches a substring of T
 // Enumerate joint completions over $F \cup \{\#\}$ in Gray-code order

```

1 foreach completion vector  $(a_1, \dots, a_k) \in (F \cup \{\#\})^k$  in Gray-code order do
2   Apply the single-coordinate change implied by  $(a_1, \dots, a_k)$  to obtain  $(\tilde{P}, \tilde{T})$ ;
3   if  $LCS(\tilde{P}, \tilde{T}) = m$  then
4     return Match Found
5 return No Match Found
```

■ **Algorithm 4** Main Algorithm: Dynamic Pattern Matching with Wildcards.

Input: Pattern P of length m , text T of length n , stream of updates
Output: After each update, whether P matches a substring of T

```

1 Preprocess( $P, T$ ) ; // One-time preprocessing (Alg. 1)
2 foreach update on  $P$  or  $T$  do
3   Maintain(current update) ; // Maintain occurrences and  $T'$  in  $\mathcal{O}(\tau)$ 
4   if  $P$  contains a rare symbol then
5     // Case 1 (Alg. 2)
6     if CaseOne( $P, T$ ) is true then
7       return Match Found
8     else
9       return Match Not Found
9   else
10    // Case 2 (Alg. 3)
11    if CaseTwo( $P, T', F, k$ ) is true then
12      return Match Found
13    else
14      return No Match Found
```

5 Sparse Pattern

In this section we study *sparse* patterns, namely patterns $P \in (\Sigma \cup \{?\})^m$ in which only a few positions are non-wildcard and the remaining positions are wildcards. Let $S = \{i \in [m] : P_i \in \Sigma\}$ be the set of non-wildcard indices and let $s = |S| \ll m$. Such sparse patterns naturally arise in applications such as biological sequence analysis, where only a few positions of a motif are fixed while the rest may vary, or in intrusion detection, where signatures often contain only a few fixed bytes. In this section we consider two regimes: (i) the case where the pattern contains at most two non-wildcard symbols, and (ii) the case where the wildcard positions in the pattern are fixed in advance and the text T contains no wildcards. Throughout this section we work in a *substitution-only* dynamic model: updates may change symbols in the text or in the pattern, but no insertions or deletions are allowed, so both lengths n and m (and, in the fixed-wildcard regime, the wildcard index set) remain fixed.

5.1 At Most Two Non-Wildcard Symbols

In this section, we study a special case of *Dynamic Pattern Matching with Wildcards* under the additional assumption that the pattern P contains at most two non-wildcard symbols at any time. We focus on this substitution-only regime of the problem, and the text T contains no wildcards. But in the remarks we will explain how the algorithm can be extended to handle insertions/deletions in the pattern as well as wildcards in the text. Moreover, in this regime we are also able to count the exact number of matches in the same asymptotic complexity. For consistency with the other sections we only present the decision version in the main algorithm, but counting can also be supported. The detailed explanations of these remarks are given in the full version. For this setting, we design an algorithm with preprocessing time $\mathcal{O}(n^{9/5})$, pattern update time $\mathcal{O}(1)$, text update time $\mathcal{O}(n^{4/5} \log n)$, and query time $\mathcal{O}(n^{4/5})$.

A general observation in sparse patterns is that leading or trailing runs of $?$ can be ignored when reasoning about matches. These wildcards only shift the range of permissible starting positions but do not affect the relative positions among the concrete symbols. Consequently, we can conceptually divide any pattern into three parts: a prefix of wildcards, a block of concrete symbols (the *query block*), and a suffix of wildcards:

$$\underbrace{? \dots ?}_{\text{prefix wildcards}} \quad \underbrace{c_1 \dots c_k}_{\text{query block}} \quad \underbrace{? \dots ?}_{\text{suffix wildcards}} .$$

After removing the leading and trailing wildcards, let the remaining concrete positions in P be $i_1 < i_2 < \dots < i_k$ with symbols c_1, c_2, \dots, c_k . A substring $T_{s:s+m-1}$ matches P if and only if

$$T_{s+i_j-1} = c_j \quad \text{for all } j = 1, \dots, k.$$

Equivalently, a match occurs when the concrete symbols appear in the text at positions that preserve the relative offsets $i_{j+1} - i_j$. This perspective allows match queries to be reduced to checking whether these symbols appear together at the required distances within the allowed range of starting positions.

