Dynamic Data Structures for Parameterized String Problems

Jędrzej Olkowski
Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Poland

Michał Pilipczuk
Institute of Informatics, University of Warsaw, Poland

Mateusz Rychlicki
School of Computing, University of Leeds, UK

Karol Węgrzycki
Saarland University, Saarbrücken, Germany
Max Planck Institute for Informatics, Saarbrücken, Germany

Anna Zych-Pawlewicz
Institute of Informatics, University of Warsaw, Poland

Abstract

We revisit classic string problems considered in the area of parameterized complexity, and study them through the lens of dynamic data structures. That is, instead of asking for a static algorithm that solves the given instance efficiently, our goal is to design a data structure that efficiently maintains a solution, or reports a lack thereof, upon updates in the instance.

We first consider the Closest String problem, for which we design randomized dynamic data structures with amortized update times $d^{\mathcal{O}(d)}$ and $|\Sigma|^{\mathcal{O}(d)}$, respectively, where Σ is the alphabet and d is the assumed bound on the maximum distance. These are obtained by combining known static approaches to Closest String with color-coding.

Next, we note that from a result of Frandsen et al. [J. ACM'97] one can easily infer a meta-theorem that provides dynamic data structures for parameterized string problems with worst-case update time of the form $\mathcal{O}_k(\log \log n)$, where k is the parameter in question and n is the length of the string. We showcase the utility of this meta-theorem by giving such data structures for problems Disjoint Factors and Edit Distance. We also give explicit data structures for these problems, with worst-case update times $\mathcal{O}(k2^k \log \log n)$ and $\mathcal{O}(k^2 \log \log n)$, respectively. Finally, we discuss how a lower bound methodology introduced by Amarilli et al. [ICALP'21] can be used to show that obtaining update time $\mathcal{O}(f(k))$ for Disjoint Factors and Edit Distance is unlikely already for a constant value of the parameter k.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Theory of computation → Predecessor queries

Keywords and phrases Parameterized algorithms, Dynamic data structures, String problems, Closest String, Edit Distance, Disjoint Factors, Predecessor problem

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.50

Related Version *Full Version*: https://arxiv.org/abs/2205.00441

Funding This work is the result of research conducted within research project number 2017/26/D/ST6/00264 financed by National Science Centre, Poland (Jędrzej Olkowski and Anna Zych-Pawlewicz). This work is a part of projects BOBR (Michał Pilipczuk) and TIPEA (Karol Węgrzycki) that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreements no. 948057 and 850979, respectively). Mateusz Rychlicki acknowledges support from the Engineering and Physical Sciences Research Council (EPSRC, project EP/V00252X/1).

40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023).
Editors: Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté;
Article No. 50; pp. 50:1–50:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The field of parameterized complexity is based on the principle of *parameterization*: measuring the usage of resources not only in terms of the total input size, but also in terms of auxiliary complexity measures called *parameters*. Traditionally, the principle is applied to static algorithms and their running times, but the idea can be – and has been – used within essentially every algorithmic paradigm. Among these, a recent line of research has identified the area of dynamic data structures as one where the application of the parameterized approach leads to new and interesting results, see e.g. [3, 15, 17, 18, 23, 30]. In this work, we continue this promising direction by investigating classic string problems considered in parameterized complexity.

Arguably, the most widely known parameterized string problem is Closest String.

Closest String
Input: Integer d and words $s_1, s_2, \ldots, s_n \in \Sigma^L$ over an alphabet Σ, each of length L
Task: Decide whether there exists a word $c \in \Sigma^L$ such that for every $i \in \{1, \ldots, n\}$, the Hamming distance between s_i and c is at most d.

Closest String has several natural parameters: n, d, L, $|\Sigma|$. For the parameterization by d and Σ, Gramm et al. [22] gave a $d^{\mathcal{O}(d)} \cdot (nL)^{\mathcal{O}(1)}$-time algorithm, while Ma and Sun [29] gave a $|\Sigma|^{\mathcal{O}(d)} \cdot (nL)^{\mathcal{O}(1)}$-time algorithm. By now, these are literally textbook examples of the technique of branching [16, Theorem 3.14 and Exercise 3.25], and their running time dependence on d is known to be asymptotically optimal under the Exponential Time Hypothesis (ETH) [28]. For the parameterization by n, the classic algorithm of Gramm et al. [22, Section 4] solves the problem in time $2^{n^{\mathcal{O}(n)}} \cdot L^{\mathcal{O}(1)}$ by a reduction to integer programming in dimension $n^{\mathcal{O}(n)}$. Recently, Koutecký et al. [25] improved this running time to $n^{\mathcal{O}(n^2)} \cdot L^{\mathcal{O}(1)}$ using exciting developments in parameterized algorithms for block-structured integer programs. Kernelization algorithms for Closest String were studied in [8].

We study the *dynamic variant* of Closest String, which is to design a dynamic data structure supporting the following operations:

- *Initialize* the data structure for a given instance of Closest String.
- *Update* the data structure upon modification of a single symbol in a single string s_i.
- *Query* whether the current instance is a yes-instance of Closest String.

Note that parameters n, d, L, and Σ are fixed on the initialization and do not change over the life of the data structure; only the strings $s_1, \ldots, s_n$ can be modified, and by one symbol at the time. Also, we assume that upon query, the data structure is only required to answer *yes* or *no*, and does not need to provide the solution c.

For this variant we give randomized dynamic data structures whose update times match the parametric factors in the runtimes of the algorithms of Gramm et al. [22] and of Ma and Sun [29].

▶ **Theorem 1.** *The dynamic variant of* Closest String *admits a randomized data structure with initialization time* $2^{\mathcal{O}(d)} \cdot nL|\Sigma|^{1+o(1)}$*, amortized update time* $2^{\mathcal{O}(d)}$*, and worst-case query time* $d^{\mathcal{O}(d)}$ *or* $|\Sigma|^{\mathcal{O}(d)}$*, whichever is smaller. The answer to each query may result with a false positive with probability at most* $2^{-\Omega(d)}$*; there are no false negatives.*

In the proof of Theorem 1 we combine the classic approach to Closest String, originating in [22, 29], with an interesting application of color-coding. The randomization comes from the color-coding; we can dispose of it using standard derandomization techniques (see [16, Section 5.6]), but at the cost of introducing an additional $\mathcal{O}(\log(nL))$ factor in the update time. Also, note that by the results of [28], under ETH one cannot expect to improve the query time to $d^{o(d)}$ or $|\Sigma|^{o(d)}$, even in the amortized sense.

Next, we turn attention to other problems. First, we note that combining a result of Frandsen et al. [19] on the dynamic word problem for aperiodic semigroups with the classic Schützenberger-McNaughton-Papert Theorem [32, 40] yields the following meta-theorem [1].

▶ **Theorem 2.** *Suppose Σ is a finite alphabet and $\mathcal{L} \subseteq \Sigma^\star$ is a language definable in logic $\mathsf{FO}[\Sigma, <]$. Then there exists a data structure that for a given word $w \in \Sigma^\star$, which can be updated over time by replacing single symbols, maintains whether $w \in \mathcal{L}$. The data structure can be initialized on a given word w in time $\mathcal{O}(n)$ where $n = |w|$, and then every update takes worst-case time $\mathcal{O}(\log \log n)$.*

Theorem 2 follows immediately from the combination explained above, so we consider it an essentially known result (though we could not find this precise formulation in the literature). What is new is the observation that this result is a very convenient tool for obtaining dynamic data structures in the parameterized setting. We showcase this by considering the following two text problems.

Disjoint Factors
Input: A word $w \in \{1, \ldots, k\}^\star$, where k is an integer **Task:** Decide whether there exist k pairwise disjoint (non-overlapping) substrings $w_1, w_2, \ldots, w_k$ of w such that for each $i \in \{1, \ldots, k\}$, w_i has length at least 2 and begins and ends with symbol i.