In the special case where the pattern P contains at most two non-wildcard symbols, the matching task reduces to checking whether two symbols occur in the text at a specific distance from each other. This motivates the following formal problem.

► **Problem 5.1** (Dynamic Range-Pair Query). Let Λ be an alphabet of size λ . The input is a dynamic string $\mathcal{T} \in \Lambda^n$. The task is to support the following operations:

- **UpdateSymbol**(i, c): update the symbol at position i to $c \in \Lambda$.
- **PairQuery**(l, r, a, b, d): return the number of indices $i \in [l, r - d - 1]$ such that $\mathcal{T}_i = a$ and $\mathcal{T}_{i+d+1} = b$, where $a, b \in \Lambda$ and $d \geq 0$.

This setting is closely related to the classical *Gapped String Indexing* problem [9]. In Gapped String Indexing, one preprocesses a static string S and answers queries given by two patterns P_1, P_2 and a gap interval $[\alpha, \beta]$, asking whether there exist occurrences at positions i and $j \geq i$ such that $j - i \in [\alpha, \beta]$ [9]. In the special case where $|P_1| = |P_2| = 1$ and $\alpha = \beta = d + 1$, this reduces to deciding or counting occurrences of two symbols whose positions differ by exactly $d + 1$. The Dynamic Range-Pair Query problem captures this special case while additionally restricting valid pairs to lie within a query range $[l, r]$, and allowing the underlying text to be updated dynamically. Viewed this way, the problem can also be interpreted as a dynamic, range-restricted variant of *Shifted Set Intersection* (equivalent to 3SUM indexing in the static setting) [9, 7].

68:12 Dynamic Pattern Matching with Wildcards

■ **Algorithm 5** Preprocessing for Range-Pair Queries.

Input: String $\mathcal{T}_{1..n}$, block size B
Output: Tables `Near`, `Pair`, snapshot $\bar{\mathcal{T}}$, pending sets `Pendingi`

- 1 Partition indices into $M = \lceil \frac{n}{\beta} \rceil$ blocks ;
- 2 Set $\bar{\mathcal{T}} \leftarrow \mathcal{T}$ and `Pendingi` $\leftarrow \emptyset$ for all blocks i ; // $\mathcal{O}(\eta)$
- 3 Build `Neari,a,b,d` for all blocks i , symbols $a, b \in \Lambda$, and distances $0 \leq d < \beta$; // $\mathcal{O}(\eta \cdot \beta)$
- 4 Build `Pairi,j,a,b,d'` for all block pairs $i < j$, symbols $a, b \in \Lambda$, and $0 \leq d' \leq 2\beta - 2$ (using FFT on block pairs) ; // $\mathcal{O}\left(\frac{\eta^2}{\beta} \log \beta \cdot \lambda^2\right)$

■ **Algorithm 6** `PairQuery(l, r, a, b, d)`.

Input: Range $[l, r]$, symbols $a, b \in \Lambda$, distance $d \geq 0$
Output: Integer `cnt` = number of $i \in [l, r - d - 1]$ with $\mathcal{T}_i = a$ and $\mathcal{T}_{i+d+1} = b$

- 1 `cnt` $\leftarrow 0$;
- 2 **if** $d < \beta$ **then**
- 3 **foreach** *full block i inside $[l, r - d - 1]$* **do**
- 4 `cnt` \leftarrow `cnt` + `Neari,a,b,d`;
- 5 **else**
- 6 **foreach** *full block i inside $[l, r - d - 1]$* **do**
- 7 compute the target blocks j and $j+1$ together with their respective offsets d' and d'' ;
- 8 `cnt` \leftarrow `cnt` + `Pairi,j,a,b,d'` + `Pairi,j+1,a,b,d''`;
- 9 apply corrections for `Pendingi` \cup `Pendingj` \cup `Pendingj+1`;
- 10 **foreach** *index i in the two boundary partial blocks of $[l, r - d - 1]$* **do**
- 11 **if** $\mathcal{T}_i = a$ and $\mathcal{T}_{i+d+1} = b$ **then**
- 12 `cnt` \leftarrow `cnt` + 1
- 13 **return** `cnt`;

To tackle this problem, we use a block decomposition approach. The text is partitioned into consecutive blocks, and for each block we maintain precomputed counts of symbol pairs whose left endpoint lies inside the block. Updates are handled lazily: when only a small number of positions in a block are modified, we record these changes explicitly and leave the precomputed counters unchanged; once the number of pending changes exceeds a fixed threshold, the block is rebuilt from scratch. For a query range $[l, r]$, blocks that lie entirely inside the query interval – referred to as *full blocks* – are answered using their precomputed counters, with a correction for any pending updates stored for those blocks. Only positions belonging to blocks that intersect the query interval partially – namely, the boundary blocks at the left and right ends of the range – are handled by explicit scanning. Pseudocode of Range-Pair data structure is available in this version, but for the full algorithm explanation and technical details of the data structure, please see the full version.

► **Theorem 5.2.** *By combining the preprocessing procedure (Algorithm 5), the update procedure (Algorithm 7), and the query procedure (Algorithm 6), the data structure answers the Dynamic Range-Pair Query Problem (Problem 5.1) on $\mathcal{T} \in \Lambda^n$ with:*

- Preprocessing time $\mathcal{O}\left(\frac{\eta^2}{\beta} \log \beta \cdot \lambda^2 + \eta\beta\right)$,
- per-query time $\mathcal{O}\left(\frac{\eta}{\beta} \cdot \mu + \beta\right)$,

■ **Algorithm 7** Update Operation.

Input: Position pos , new symbol $c \in \Lambda$
Output: Updated data structures

- 1 $\mathcal{T}_{\text{pos}} \leftarrow c$;
- 2 Let b be the index of the block containing pos ;
- 3 **if** $\mathcal{T}_{\text{pos}} \neq \overline{\mathcal{T}}_{\text{pos}}$ **then**
- 4 insert pos into Pending_b ;
- 5 **else**
- 6 remove pos from Pending_b ;
- 7 Update all $\text{Near}_{i,a,b,d}$ entries affected by pos for $d < \beta$; // $\mathcal{O}(\beta)$
- 8 **if** $|\text{Pending}_b| > \mu$ **then**
- 9 Overwrite $\overline{\mathcal{T}}$ in block b with the current \mathcal{T} values; // $\mathcal{O}(\beta)$
- 10 $\text{Pending}_b \leftarrow \emptyset$;
- 11 Recompute all Pair slices with one endpoint equal to b ; // $\mathcal{O}(\eta \log \beta \cdot \lambda^2)$

■ **Algorithm 8** Preprocess for At-Most-Two Non-Wildcard Symbols.

Input: Text $\mathbb{T} \in \Sigma^n$, threshold τ , block size β
Output: Mapped text $\mathcal{T} \in \Lambda^n$, map ϕ ; Range-Pair structures on \mathcal{T}

- 1 Classify symbols of Σ into frequent/rare using τ ; // $\mathcal{O}(n)$
- 2 Define $\Lambda = \{0, 1, \dots, M\}$ with $M := \lceil 2n/\tau \rceil$; set $\phi(x) = 0$ for rare x , and a nonzero code in $\{1, \dots, M\}$ for frequent x ; // $\mathcal{O}(\sigma)$
- 3 Build $\mathcal{T}_t \leftarrow \phi(\mathbb{T}_t)$ for all $t = 1..n$; // $\mathcal{O}(n)$
- 4 Run Range-Pair preprocessing algorithm on \mathcal{T} ; // $\mathcal{O}\left(\frac{\eta^2}{\beta} \log \beta \cdot \lambda^2 + \eta\beta\right)$

■ *amortized update time* $\mathcal{O}\left(\beta + \frac{\eta \log \beta \cdot \lambda^2}{\mu}\right)$,

■ *space usage* $\mathcal{O}\left(\frac{\eta^2}{\beta} \lambda^2\right)$.

Here β and μ are configurable parameters, fixed in advance, that must satisfy $\mu \leq \beta \leq \eta$.

Now for the main problem, the algorithm distinguishes the possible cases depending on how many non-wildcard symbols appear in the pattern. If the pattern contains a rare symbol, then all its occurrences in the text can be inspected directly, and since at most one further non-wildcard position needs to be checked, the query runs in $\mathcal{O}(\tau)$ time. If the pattern contains only frequent symbols, three subcases arise. With no non-wildcards the pattern trivially matches any substring of length m (a constant-time check). With exactly one non-wildcard, the query reduces to checking whether that symbol appears in the valid range of positions, which can be done using the set $\mathfrak{R}(c)$ in $\mathcal{O}(\log n)$ time. Finally, with two non-wildcards the problem reduces to asking whether two symbols appear in the text at the required distance, which is exactly the Range-Pair Problem defined earlier. To support this reduction we maintain a mapping $\phi: \Sigma \rightarrow \Lambda$ that assigns all rare symbols to 0 and gives each frequent symbol a distinct nonzero code, so that the mapped text \mathcal{T} serves as input to the Range-Pair structure. The procedures for preprocessing, queries, and updates are summarized in Algorithms 8, 9, and 10. By Theorem 5.2, with parameters $\beta = n^{4/5}$, $\mu = n^{3/5}$, and $\tau = n^{4/5}$, the resulting bounds match those stated in Theorem 5.3. For the complete algorithms and detailed calculations we refer to the proof in the full version.