Edit Distance
Input: Integer k and two words $u, v \in \Sigma^\star$, where Σ is an alphabet **Task:** Decide whether $\mathsf{ed}(u, v) \leqslant k$, that is, whether v can be obtained from u by a sequence of at most k *edits*, each consisting of a deletion, insertion, or substitution of a single symbol.

Disjoint Factors has been introduced in [11] as a stepping stone for kernelization hardness of the Disjoint Cycles and Disjoint Paths problems. We choose to use it in this work as an example, because its simple combinatorial structure makes many basic ideas clearly visible. On the other hand, Edit Distance is a problem of immense importance with multiple applications (see for survey [34]). It can be solved in time $\mathcal{O}(n^2)$ by standard dynamic programming (where n is the total length of the words). The best currently known algorithm for Edit Distance runs in $\mathcal{O}(n^2/(\log n)^2)$ time [31] and under the Strong ETH, there is no strongly subquadratic algorithm [7, 1, 13, 2]. Here, we focus on parameterization by the size of the solution k. In terms of this parametrization Edit Distance can be solved in $\mathcal{O}(n + k^2)$ by the celebrated Landau and Vishkin algorithm [27] and even in sublinear time when approximation is allowed [9, 6, 21].

We observe that both for Disjoint Factors and for Edit Distance, the language of yes-instances can be defined in $\mathsf{FO}[\Sigma, <]$ using a sentence of length bounded in terms of the parameters. Therefore, by simply applying Theorem 2, we obtain data structures for the dynamic variants of Disjoint Factors and Edit Distance (defined similarly as for Closest String) with worst-case update times $f(k) \cdot \log \log n$ for some computable function $f(k)$. As usual with meta-theorems, the parametric dependence in these complexity guarantees is not explicit. For this reason, we also design explicit data structures for both problems.

[1] See Section A for formal definition of $\mathsf{FO}[\Sigma, <]$

▶ **Theorem 3.** *The dynamic variant of* Disjoint Factors *admits a data structure with initialization time* $\mathcal{O}(k2^k + kn)$*, worst-case query time* $\mathcal{O}(1)$*, and worst-case update time* $\mathcal{O}(k2^k \log\log n)$*.*

▶ **Theorem 4.** *The dynamic variant of* Edit Distance *admits a data structure with initialization time* $\mathcal{O}(kn)$*, worst-case query time* $\mathcal{O}(1)$*, and worst-case update time* $\mathcal{O}(k^2 \log\log n)$*.*

Our key component are van Emde Boas trees [42]. This is not surprising, as van Emde Boas trees are also the main tool underlying the proof of Theorem 2 (see [19]). In both cases, we heavily build upon known static algorithms [27, 11]. We point out that these results serve mainly as a demonstration that one can improve the dependence on the parameter guaranteed by Theorem 2 for concrete problems. We are not aware of any previous works on Disjoint Factors in exactly this dynamic setting. On the other hand Edit Distance was considered in the dynamic setting for unbounded values of k and only polynomial in n updates are known [24, 14, 5]. Landau et al. [26] considered parameterization by k of Edit Distance in the incremental setting. Moreover it is folklore that dynamic Edit Distance (even with insertions and deletions) can be maintained in $k^2 \operatorname{polylog}(n)$ update/query time (by combining [27, 36]). We reiterate that Theorems 3 and 4 mainly serve here as examples that dependence on k in the general framework in Theorem 2 can be improved for concrete problems.

Finally, we observe that we can use the hardness methodology proposed by Amarilli et al. [4] to establish conditional lower bounds against improving the update time in Theorems 3 and 4. More precisely, we prove that already for constant values of the parameters, the problems Disjoint Factors and Edit Distance are *prefix-U_1 hard*, which means that finding a data structure for them is at least as hard as designing a data structure for the following problem: for a dynamic word w over $\{0,1\}^\star$, support queries of the form "given i, is the first/leftmost occurence of the symbol 1 in w at position $\leqslant i$". Amarilli et al. [4] conjectured that no data structure for this problem achieves update time $\mathcal{O}(1)$, and our reduction carries this hardness over to the dynamic variants of Disjoint Factors and Edit Distance. Let us point out that the two discussed problems are just examples, and the obtained hardness methodology can be potentially applied to a other dynamic string problems.

Organization. In the Section 2 we give a short preliminaries. In Section 3 we present a proof of Theorem 1 for large alphabets. In Section 4 we prove Theorem 1 for small alphabets. The remaining proofs are deferred to the appendix. In Appendix A we prove Theorem 2. Appendix B contains omitted proofs.

In the full-version of this paper [37] we include we give dynamic data structures for Disjoint Factors and Edit Distance and show lower bounds for them.

2 Preliminaries

For a parameter ℓ, we write $\mathcal{O}_\ell(\cdot)$ to hide factors depending only on ℓ. The $\operatorname{poly}(n_1, n_2)$ denotes $(n_1 n_2)^{\mathcal{O}(1)}$. We use a shorthand notation $[n] \coloneqq \{1, \ldots, n\}$. For two sets X, Y, $X \triangle Y$ denotes their symmetric difference $(X \setminus Y) \cup (Y \setminus X)$. For two words $u, v \in \Sigma^L$, where $L \in \mathbb{N}$, by $\operatorname{dist}(u, v)$ we denote the Hamming distance between u and v. For a word $u \in \Sigma^L$ and a set $X \subseteq [L]$, we write $u[X] \in \Sigma^{|X|}$ for the word obtained from u by removing all positions outside of X. For $1 \leqslant i \leqslant j \leqslant m$, we write $u[i:j] \in \Sigma^{j-i+1}$ for $u[\{i, \ldots, j\}]$.

Computation Model. In this paper we work in the standard word-RAM model. In all our results the $\mathcal{O}(\log\log n)$ factors come exclusively from application of van Emde Boas trees that solve the Predecessor problem, where one needs to maintain a set S of $n \in \mathbb{N}$ integers with $w \in \mathbb{N}$ bits. In update one can add/remove integers to/from set S. During query, for a given integer x one should returns the largest integer $y \in S$ such that $x \geqslant y$. The Predecessor problem is a well-studied problem both in terms of lower and upper bounds (see the recent survey [35]). In the word-RAM model the complexity of Predecessor operations is well understood to be

$$\Theta\left(\max\left[1, \min\left\{\log_w(n), \frac{\log \frac{w}{\log w}}{\log\left(\log \frac{w}{\log w} / \log \frac{\log n}{\log w}\right)}, \log \frac{\log(2^w - n)}{\log w}\right\}\right]\right) \tag{1}$$

The upper and lower bounds were given by Pătraşcu and Thorup [38], see also [10, 20]. This means that strictly speaking $\mathcal{O}(\log\log n)$ factors in our paper, could be replaced with Equation 1 in the word-RAM model depending on word size. We are using the worse $\mathcal{O}(\log\log n)$ bound in order to keep the results transparent. Note that the $\mathcal{O}(\log\log n)$ bound for the Predecessor is tight in more restricted computation models (see, e.g., [33]).

3 Closest String

In this section, we show the first half of Theorem 1 by proving the following theorem.

▶ **Theorem 5.** *The dynamic variant of* Closest String *admits a randomized data structure with initialization time $2^{\mathcal{O}(d)}nL$, amortized update time $2^{\mathcal{O}(d)}$, and worst-case query time $d^{\mathcal{O}(d)}$. The answer to each query may result with a false positive with probability at most $2^{-\Omega(d)}$; there are no false negatives.*

Throughout this section we fix the parameter $d \in \mathbb{N}$ and denote $\mathcal{S} \coloneqq \{s_1, \ldots, s_n\}$ for brevity, and call it a *dictionary*. Then updates on such a dictionary consist of replacing one symbol in one word with another symbol. Our data structure is based on the static algorithm for Closest String due to Gramm et al. [22].