■ **Algorithm 9** Query for At-Most-Two Non-Wildcard Symbols.

Input: Pattern $P \in (\Sigma \cup \{?\})^m$, text length n , map ϕ
Output: Boolean: does P occur in T ?

```

1 Let  $C$  be the set of non-wildcard positions of  $P$ ; assume  $m \leq n$ ;
2 if  $|C| = 0$  then
3   return true ; //  $\mathcal{O}(1)$ 
4 else if  $|C| = 1$  then
5   Let  $C = \{i\}$  with symbol  $a$ ; let  $I \leftarrow [i, n - m + i]$ ;
6   return whether  $\mathfrak{R}(a)$  contains an index in  $I$  ; //  $\mathcal{O}(\log n)$ 
7 else // Two non-wildcard symbols
8   Let  $C = \{i < j\}$  with symbols  $a, b$ ; let  $d \leftarrow j - i - 1$ ; let  $l \leftarrow i, r \leftarrow n - m + i$ ;
9   if  $a$  is rare then
10    foreach  $p \in \mathfrak{R}(a)$  with  $p \in [l, r]$  do
11      if  $T_{p+d+1} = b$  then
12        return true
13    return false ; //  $\mathcal{O}(\tau)$ 
14   else if  $b$  is rare then
15    foreach  $q \in \mathfrak{R}(b)$  with  $q - d + 1 \in [l, r]$  do
16      if  $T_{q-d+1} = a$  then
17        return true
18    return false ; //  $\mathcal{O}(\tau)$ 
19   else
20    return PairQuery( $l, r, \phi(a), \phi(b), d$ ) > 0 ; //  $\mathcal{O}\left(\frac{n}{\beta} \cdot \mu + \beta\right)$ 

```

► **Theorem 5.3.** *Combining the preprocessing in Algorithm 8, the query procedure in Algorithm 9, and the text/pattern updates in Algorithm 10, the data structure answers the at-most-two non-wildcard dynamic wildcard matching variant with the following guarantees (for $T \in \Sigma^n$):*

- preprocessing time $\mathcal{O}(n^{\frac{9}{5}})$,
- pattern-update time $\mathcal{O}(1)$,
- text-update time $\mathcal{O}(n^{\frac{4}{5}} \log n)$ (amortized),
- per-query time $\mathcal{O}(n^{\frac{4}{5}})$.

5.2 Fixed Wildcard Positions in Pattern

In this subsection we study the dynamic pattern matching problem with fixed wildcards: all k wildcard symbols appear only in the pattern P , their index set $W = \{i \in [m] : P_i = ?\}$ is known in advance and immutable, and the text T contains no wildcards. Both P and T evolve only via substitution updates, and insertions and deletions are not allowed; therefore updates to P are confined to the solid coordinates $[m] \setminus W$, while updates to T may change any $T_i \in \Sigma$. We parameterize by the number of non-wildcard positions $\omega = |[m] \setminus W|$ and target the sparse regime $\omega \ll m$. To handle this setting, we adopt a masked polynomial rolling hash that ignores the fixed wildcard indices: equivalently, we view P as a concatenation of solid blocks separated by wildcards and compute the hash only over these blocks, enabling comparison against substrings of T under substitutions while treating wildcard coordinates as don't-cares; this leads to the following definition.

■ **Algorithm 10** Update operations for At-Most-Two Non-Wildcard Symbols.

```

1 Pattern update ( $i, x$ ):
2    $P_i \leftarrow x$ ; //  $\mathcal{O}(1)$ 
3   Maintain the set  $C$  of up to two non-wildcard positions of  $P$ ; //  $\mathcal{O}(1)$ 
4 Text update ( $i, x$ ):
5    $T_i \leftarrow x$ ;
6   maintain the frequency class of  $x$  as per threshold  $\tau$  (promotion/demotion);
   // amort.  $\mathcal{O}(1)$  per symbol
7   Perform the range-pair update on index  $i$  with new symbol  $\phi(x)$ ;
   // amort.  $\mathcal{O}\left(\beta + \frac{n \log \beta \cdot \lambda^2}{\mu}\right)$ 

```

► **Definition 5.4.** Let S be a string of length ℓ , and let P be a pattern with k wildcard positions $w_1 < w_2 < \dots < w_k$, where each w_i is the index of the i -th wildcard in P . We define

$$H'_{b,p}(S) = H_{b,p}(S_{1:w_1-1} \cdot S_{w_1+1:w_2-1} \cdot \dots \cdot S_{w_k+1:\ell}),$$

where $H_{b,p}(\cdot)$ is the polynomial rolling hash from Definition 3.2.

The value $H'_{b,p}(S)$ can be computed in $\mathcal{O}(k)$ time by concatenating the precomputed hash values of the $k+1$ non-wildcard intervals. Moreover, after each update in S , the value of $H'_{b,p}(S)$ can be updated in $\mathcal{O}(1)$ time. Let us now define an auxiliary vector A , where

$$A_i = H'_{b,p}(T_{i:i+m-1}) \quad \text{for } 1 \leq i \leq n - m + 1,$$

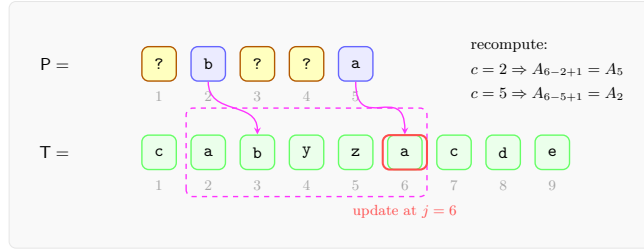
so that A_i stores the hash of the length- m substring of T starting at position i , with the fixed wildcards of P ignored.

► **Lemma 5.5.** *If the pattern P decomposes into l non-empty intervals separated by its wildcard positions, then the vector A can be constructed in $\mathcal{O}(n \cdot l)$ time.*

Together, Definition 5.4 and Lemma 5.5 provide the key ingredients for efficient preprocessing in the sparse-pattern setting. The modified hash $H'_{b,p}(S)$ ensures that wildcards are ignored, while the vector A stores precomputed values for all substrings of T that need to be compared against P . The lemma guarantees that this preprocessing can be done in $\mathcal{O}(n \cdot l)$ time, where l is the number of non-empty intervals in P . Algorithm 11 then presents the full method for handling updates. Intuitively, when the pattern has only a few fixed symbols and the rest are wildcards, we can ignore wildcard positions in all computations. Let $C = \{c_1 < \dots < c_\omega\}$ be the indices of the non-wildcard positions in P . For every window $T_{i:i+m-1}$ of the text, we compute $A_i = H'_{b,p}(T_{i:i+m-1})$ based only on the symbols aligned with C . We also maintain the single value $H'_{b,p}(P)$. A match exists iff $H'_{b,p}(P)$ appears among $\{A_1, \dots, A_{n-m+1}\}$; this can be checked efficiently with a membership query, e.g., in $\mathcal{O}(\log n)$ time using a balanced BST.

If the pattern changes at a non-wildcard position, we simply recompute $H'_{b,p}(P)$ and test membership once. If the text changes at position j , only those windows whose alignment uses j at one of the non-wildcard positions can be affected. Equivalently, for each $i \in C$ with $j \geq i$, only A_{j-i+1} must be recomputed. Thus a single text update triggers at most $|C| = \omega$ window-hash updates.

68:16 Dynamic Pattern Matching with Wildcards



■ **Figure 4** Sparse-pattern example for $P = ?b??a$ and $T = cabyzacde$. The dashed magenta box highlights the alignment at $i = 2$ (substring $T_{2:6} = abyza$); arrows show the two non-wildcard checks at pattern positions 2 and 5. The red rectangle marks the text update at $j = 6$ ($a \rightarrow x$), which invalidates the match at $i = 2$ and forces recomputation of A_5 and A_2 (as indicated on the right).

■ **Algorithm 11** Updates with Preprocessed Data (Sparse Pattern Matching).