3.1 Branching for Closest String

Algorithm 1 presents a pseudocode for a $(3d)^d\mathrm{poly}(n, L)$ time algorithm for Closest String loosely based on [22]. We first check if every pair of words of $\mathcal{S}$ are at distance at most $2d$ from each other; otherwise, by triangle inequality, we can safely terminate and return that there is no solution. Following this, we run a recursive search that maintains a candidate q for a solution, together with an upper bound x on how far from q, in terms of Hamming distance, we allow the sought solution to be. Candidate q is initially set to be any word in $\mathcal{S}$ and upper bound x is initially set to d. Within the search, we first verify whether q is already a solution. If yes, then we can terminate, this time yielding a positive answer. Otherwise, there is some $s \in \mathcal{S}$ at distance more than d from q (and at most $2d$). Observe that due to the initial check and the fact that during recursion we modify q at most d times, it will be always the case that s and q differ on at most $3d$ positions. Hence, we can branch over one of at most $3d$ possibilities of modifying q by a single letter so that q gets closer to s. The nontrivial observation is that if there exists a solution, one of the modifications will take us closer to it in terms of the Hamming distance.

Algorithm 1 Pseudocode of static $\mathcal{O}((3d)^d \text{poly}(n, L))$ time algorithm for Closest String. To get a dynamic data structure, use Lemma 6 to perform manipulations on q.

```
Algorithm ClosestString(S, d)
1   if there exist s_i, s_j ∈ S such that dist(s_i, s_j) > 2d then
2       return False
3   Set q to be any word from S
4   return ClosestStringRec(S, q, d)

Procedure ClosestStringRec(S, q, x)
5   if x < 0 then
6       return False
7   if there exists s ∈ S such that dist(s, q) > d then
8       Find P := {i ∈ [L] | s[i] ≠ q[i]}          // Observe that |P| ⩽ 3d
9       for i ∈ P do
10          Set q' = q
11          Replace q'[i] = s[i]
12          if ClosestString(S, q', x − 1) then
13              return True
14      return False
15  return True
```

For the running time, observe that in each call we can make at most $|P| \leqslant 3d$ guesses. Moreover, through the execution of the algorithm we can only modify at most d letters in q. This means that the total size of the recursion tree is $\mathcal{O}((3d)^d)$.[2]

Let us take a closer look at the polynomial factors of the Algorithm 1 and discuss problems with dynamization. In line 1 we need to check if there exist words $s_i, s_j \in \mathcal{S}$ with $\text{dist}(s_i, s_j) > 2d$. Naively, one needs to iterate over every pair of words in $\mathcal{S}$ and compute the distance exactly which already requires n^2 iterations, where $n = |\mathcal{S}|$. Even if somehow, this number could be decreased, observe that in order to compute a distance between a fixed pair of words one needs to at least read them in $\mathcal{O}(L)$ time, which is too slow. Later, manipulations on the candidate word q also require $\mathcal{O}(nL)$ time in each call of the recursive procedure `ClosestStringRec()`, as q is checked against all words in $\mathcal{S}$.

We remedy these problems by introducing a data structure that maintains the dictionary $\mathcal{S}$ and provides access to all operations needed in the Algorithm 1, including efficient manipulation of the candidate q. This data structure is described in the following lemma.

▶ **Lemma 6** (Far word data structure). *There exists a randomized data structure that maintains the dictionary $\mathcal{S}$ of n words in Σ^L with amortized $2^{\mathcal{O}(d)}$ time updates; the initialization time is $2^{\mathcal{O}(d)} nL|\Sigma|$. The data structure provides the following method:*

- `QueryFarPair()`*: Decide if there exist $s, s' \in \mathcal{S}$ with $\text{dist}(s, s') > 2d$. The query may also return a positive answer in case there are no s, s' as above, but then it is guaranteed that the answer to the instance $(\mathcal{S}, d)$ of Closest String is negative.*

Further, the data structure provides access to a auxiliary word $q \in \Sigma^L$ through the following methods:

[2] With clever optimizations, one can decrease the running time to be $\mathcal{O}((d+1)^d \text{poly}(n, L))$ [16, Section 3.5].

- `Reset`(): *Reset q to the first word in $\mathcal{S}$.*
- `UpdateCandidate`(i, a)*: Change the ith position of q to symbol a.*
- `QueryFarWord`(): *Query if there exists $s \in \mathcal{S}$ with $\operatorname{dist}(s, q) > d$, and if so, return the pointer to s and the set of positions where s and q differ.*

Usage of the above requires the following promises:

- *Usage of* `Reset`() *must be preceded by obtaining a negative answer to* `QueryFarPair`().
- *Following resetting q to $s \in \mathcal{S}$ through usage of* `Reset`()*, the user has to guarantee that the assertion $\operatorname{dist}(q, s) \leqslant d$ will hold at all times till the next usage of* `Reset`().
- *Every update to any word in $\mathcal{S}$ resets q to be undefined, so that* `Reset`() *needs to be invoked again to enable operations on q.*

Methods `QueryFarPair`(), `Reset`(), `UpdateCandidate`(), `QueryFarWord`() *work in worst-case time $2^{\mathcal{O}(d)}$. Queries* `QueryFarPair`() *and* `QueryFarWord`() *return a false negative with probability $2^{-\Omega(d)}$; there are no false positives.*

A few remarks are in place regarding the use of randomness in the data structure of Lemma 6. Namely, random bits are used solely in the initialization of the data structure, and the correctness of subsequent uses of query methods depends on those initial random bits. Hence, the events when algorithm returns correct answers are *not* independent. This means that the error probability cannot be improved in the standard way by repeating each query many times. Instead, one can improve the error probability by setting up multiple independent copies of the data structure of Lemma 6.

With Lemma 6 stated, we can show how to derive Theorem 5 from it.

Proof of Theorem 5 assuming Lemma 6. We initialize and maintain $\alpha \log d$ independent copies of the data structure provided by Lemma 6 for some large enough constant α, to be determined later. Each update and each query is accordingly relayed to all these data structures; the output of a query is the disjunction of outputs provided by the individual data structures. In this way, we may assume that we have one instance of the data structure of Lemma 6 where the probability of a false negative is reduced to $(2^{-\Omega(d)})^{\alpha \log d} = (d^{-\Omega(d)})^{\alpha}$. The cost for this is that the running times of all methods are increased by a multiplicative factor of $\mathcal{O}(\log d)$; this will be immaterial in the forthcoming complexity analysis.

It remains to implement the query: we look for a word c that is at Hamming distance at most d from all the words in $\mathcal{S}$. The idea is to run Algorithm 1 with all operations replaced by suitable invocations of methods of the data structure of Lemma 6. Lines 1 and 3 are replaced by invocations of methods `QueryFarPair`() and `Reset`(), respectively. In line 7, we invoke method `QueryFarWord`(). Finally, in line 11 we use one `UpdateCandidate`() operation before recursing, and we roll-back this update (using the `UpdateCandidate`() method again) when returning from the recursion. The running time and the correctness (assuming no false negatives from the data structure of Lemma 6) follow from the correctness of the original static algorithm and Lemma 6. It is also straightforward to verify that the assumptions of Lemma 6 hold.

It remains to bound the probability of a false positive. Clearly, a false positive might arise only if some invocation of a method of the data structure of Lemma 6 returns a false negative. Since the recursion tree of procedure `ClosestString`() has depth at most d and branching at most $3d$, it has at most $2(3d)^d$ nodes, hence in total there are at most $1 + 2(3d)^d$ invocations of methods of the data structure of Lemma 6. By setting α large enough, we have $(1 + 2(3d)^d) \cdot (d^{-\Omega(d)})^{\alpha} \leqslant 2^{-\Omega(d)}$. So by the union bound, the probability of an error is bounded by $2^{-\Omega(d)}$. ◀

Now, we discuss the technical ideas behind the proof of Lemma 6. The key idea is that we can efficiently maintain an approximate solution, as explained in the following lemma.