Input : Current T, P ; preprocessed A, C ; lengths n, m
Output : After each update, report whether P matches a substring of T

- 1 Apply the substitution in the pattern;
- 2 **if** $H'_{b,p}(P) \in \{A_1, \dots, A_{n-m+1}\}$ **then**
- 3 **return** Match Found
- 4 **return** No Match Found

// (B) Text update at position j : $T_j \leftarrow c_{\text{new}}$

- 5 Apply the substitution in the text;
- 6 **foreach** $i \in C$ **with** $j \geq i$ **do**
- // Recompute only the affected window A_{j-i+1} if it is in range
- 7 **if** $1 \leq j - i + 1 \leq n - m + 1$ **then**
- 8 $A_{j-i+1} \leftarrow H'_{b,p}(T_{j-i+1:j-i+m})$;
- 9 **if** $H'_{b,p}(P) \in \{A_1, \dots, A_{n-m+1}\}$ **then**
- 10 **return** Match Found
- 11 **return** No Match Found

► **Example 5.6.** Let $P = ?b??a$ with $m = 5$ and non-wildcard indices $C = \{2, 5\}$. Let $T = cabyzacde$ with $n = 9$. Consider the alignment starting at $i = 2$ (i.e., substring $T_{2:6} = abyza$). The two non-wildcard checks are:

$$P_2 = b \stackrel{?}{=} T_{2+2-1} = T_3 = b, \quad P_5 = a \stackrel{?}{=} T_{2+5-1} = T_6 = a,$$

so this alignment matches. Accordingly, $A_2 = H'_{b,p}(T_{2:6})$ equals $H'_{b,p}(P)$.

Now suppose we update the text at position $j = 6$, changing T_6 from a to x . Only the window hashes whose aligned non-wildcard positions use j can change: for $i = 2$ we must recompute $A_{j-c+1} = A_5$; for $i = 5$ we must recompute $A_{j-i+1} = A_2$. In particular, A_2 will no longer equal $H'_{b,p}(P)$ since T_6 no longer matches $P_5 = a$. See Figure 4 for a visualization of the alignment at $i = 2$, the update at $j = 6$, and the affected window hashes.

► **Theorem 5.7.** *Algorithm 11 maintains the dynamic pattern matching problem with fixed wildcards for a pattern P of length m containing at most ω non-wildcard positions. It requires preprocessing time $\mathcal{O}(n \cdot \omega)$ and supports updates to either P or T in $\mathcal{O}(\omega + \log n)$ time per update. The algorithm always detects a match correctly when one exists. When no match exists, it reports “no match” with high probability.*

The general statement above can be specialized to different regimes of sparsity. First, if the number of non-wildcard positions is bounded by a sublinear power of n , we obtain the following corollary.

► **Corollary 5.8.** *If the number of non-wildcard positions satisfies $\omega \leq n^{1-\delta}$ for some constant $0 < \delta < 1$, then Algorithm 11 runs in preprocessing time $\mathcal{O}(n^{2-\delta})$ and supports updates in $\mathcal{O}(n^{1-\delta})$ time.*

Moreover, since the number of non-wildcard positions can never exceed the pattern length m , we can directly derive the following bound.

► **Corollary 5.9.** *Since $\omega \leq m$, the algorithm also admits the trivial upper bound obtained by setting $\omega = m$. In this case, Algorithm 11 requires preprocessing time $\mathcal{O}(n \cdot m)$ and update time $\mathcal{O}(m + \log n)$.*

6 Hardness of Dynamic Pattern Matching with Wildcards

In this section, we establish a conditional lower bound for Dynamic Pattern Matching with Wildcards through a reduction from the Orthogonal Vectors problem. Our goal is to show that any significant improvement in the query time for Dynamic Pattern Matching with Wildcards would imply a breakthrough for Orthogonal Vectors, which is widely believed to be unlikely under the Strong Exponential Time Hypothesis (SETH).

► **Problem 6.1** (The Orthogonal Vectors Problem). Given a set A of n vectors from $\{0, 1\}^d$, the Orthogonal Vectors problem asks whether there exists a pair of distinct vectors $U, V \in A$ such that their inner product is zero:

$$\sum_{h=1}^d U_h \cdot V_h = 0.$$

That is, no coordinate position contains a 1 in both U and V .

Williams [44] establishes the following connection between Orthogonal Vectors and SETH.

► **Theorem 6.2** ([44]). *If there exists $\varepsilon > 0$ such that, for all constants c , the Orthogonal Vectors problem on a set of n vectors of dimension $d = c \log n$ can be solved in $2^{o(d)} \cdot n^{2-\varepsilon}$ time, then SETH is false.*

We now show a reduction of Orthogonal Vectors to Dynamic Pattern Matching with Wildcards. This reduction forms the backbone of our lower bound.

► **Lemma 6.3.** *An instance of Orthogonal Vectors with a set A of n vectors in dimension d can be reduced to an instance of Dynamic Pattern Matching with Wildcards with pattern length $m = d + 2$ and text length $\mathcal{O}(n \cdot d)$. If Dynamic Pattern Matching with Wildcards can be solved with preprocess time $h(n)$ and query time $g(n)$, then Orthogonal Vectors can be solved in $\mathcal{O}(h(d \cdot n) + d \cdot n \cdot g(d \cdot n))$ time.*

Proof. Let $A = \{V_1, V_2, \dots, V_n\}$, where each V_i is a d -dimensional binary vector. For each vector V_i , construct a modified vector V_i' by replacing every 1 in V_i with 0, and every 0 in V_i with a wildcard symbol. Define the pattern and text as

$$P = \#V_1'\#, \quad T = \#V_1\#V_2\#\dots\#V_n\#,$$

where $\#$ is a delimiter not in $\{0, 1, ?\}$. By construction, P has length $m = d + 2$ and T has length $\mathcal{O}(n \cdot d)$. We claim that P matches a substring of T if and only if the Orthogonal Vectors instance contains an orthogonal pair. Indeed, suppose $P = \#V_i\#$ matches $\#V_j\#$ in T . Then, for each coordinate k :

- If $V_{i,k} = 1$, then $V'_{i,k} = 0$. Since 0 can only match 0, we must have $V_{j,k} = 0$.
- If $V_{i,k} = 0$, then $V'_{i,k} = ?$, which matches both 0 and 1.

Thus, there is no coordinate where both $V_{i,k} = 1$ and $V_{j,k} = 1$, which means V_i and V_j are orthogonal. The converse follows by reversing this reasoning.

To check all pairs, we iteratively transform the pattern into each $\#V_i\#$ for $i = 1, \dots, n$. Each transformation requires at most d symbol changes, followed by a query to check if the pattern matches a substring of T . Over all n vectors, this yields $\mathcal{O}(d \cdot n)$ updates and queries. Each query costs $g(d \cdot n)$, so the total runtime is $\mathcal{O}(d \cdot n \cdot g(d \cdot n))$. ◀

The reduction allows us to transfer lower bounds from Orthogonal Vectors to Dynamic Pattern Matching with Wildcards.

To keep the presentation concise, we defer the proof of Theorem 6.4 to the full version.

► **Theorem 6.4.** *Assuming SETH holds, the fully dynamic wildcard pattern matching problem with $k = \Omega(\log n)$ wildcards cannot be solved with preprocessing time $\mathcal{O}(n^{2-\delta})$ and query time $\mathcal{O}(n^{1-\varepsilon})$ for any constants $\delta, \varepsilon > 0$.*

References

- 1 Karl Abrahamson. Generalized string matching. *SIAM journal on Computing*, 16(6):1039–1051, 1987. doi:10.1137/0216067.
- 2 V. Abrishami, A. Zaldívar-Peraza, J. M. de la Rosa-Trevín, J. Vargas, J. Otón, R. Marabini, Y. Shkolnisky, J. M. Carazo, and C. O. S. Sorzano. A pattern matching approach to the automatic selection of particles from low-contrast electron micrographs. *Bioinformatics*, 29(19):2460–2468, 2013. doi:10.1093/BIOINFORMATICS/BTT429.
- 3 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 819–828, 2000. URL: <http://dl.acm.org/citation.cfm?id=338219.338645>.
- 4 Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms (TALG)*, 3(2):19–es, 2007.
- 5 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.
- 6 Mika Amit, Rolf Backofen, Steffen Heyne, Gad M. Landau, Mathias Möhl, Christina Otto, and Sebastian Will. Local exact pattern matching for non-fixed rna structures. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(1):219–230, 2014. doi:10.1109/TCBB.2013.2297113.
- 7 Boris Aronov, Jean Cardinal, Justin Dallant, and John Iacono. A general technique for searching in implicit sets via function inversion. In *2024 Symposium on Simplicity in Algorithms (SOSA)*, pages 215–223. SIAM, 2024. doi:10.1137/1.9781611977936.20.
- 8 Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern matching with mismatches and wildcards. *arXiv preprint arXiv:2402.07732*, 2024. doi:10.48550/arXiv.2402.07732.
- 9 Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P Pissis, Eva Rotenberg, and Teresa Anna Steiner. Gapped string indexing in subquadratic space and sublinear query time. *arXiv preprint arXiv:2211.16860*, 2022. doi:10.48550/arXiv.2211.16860.
- 10 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977. doi:10.1145/359842.359859.