▶ **Lemma 7** (Approximate Closest String)**.** *There exists a data structure that maintains a dictionary $\mathcal{S}$ of words in Σ^L with amortized update time $\mathcal{O}(|\Sigma|)$, as well as a word $o \in \Sigma^L$ with the following guarantee: if the answer to the* Closest String *instance $(\mathcal{S}, d)$ is positive, then $\operatorname{dist}(o, s) \leqslant 4|\Sigma| \cdot d$ for every $s \in \mathcal{S}$. The data structure can be initialized in $\mathcal{O}(nL)$ time.*

Moreover, the data structure also maintains a set $\Delta(o, s) \coloneqq \{i \in [L] \mid o[i] \neq s[i]\}$ for every $s \in \mathcal{S}$ and, upon request, can return $\Delta(o, s)$ in time $\mathcal{O}(|\Delta(o, s)|)$. Finally, the data structure can check whether $\operatorname{dist}(o, s) \leqslant 4|\Sigma| \cdot d$ for all $s \in \mathcal{S}$ in time $\mathcal{O}(1)$.

In Section 3.2 we prove Lemma 7. Next, in Section 3.3 we use an approach based on color coding to leverage Lemma 7 to maintain a dictionary $\mathcal{S}$ and implement query `QueryFarPair`(). Adding the functionality concerning the candidate word q uses similar arguments and is presented in Section 3.4. Looking at the statement of Lemma 7, the reader might be at this point worried that this plan involves complexities dependent also on $|\Sigma|$. However, in Section 3.3 we will show how to reduce $|\Sigma|$ to $\mathcal{O}(d)$ using color coding.

3.2 Approximate Closest String

In this section, we prove Lemma 7. The main idea is to define $o \in \Sigma^L$ through an approximate majority vote for every position, maintained in a lazy fashion. We formalize this through the following definition.

▶ **Definition 8** (Origin Word)**.** *An* origin word *for a dictionary $\mathcal{S}$ of words in Σ^L is a word $o \in \Sigma^L$ such that*

$$|\{s \in \mathcal{S} \mid s[i] = o[i]\}| \geqslant \frac{|\mathcal{S}|}{2|\Sigma|} \text{ for every } i \in [L].$$

We say that the origin word o is good *if $\operatorname{dist}(o, s) \leqslant 4|\Sigma| \cdot d$ for every $s \in \mathcal{S}$.*

By definition, if an origin word is good, then it is a solution for the Closest String instance $(\mathcal{S}, 4|\Sigma|d)$. We now show a reverse "soundness" implication: if some origin word is not good, then for sure there is no solution for $(\mathcal{S}, d)$.

▶ **Lemma 9.** *If for an instance $(\mathcal{S}, d)$ there exists an origin word that is not good, then the answer to $(\mathcal{S}, d)$ is negative.*

Proof. For the sake of contradiction, let us assume that there exists $c \in \Sigma^L$ such that $\operatorname{dist}(c, s) \leqslant d$ for every $s \in \mathcal{S}$. Moreover, there exists some origin word $o \in \Sigma^L$ and a witness $w \in \mathcal{S}$ such that $\operatorname{dist}(o, w) > 4d|\Sigma|$.

Let $C_{o,w}$ be the total count of matches between o and all words in $\mathcal{S}$ at the positions where o and w differ. That is,

$$C_{o,w} \coloneqq |\{(i, s) \in [L] \times \mathcal{S} \text{ such that } w[i] \neq o[i] \text{ and } s[i] = o[i]\}|.$$

Let us show a lower bound on $C_{o,w}$. Observe that for a witness $w \in \mathcal{S}$ there are at least $\operatorname{dist}(o, w)$ positions i that are taken into account when computing $C_{o,w}$. Moreover, by the definition of origin word o, for every position $i \in [L]$ at least $|\mathcal{S}|/(2|\Sigma|)$ words match o on position i. Therefore,

$$\frac{|\mathcal{S}| \cdot \operatorname{dist}(o, w)}{2|\Sigma|} \leqslant C_{o,w}. \tag{2}$$

On the other hand, we assumed that there exists $c \in \Sigma^L$ such that $\mathrm{dist}(c, s) \leqslant d$ for every $s \in \mathcal{S}$. Since $w \in \mathcal{S}$, by triangle inequality we have $\mathrm{dist}(s, w) \leqslant 2d$ for every $s \in \mathcal{S}$. Hence

$$C_{o,w} \leqslant 2d \cdot |\mathcal{S}|. \tag{3}$$

By combining Inequalities (2) and (3) we conclude that $\mathrm{dist}(o, w) \leqslant 4|\Sigma|d$, a contradiction. ◀

Next, we argue that in $\mathcal{O}(|\Sigma|)$ time we can maintain some origin word for a given dictionary.

▶ **Lemma 10.** *In $\mathcal{O}(nL)$ time we can initialize a data structure that for a given dictionary $\mathcal{S}$ of words in Σ^L maintains some origin word $o \in \Sigma^L$ with amortized update time $\mathcal{O}(|\Sigma|)$. The data structure also maintains the set $\Delta(o, s) \coloneqq \{i \in [L] \mid s[i] \neq o[i]\}$ for every $s \in \mathcal{S}$ and upon request, can return each set $\Delta(o, s)$ in time $\mathcal{O}(|\Delta(o, s)|)$. Finally, the data structure can check whether o is good in time $\mathcal{O}(1)$.*

Proof. Upon initialization, we set $o \in \Sigma^L$ so that for every position $i \in [L]$, $o[i]$ is a symbol that occurs the most often among $s[i]$ for all $s \in \mathcal{S}$. Clearly, o constructed in this way is an origin word. We also compute the relevant sets $\Delta(o, s)$.

The data structure stores the following additional data. For every position $i \in [L]$ and every symbol $\beta \in \Sigma$, we maintain a counter indicating the number of words $s \in \mathcal{S}$ such that $s[i] = \beta$. Each set $\Delta(o, s)$ is stored as a linked list (with no assumption on the order), plus there is an array of length L whose ith entry is either null if $i \notin \Delta(o, s)$, or contains a pointer to the relevant object on the linked list representing $\Delta(o, s)$. Additionally, with each set $\Delta(o, s)$, we maintain its size. Additionally, we store a single counter indicating the number of words $s \in \mathcal{S}$ such that $|\Delta(o, s)| \geqslant 4|\Sigma|d$. This counter can be used to answer queries about the goodness of o in time $\mathcal{O}(1)$. Upon initialization, all of the above can be computed in time $\mathcal{O}(nL)$ in a straightforward way.

We now explain how the data structure behaves upon an update. Suppose position $s_j[i]$ is modified. We update the relevant counters for position i and update $\Delta(o, s_j)$ accordingly. Next, we check whether the counter for the symbol $o[i]$ at position i did not drop below $|\mathcal{S}|/(2|\Sigma|)$. If not, then o remains an origin word and there is no need to change o. Otherwise, we modify $o[i]$ as follows.

By iterating through all words in $\mathcal{S}$, we compute the most frequent symbol among $s[i]$ for $s \in \mathcal{S}$, and we set $o[i]$ to be this symbol. Moreover, we iterate over all $s \in \mathcal{S}$ and update $\Delta(o, s)$ accordingly, by adding or removing the position i if needed. These operations require total time $\mathcal{O}(|\mathcal{S}|)$.

We now argue that the amortized update time is $\mathcal{O}(|\Sigma|)$. By the pigeon-hole principle, when symbol $o[i]$ gets modified, it is replaced by the most frequent symbol that occurs at least $|\mathcal{S}|/|\Sigma|$ times on position i in words from $\mathcal{S}$. Also, this is true for the symbol placed as $o[i]$ upon initialization. Therefore, before an update triggers a change of $o[i]$, there were at least $|\mathcal{S}|/(2|\Sigma|)$ updates on position i. We can charge the running time $\mathcal{O}(|\mathcal{S}|)$ used when modifying $o[i]$ to those previous updates, thus obtaining amortized update time $\mathcal{O}(|\Sigma|)$. ◀

Now Lemma 7 follows by combining Lemmas 10 and 9.

3.3 Detecting dissimilar words

In this section, we present a data structure, that maintains the dictionary and implements the method `QueryFarPair`(), and for now we ignore the methods for handling q.

In the data structure, we will maintain hashes of all words in $\mathcal{S}$ to the binary alphabet. More precisely, upon initialization of the data structure, we uniformly at random sample a function $h\colon [L] \times \Sigma \to \{0,1\}$ that assigns a label 0 or 1 to every position and symbol in the alphabet. This function is fixed for the whole life of the data structure and stored in it. In notation, we shall use a natural lift of $h\colon \Sigma^L \to \{0,1\}^L$ that applies h position-wise. In the data structure we store, together with $\mathcal{S}$, the hashed dictionary $\widetilde{\mathcal{S}} \coloneqq \{h(s) \mid s \in \mathcal{S}\}$. Observe that upon every update to $\mathcal{S}$ we can also update $\widetilde{\mathcal{S}}$ in constant time.

We also maintain an approximate solution $\tilde{o} \in \{0,1\}^L$ for dictionary $\widetilde{\mathcal{S}}$ using the data structure of Lemma 7. Recall that we can query the data structure of Lemma 7 about whether $\mathrm{dist}(\tilde{o}, \tilde{s}) \leqslant 8d$ for all $\tilde{s} \in \widetilde{\mathcal{S}}$ and if this is not the case, then we know for sure that the instance $(\widetilde{\mathcal{S}}, d)$ of Closest String has a negative answer. Note that this conclusion implies that the original instance $(\mathcal{S}, d)$ also has a negative answer.

In addition to the approximate solution $\tilde{o}$, the data structure of Lemma 7 provides an access to the sets $\Delta(\tilde{o}, \tilde{s})$ of positions where $\tilde{o}$ and $\tilde{s}$ differ, for all $\tilde{s} \in \widetilde{\mathcal{S}}$.

Finally, we also hash positions as follows. Upon initialization, we sample uniformly at random a function $\pi\colon [L] \to [16d]$ which maps positions to a set of $16d$ colors (numbers from 1 to $16d$). Again, this function is fixed for the whole life of the data structure and stored in it. For a word $\tilde{s} \in \widetilde{\mathcal{S}}$, let $\mathsf{colors}_{\tilde{o},\pi}(s) = \{\pi(i) \mid i \in [L] \text{ and } s[i] \neq \tilde{o}[i]\}$ be the set of colors assigned to the symbols in $\tilde{s}$ that are on positions where $\tilde{s}$ does not match the origin word $\tilde{o} \in \{0,1\}^L$.

In the data structure we maintain, for every $C \subseteq [16d]$, the set $\Phi(C)$ defined as follows:

$$\Phi(C) = \{\tilde{s} \in \widetilde{\mathcal{S}} \mid \mathsf{colors}_{\tilde{o},\pi}(\tilde{s}) = C\}.$$

In other words, $\Phi(C)$ is the set of words from $\widetilde{\mathcal{S}}$ that get assigned the color set C. The next statement shows that the sets $\Phi(C)$ can be maintained in $2^{\mathcal{O}(d)}$ time per update.

▶ **Lemma 11.** *We can initialize in $2^{\mathcal{O}(d)} \cdot nL$ time a data structure that for every $C \subseteq [16d]$ maintains the set $\Phi(C)$ in amortized $2^{\mathcal{O}(d)}$ time per update to $\mathcal{S}$. When queried about any $C \subseteq [16d]$, the data structure in $\mathcal{O}(1)$ time either returns any element from $\Phi(C)$, or asserts that $\Phi(C)$ is empty.*

The proof of Lemma 11 is deferred to Appendix B. It is rather technical and builds on the data structure of Lemma 7 by additionally storing sets $\Phi(C)$ as doubly-linked lists. Every modification to $\widetilde{\mathcal{S}}$ and o triggers a number of modifications to lists representing $\Phi(C)$, consisting of moving some elements from one list to another. The same amortization argument as the one used in the proof of Lemma 7 shows that the amortized update time is $2^{\mathcal{O}(d)}$.

Algorithm 2 Pseudocode for the method `QueryFarPair`().

```
Method: QueryFarPair()
if exists s̃ ∈ S̃ with dist(õ, s̃) > 8d then
    return Answer to (S, d) is negative.
for every X, Y ⊆ [16d] with |X △ Y| > 2d do
    if Φ(X) and Φ(Y) are both nonempty then
        return True        // There are words s, s' ∈ S with dist(s, s') > 2d.
return False               // It holds that dist(s, s') ⩽ 2d for all s, s' ∈ S.
```

We now present an implementation of the query operation; see Algorithm 2 for a pseudocode. We first check whether $\mathrm{dist}(\tilde{o}, \tilde{s}) \leqslant 8d$ for all $\tilde{s} \in \widetilde{\mathcal{S}}$. As argued in Lemma 9, if this is not the case, then we can safely conclude that the answer to the instance $(\mathcal{S}, d)$

is negative. Otherwise, we iterate over every pair of sets $X, Y \subseteq [16d]$ with $|X \triangle Y| = |(X \setminus Y) \cup (Y \setminus X)| > 2d$. Then, we check whether both $\Phi(X)$ and $\Phi(Y)$ are nonempty. If that is the case, then (as we will argue) any pair $(\tilde{s}, \tilde{s}') \in \Phi(X) \times \Phi(Y)$ satisfies $\mathrm{dist}(\tilde{s}, \tilde{s}') > 2d$, implying that the original words $s, s' \in \mathcal{S}$ also satisfy $\mathrm{dist}(s, s') > 2d$. Otherwise, if for every such X and Y at least one of $\Phi(X)$ or $\Phi(Y)$ is empty, we conclude that there is no pair $s, s' \in \mathcal{S}$ with $\mathrm{dist}(s, s') > 2d$.

Because the number of pairs $X, Y \subseteq [16d]$ is $2^{\mathcal{O}(d)}$, the query algorithm runs in $2^{\mathcal{O}(d)}$ time in total. The next lemma shows that if the algorithm finds some pair of words and reports that they are at distance larger than $2d$, then this answer is correct.

▶ **Lemma 12.** *Suppose Algorithm 2 finds a pair $X, Y \subseteq [16d]$ with $|X \triangle Y| > 2d$ and $\Phi(X) \neq \emptyset$ and $\Phi(Y) \neq \emptyset$. Then there are $s, s' \in \mathcal{S}$ such that $\mathrm{dist}(s, s') > 2d$.*

Proof. Consider any pair $(\tilde{s}, \tilde{s}') \in \Phi(X) \times \Phi(Y)$. Observe that for every color $r \in X \setminus Y$, there is a position i with $\pi(i) = r$ such that $\tilde{s}[i] \neq \tilde{o}[i]$ (due to $r \in X$), and we have $\tilde{o}[i] = \tilde{s}'[i]$ (due to $r \notin Y$). So $\tilde{s}[i] \neq \tilde{s}'[i]$, implying $s[i] \neq s'[i]$. Similar statements hold for every $r \in Y \setminus X$. Positions i as above have to be pairwise different due to receiving different colors in π. Therefore, because the number of such positions is more than $2d$, we conclude that s and s' differ on more than $2d$ positions. ◀

To finish the proof, it remains to analyze the success probability of Algorithm 2.