- 11 Vincent Bushong, Russell Sanders, Jacob Curtis, Mark Du, Tomas Cerny, Karel Frajtak, Miroslav Bures, Pavel Tisnovsky, and Dongwan Shin. On matching log analysis to source code: A systematic mapping study. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS)*, pages 181–187, 2020.
- 12 J. V. Capacho, A. Subias, L. Trave-Massuyès, and F. Jimenez. Alarm management via temporal pattern learning. *Engineering Applications of Artificial Intelligence*, 65:506–516, 2017. doi:10.1016/J.ENGAPPAI.2017.07.008.
- 13 Timothy M Chan, Ce Jin, Virginia Vassilevska Williams, and Yinzhan Xu. Faster algorithms for text-to-pattern hamming distances. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2188–2203. IEEE, 2023. doi:10.1109/FOCS57990.2023.00136.
- 14 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic Longest Common Substring in Polylogarithmic Time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:19, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2020.27.
- 15 Y. Cheng, I. Izadi, and T. Chen. Pattern matching of alarm flood sequences by a modified smith–waterman algorithm. *Chemical Engineering Research and Design*, 91(6):1085–1094, 2013.
- 16 Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi:10.1016/J.IPL.2006.08.002.
- 17 Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 592–601. ACM, 2002. doi:10.1145/509907.509992.
- 18 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.
- 19 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.
- 20 Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys*, 45(2), 2013. doi:10.1145/2431211.2431212.
- 21 Paolo Ferragina and Roberto Grossi. Fast incremental text editing. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 531–540, 1995. URL: <http://dl.acm.org/citation.cfm?id=313651.313815>.
- 22 Michael J. Fischer and Michael S. Paterson. String matching and other products. In Richard M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125, Providence, RI, 1974. AMS.
- 23 Zvi Galil and Kunsoo Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990. doi:10.1137/0219067.
- 24 Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 697–704, 1994. URL: <http://dl.acm.org/citation.cfm?id=314464.314675>.
- 25 Saqib Iqbal Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan Amin Gilkar, Wazir Zada Khan, and Muhammad Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE Access*, 7:69614–69637, 2019. doi:10.1109/ACCESS.2019.2914071.
- 26 Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *39th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 166–173, Washington, DC, USA, 1998. IEEE Computer Society. doi:10.1109/SFCS.1998.743440.

- 27 Ce Jin and Yinzhan Xu. Shaving logs via large sieve inequality: Faster algorithms for sparse convolution and more. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 1573–1584, 2024. doi:10.1145/3618260.3649605.
- 28 Michael C. Johannesmeyer, Ashish Singhal, and Dale E. Seborg. Pattern matching in historical data. *AIChE Journal*, 48(9):2022–2038, 2002.
- 29 Adam Kalai. Efficient pattern-matching with don't cares. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–656, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=545381.545468>.
- 30 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.
- 31 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/RD.312.0249.
- 32 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 33 Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1988. doi:10.1016/0196-6774(89)90010-2.
- 34 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
- 35 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 36 Peyman Neamatollahi, Montassir Hadi, and Mahmoud Naghibzadeh. Simple and efficient pattern matching algorithms for biological sequences. *IEEE Access*, 8:23838–23846, 2020. doi:10.1109/ACCESS.2020.2969038.
- 37 P. Pandiselvam, T. Marimuthu, and R. Lawrance. A comparative study on string matching algorithms of biological sequences. *arXiv preprint arXiv:1401.7416*, 2014. arXiv:1401.7416.
- 38 Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 320–328. IEEE, 1996.
- 39 Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996. doi:10.1007/BF01940876.
- 40 Muhammad Tahir. Efficient pattern matching algorithm for dna sequences. *Expert Systems with Applications*, 82:96–112, 2017.
- 41 Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985. doi:10.1016/S0019-9958(85)80046-2.
- 42 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 43 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.
- 44 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.