▶ **Lemma 13.** *If there exists a pair $a, b \in \mathcal{S}$ with $\mathrm{dist}(a, b) > 2d$, then Algorithm 2 detects such a pair with the probability at least $2^{-\mathcal{O}(d)}$, or concludes that the answer to the instance $(\mathcal{S}, d)$ is negative.*

Note that in Lemma 6 we promised error probability bounded by $2^{-\Omega(d)}$, while Lemma 13 provides a bound of $1 - 2^{-\mathcal{O}(d)}$ on the error probability. This can be easily remedied by maintaining $2^{\Theta(d)}$ independent copies of the data structure. This increases the time of update and initialization by a $2^{\mathcal{O}(d)}$ factor.

Proof of Lemma 13. Let $a, b \in \mathcal{S}$ be a pair of words in $\mathcal{S}$ with $\mathrm{dist}(a, b) > 2d$. First, we argue that after hashing the alphabet, we still have $\mathrm{dist}(h(a), h(b)) > 2d$ with sufficiently high probability. Let $P \subseteq \{i \in [L] \mid a[i] \neq b[i]\}$ be any set of size exactly $2d + 1$ consisting of positions where a and b differ.

Let $\tilde{a} = h(a)$ and $\tilde{b} = h(b)$ for First, we claim that with probability at least $2^{-\mathcal{O}(d)}$ it holds that $\mathrm{dist}(\tilde{a}, \tilde{b}) > 2d$. Observe that for a fixed position $i \in P$, the probability that h assigns different symbols to $a[i]$ and to $b[i]$ is $1/2$. Since h is sampled on each position $i \in [L]$ independently, the probability that this happens for all positions in P is $2^{-|P|} = 2^{-\mathcal{O}(d)}$.

From now on, let us assume that $\mathrm{dist}(\tilde{a}, \tilde{b}) > 2d$. Moreover, by Line 1 we may assume that $\mathrm{dist}(\tilde{o}, \tilde{s}) \leqslant 8d$ for every $\tilde{s} \in \widetilde{\mathcal{S}}$. Consider the set

$$\Delta_{\tilde{o}}(\tilde{a}, \tilde{b}) \coloneqq \{i \in [L] \mid \tilde{a}[i] \neq \tilde{o}[i] \text{ or } \tilde{b}[i] \neq \tilde{o}[i]\}.$$

Observe that since $\mathrm{dist}(\tilde{o}, \tilde{a}) \leqslant 8d$ and $\mathrm{dist}(\tilde{o}, \tilde{b}) \leqslant 8d$, we have $k \coloneqq |\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})| \in [2d + 1, 16d]$. Now, we claim that with probability $2^{-\mathcal{O}(d)}$ the function π assigns different colors to all positions in $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$. There are $(16d)^k$ different colorings on $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$. However, only $\binom{16d}{k} k!$ of them assign different colors to $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$. Therefore the probability that π assigns different colors on $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$ is:

$$\Pr\left[|\{\pi(i) \mid i \in \Delta_{\tilde{o}}(\tilde{a}, \tilde{b})\}| = k\right] = \frac{\binom{16d}{k} k!}{(16d)^k} = \frac{(16d)}{(16d)} \cdots \frac{(16d - k + 1)}{(16d)} > \frac{(16d)!}{(16d)^{16d}} > e^{-16d}$$

where the last inequality follows from the well-known bound $n! > (n/e)^n$. Hence, the probability that π assigns different colors to all positions in $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$ is $2^{-\mathcal{O}(d)}$. Now, we claim that if that indeed happens, then Algorithm 2 detects a suitable pair.

Let X and Y be sets of colors such that $\tilde{a} \in \Phi(X)$ and $\tilde{b} \in \Phi(Y)$. It suffices to show that $|X \triangle Y| > 2d$. Observe that every position where $\tilde{a}$ and $\tilde{b}$ differ belongs to $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$, hence these (more than $2d$) positions receive different colors in π. Further, for every position where $\tilde{a}$ and $\tilde{b}$ differ, the color of this position belongs to $X \triangle Y$, for outside of positions of $\Delta_{\tilde{o}}(\tilde{a}, \tilde{b})$ the words $\tilde{o}, \tilde{a}, \tilde{b}$ all agree. It follows that $|X \triangle Y| > 2d$. ◀

3.4 Maintaining a candidate solution

In this section, we finish the proof of Lemma 6 by implementing the operations on the candidate word q. This proof builds upon the construction from Section 3.3 using the same ideas, so we only briefly discuss the additional elements that need to be maintained.

Observe that q is always reset to the first word $s_1 \in \mathcal{S}$ and operations on q are performed under the promise that $\mathrm{dist}(q, s_1) \leqslant d$ at all times. Therefore, we maintain q implicitly by remembering only at most d positions on which s_1 and q differ, and what are the symbols of q on those positions. This allows us to implement the reset and update operations for q in time $2^{\mathcal{O}(d)}$. (Recall here that in Section 3.3 we in fact maintained $2^{\mathcal{O}(d)}$ independent copies of the data structure in order to boost the error probability.) Also, we maintain the hashed version $\tilde{q} := h(q)$.

The method `QueryFarWord()` is implemented using a similar mechanism as was used in Section 3.3. For technical reasons, we extend the palette of colors used by π from $[16d]$ to $[17d]$. Then we maintain the sets $\Phi(C) \subseteq \widetilde{\mathcal{S}}$ for $C \subseteq [17d]$ as before. However, instead of iterating over all pairs $X, Y \subseteq [17d]$ with $|X \triangle Y| > 2d$, we first compute $Q := \mathsf{colors}_{\tilde{o},\pi}(\tilde{q})$ and then iterate over all $X \subseteq [17d]$ such that $|X \triangle Q| > d$ and check whether $\Phi(X)$ is nonempty. The same reasoning as in Section 3.3 shows that if there exists $s \in \mathcal{S}$ with $\mathrm{dist}(q, s) > d$, then with high enough probability we will find such an s as any element of $\Phi(X)$.

Note that in the description above we did not specify how the set $\mathsf{colors}_{\tilde{o},\pi}(\tilde{q})$ is computed. This can be done by first obtaining the set $\Delta(\tilde{o}, \tilde{s}_1)$ from the data structure of Lemma 7, and then inspecting all the positions of $\Delta(\tilde{o}, \tilde{s}_1) \cup \Delta(s_1, q)$, where $\Delta(s_1, q)$ are the at most d positions where s_1 and q differ. Note here that we may assume that $|\Delta(\tilde{o}, \tilde{s}_1)| \leqslant 8d$, for otherwise the data structure presented in Section 3.3 must have returned that the answer to $(\mathcal{S}, d)$ is negative when resetting q. Further, this reasoning shows that $|\Delta(\tilde{o}, \tilde{q})| \leqslant 9d$ at all times. Note that in the correctness argument presented in Section 3.3, we used the assumption that $\mathrm{dist}(\tilde{o}, \tilde{a}) \leqslant 8d$ and $\mathrm{dist}(\tilde{o}, \tilde{b}) \leqslant 8d$, and this is why we chose a palette of colors of size $16d$. Now, we have $\mathrm{dist}(\tilde{o}, \tilde{q}) \leqslant 9d$ and $\mathrm{dist}(\tilde{o}, \tilde{s}) \leqslant 8d$, so a palette of $17d$ colors suffices.

It remains to argue that if s with $\mathrm{dist}(q, s) > d$ is found, the set P of positions on which q and s differ can be reported in time $\mathcal{O}(d)$. But again, P can be constructed by inspecting all positions of $\Delta(\tilde{o}, \tilde{s}_1) \cup \Delta(q, s_1)$, and this set has size $\mathcal{O}(d)$ and can be obtained by a query to the data structure of Lemma 7. This finishes the proof of Lemma 6.

4 Closest String for small alphabets

In this section we analyse the complexity of Closest String for small alphabets and show that our techniques also apply in this setting. That is, we prove the second half of Theorem 1, presented below.

▶ **Theorem 14.** *The dynamic variant of* Closest String *admits a randomized data structure with initialization time* $2^{\mathcal{O}(d)}nL|\Sigma|^{1+o(1)}$*, amortized update time* $2^{\mathcal{O}(d)}$*, and worst-case query time* $(|\Sigma|-1)^d 2^{\mathcal{O}(d)}$*. The answer to each query may result with a false negative with probability at most* $2^{-\Omega(d)}$*; there are no false positives.*

The strategy is exactly the same as in Section 3. First, we present a static algorithm with running time $(|\Sigma|-1)^d \cdot 2^{\mathcal{O}(d)} \cdot (nL)^{\mathcal{O}(1)}$, which is essentially the algorithm proposed by Ma and Sun [22]. Next, we show how to use Lemma 6 to implement this static algorithm to the dynamic setting. The algorithm is presented using pseudocode as Algorithm 3. We present it somewhat differently than Ma and Sun in order to streamline the analysis of the dynamic variant.

Algorithm 3 Pseudocode of a $(|\Sigma|-1)^d \cdot 2^{\mathcal{O}(d)} \cdot (nL)^{\mathcal{O}(1)}$-time static algorithm for Closest String. To get a dynamic data structure with query time $(|\Sigma|-1) \cdot 2^{\mathcal{O}(d)}$, use Lemma 6 for operations on q.

Algorithm : `ClosestStringSmallAlphabet`$(\mathcal{S}, d)$

Set $F := \emptyset$
Set q to be the first word $s_1 \in \mathcal{S}$
Set $b := d$
while *exists* $s \in \mathcal{S}$ *such that* $\mathrm{dist}(s,q) > d$ **do**
  Find $P := \{i \in [L] \text{ such that } s[i] \neq q[i]\} \setminus F$
  if $\mathrm{dist}(s,q) > 2d$ *or* $P = \emptyset$ **then**
    return False
  Guess $Q = \{i \in P \text{ such that } c[i] \neq q[i]\}$ `// c ∈ Σ^L denotes the sought solution`
  if $|Q| > b$ *or* $Q = \emptyset$ **then**
    return False
  for $i \in Q$ **do**
    Guess $c[i] \in \Sigma \setminus \{q[i]\}$
    Set $q[i] := c[i]$
  $F := F \cup P$
  $b := \min(d - \mathrm{dist}(s,q), b - |Q|)$
  if $b < 0$ **then**
    return False
return True

The algorithm maintains three global values. The first one is a set $F \subseteq [L]$ of *fixed indices*. The second one is a word $q \in \Sigma^L$ that is a candidate for the solution, which at the start is set to be any word from $\mathcal{S}$. The third one is a *budget* $b \in \mathbb{N}$, initially set to d.

We imagine the algorithm as a nondeterministic procedure that, having in mind some solution $c \in \Sigma^L$, guesses parts of c along the execution and appropriately modifies q. The set F is used to keep track of the positions that are already assumed to be fixed as in F. As usual, nondeterminism is determined by branching over all possibilities, and the total number of branches determines the running time of the algorithm. At every point, even in branches where guesses were inconsistent with c, the algorithm maintains the following invariant:

($\diamondsuit$) There is $s \in \mathcal{S}$ such that $s[\overline{F}] = q[\overline{F}]$ and $b = d - \mathrm{dist}(q[F], s[F])$, where we denote $\overline{F} = [L] \setminus F$.

In this way, one may think of b as of the budget that is left for changing symbols positions in q outside of F: at most b of them can be still changed, for otherwise the solution would be too far from s.

Every step of the algorithm works as follows. First, we find a word $s \in \mathcal{S}$ with $\mathrm{dist}(s, q) > d$. If no such word exists, then the current candidate q is a solution and we can terminate the procedure claiming a positive answer. Otherwise, we compute the set P of positions where s and q differ, and we remove from it all positions that were fixed before.

Next, we check whether $\mathrm{dist}(s, q) > 2d$, which translates to the condition $\mathrm{dist}(s[F \cup P], q[F \cup P]) > 2d$. If this is the case, we terminate and provide a negative answer: there is no way to obtain a word at distance at most d from s by changing at most d positions in q. If $P = \emptyset$, we can also terminate and provide a negative answer: already on fixed positions, our candidate q and s differ by more than d. Otherwise, when $\mathrm{dist}(s, q) \leqslant 2d$ and $P \neq \emptyset$, we guess exactly the symbols in c at positions from P and we modify q to have $q[P] = c[P]$. This is done through a two-stage process: first we guess the set of positions $Q \subseteq P$ where q needs to be modified, and then we guess the symbols of c at positions of Q; for each there are $|\Sigma| - 1$ possibilities. Note that we may restrict attention to sets Q that are nonempty (for $\mathrm{dist}(s, q) > d$) and satisfy $|Q| \leqslant b$ (by invariant 18). Finally, we add P to the set F of fixed indices and update b to the minimum of the two values: $d - \mathrm{dist}(s, q)$ and $b - |Q|$. It is straightforward to verify that this way invariant 18 is still maintained: either s or the previous witness for 18 may serve as the new witness for 18. Clearly, if b became negative, it is safe to terminate the branch. Otherwise we continue the search until a candidate q at distance at most d from all strings in $\mathcal{S}$ is found.

This concludes the description of the algorithm. The correctness is clear from the description as we return True only if our candidate is at the distance at most d from all input strings.

It is now straightforward to turn this algorithm into a dynamic data structure just as we did in the proof of Theorem 5. Namely, we maintain the data structure of Lemma 6, and use it to operate on the candidate word q. All distance checks can be implemented in linear time by verifying the $\mathcal{O}(d)$-sized difference sets provided by this data structure. We will later show that the whole recursion tree of Algorithm 3 has total size at most $(|\Sigma| - 1)^d \cdot 2^{\mathcal{O}(d)}$. Hence, as the operations in the data structure of Lemma 6 take amortized time $2^{\mathcal{O}(d)}$, the complexity guarantees promised in Theorem 14 follow in the same way as it was the case for Theorem 5. As for the error probability, we can maintain $\alpha \cdot \log |\Sigma|$ independent copies of the data structure of Lemma 6 for some large constant α, so that the probability that this composite data structure returns a false negative is reduced to $(|\Sigma|^{-\Omega(d)})^\alpha$. Then, just as in the proof of Theorem 5, it follows from the union bound that the probability of a false negative in Algorithm 3 is at most $2^{-\Omega(d)}$.

We are left with bounding the running time of Algorithm 3, or more precisely, showing that the whole recursion tree has size at most $(|\Sigma| - 1)^d \cdot 2^{\mathcal{O}(d)}$. The argument conceptually follows the reasoning of Ma and Sun [29]; we present it for completeness.

Runtime. The key observation is the following lemma.

▶ **Lemma 15.** *Consider ith iteration of the while loop in Algorithm 3 (with any guesses made). Let b_i be the value of b before this iteration, and b_{i+1} be the value of b after this iteration. Then $b_{i+1} \leqslant b_i/2$.*

We first argue that the claimed runtime of Algorithm 3 follows from Lemma 15. Consider any root-to-leaf path in the recursion tree of the algorithm; this corresponds to a single run of Algorithm 3 treated as a nondeterministic procedure, with some guesses made along the

way. For iterations $i = 1, 2, \ldots, p$ of the while-loop, where p is the total number of iterations made, let b_i be the value of b at the beginning of the ith iteration, and let ℓ_i be the size of the set Q considered in the ith iteration. Observe the following:

- We have $b_i \leqslant d/2^{i-1}$ for all $i \in [p]$ (because $b_1 = d$ and, by Lemma 15, $b_{i+1} \leqslant b_i/2$ for all $i \in [p-1]$).
- We have $1 \leqslant \ell_i \leqslant b_i$ for all $i \in [p]$ (because in the algorithm we consider only nonempty sets Q satisfying $|Q| \leqslant b$).
- We have $\sum_{i=1}^{p} \ell_i \leqslant d$ (because b decreases by at least ℓ_i in the ith iteration, and the procedure terminates once b becomes negative).

Therefore, every root-to-leaf path in the recursion tree can be uniquely described by specifying the following data:

(a) Positive integers $\ell_1, \ldots, \ell_p$ satisfying $\sum_{i=1}^{p} \ell_i \leqslant d$.
(b) For each $i \in [p]$, a choice of a subset Q_i of size ℓ_i of the set P_i, where P_i, Q_i are the sets P, Q considered in the ith iteration.
(c) For each $i \in [p]$, a choice of symbols guessed to be fixed at the positions of Q_i.

For a, it is well-known that the number of representations of d as a sum of numbers $\ell_1, \ldots, \ell_p$ is bounded by $2^{\mathcal{O}(d)}$. For c, the total number of choices is bounded by

$$\prod_{i=1}^{p} (|\Sigma| - 1)^{\ell_i} \leqslant (|\Sigma| - 1)^d.$$

Finally, for b we shall use the following known bound.

▶ **Lemma 16** (cf. Lemma 124 in [39]). *If m, k are nonnegative integers, then $\binom{m+k}{k} \leqslant 2^{2\sqrt{mk}}$.*

Since we always have $|P_i| \leqslant 2d$, the number of choices for b is bounded as follows:

$$\prod_{i=1}^{p} \binom{2d}{\ell_i} \leqslant \prod_{i=1}^{p} 2^{2\sqrt{2d\ell_i}} \leqslant \prod_{i=1}^{p} 2^{2\sqrt{2d \cdot d/2^{i-1}}} = 2^{2\sqrt{2} \cdot d \cdot \sum_{i=0}^{\infty} 2^{-i/2}} = 2^{\mathcal{O}(d)}.$$

So all in all, the total number of root-to-leaf paths in the recursion tree is bounded by

$$2^{\mathcal{O}(d)} \cdot 2^{\mathcal{O}(d)} \cdot (|\Sigma| - 1)^d = (|\Sigma| - 1)^d \cdot 2^{\mathcal{O}(d)},$$

as claimed. It remains to prove the Lemma 15.

Proof of Lemma 15. Let q_i and q_{i+1} be the candidate word respectively at the beginning and at the end of the ith iteration, that is, after guessing is performed. Recall that there is $s \in \mathcal{S}$ with $\mathrm{dist}(s, q_i) > d$. Further, recall that

$$b_{i+1} = \min(d - \mathrm{dist}(s, q_{i+1}), b_i - |Q|).$$

To prove that $b_{i+1} \leqslant b_i/2$, it suffices to show that

$$(d - \mathrm{dist}(s, q_{i+1})) + (b_i - |Q|) \leqslant b_i,$$

or equivalently,

$$d \leqslant \mathrm{dist}(s, q_{i+1}) + |Q|. \tag{4}$$

By triangle inequality we have

$$d < \mathrm{dist}(s, q_i) \leqslant \mathrm{dist}(s, q_{i+1}) + \mathrm{dist}(q_i, q_{i+1}),$$

but we also have

$$\mathrm{dist}(q_i, q_{i+1}) = |Q|.$$

So this establishes (4) and finishes the proof. ◀

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.

2 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.

3 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Trans. Algorithms*, 16(4):45:1–45:46, 2020. doi:10.1145/3395037.

4 Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 116:1–116:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.116.

5 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

6 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 990–1001. IEEE, 2020. doi:10.1109/FOCS46700.2020.00096.

7 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

8 Manu Basavaraju, Fahad Panolan, Ashutosh Rai, M. S. Ramanujan, and Saket Saurabh. On the kernelization complexity of string problems. *Theor. Comput. Sci.*, 730:21–31, 2018. doi:10.1016/j.tcs.2018.03.024.

9 Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 316–324. ACM, 2003. doi:10.1145/780542.780590.

10 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. doi:10.1006/jcss.2002.1822.

11 Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo. Kernel bounds for disjoint cycles and disjoint paths. *Theor. Comput. Sci.*, 412(35):4570–4578, 2011. doi:10.1016/j.tcs.2011.04.039.

12 Mikołaj Bojańczyk. *Languages recognised by finite semigroups and their generalisations to objects such as trees and graphs, with an emphasis on definability in monadic second-order logic.* In preparation, 2020. URL: https://www.mimuw.edu.pl/~bojan/papers/algebra-26-aug-2020.pdf.

13 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.

14 Panagiotis Charalampopoulos. *Data Structures for Strings in the Internal and Dynamic Settings.* PhD thesis, King's College London, 2021.

15 Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartlomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021. doi:10.1137/1.9781611976465.50.

16 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

17 Zdeněk Dvořák, Martin Kupec, and Vojtěch Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proceedings of the 22th Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2014. doi:10.1007/978-3-662-44777-2_28.

18 Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Proceedings of the 13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2013. doi:10.1007/978-3-642-40104-6_27.

19 Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997. doi:10.1145/256303.256309.

20 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

21 Elazar Goldenberg, Tomasz Kociumaka, Robert Krauthgamer, and Barna Saha. Gap edit distance via non-adaptive queries: Simple and optimal. *CoRR*, abs/2111.12706, 2021. arXiv:2111.12706.

22 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for Closest String and related problems. *Algorithmica*, 37(1):25–42, 2003. doi:10.1007/s00453-003-1028-3.

23 Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis, and Cristian Riveros. Dynamic data structures for timed automata acceptance. In *Proceedings of the 16th International Symposium on Parameterized and Exact Computation, IPEC 2021*, volume 214 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.IPEC.2021.20.

24 Heikki Hyyrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *Journal of Discrete Algorithms*, 34:2–17, 2015. doi:10.1016/j.jda.2015.05.007.

25 Dusan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n-fold integer programming and applications. *Math. Program.*, 184(1):1–34, 2020. doi:10.1007/s10107-019-01402-2.

26 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

27 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

28 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. *SIAM J. Comput.*, 47(3):675–702, 2018. doi:10.1137/16M1104834.

29 Bin Ma and Xiaoming Sun. More efficient algorithms for closest string and substring problems. *SIAM J. Comput.*, 39(4):1432–1443, 2009. doi:10.1137/080739069.

30 Konrad Majewski, Michał Pilipczuk, and Marek Sokołowski. Maintaining $CMSO_2$ properties on dynamic structures with bounded feedback vertex number. *CoRR*, abs/2107.06232, 2021. arXiv:2107.06232.

31 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

32 Robert McNaughton and Seymour Papert. *Counter-free automata*. MIT Press, 1971.

33 Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM J. Comput.*, 17(6):1093–1102, 1988. doi:10.1137/0217070.

34 Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

35 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5):105:1–105:35, 2020. doi:10.1145/3409371.

36 Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, et al. Fully dynamic data structure for lce queries in compressed space. *arXiv preprint arXiv:1605.01488*, 2016.

37 Jędrzej Olkowski, Michał Pilipczuk, Mateusz Rychlicki, Karol Węgrzycki, and Anna Zych-Pawlewicz. Dynamic data structures for parameterized string problems. *CoRR*, abs/2205.00441, 2022. doi:10.48550/arXiv.2205.00441.

38 Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

39 Michał Pilipczuk. *Tournaments and optimality: New results in parameterized complexity.* PhD thesis, University of Bergen, 2013.

40 Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Inf. Control.*, 8(2):190–194, 1965. doi:10.1016/S0019-9958(65)90108-7.

41 Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic.* PhD thesis, MIT, 1974.

42 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

A Applications of Meta-Theorems

In this section we first state a meta-theorem for string problems definable in first-order logic FO; this result follows easily from the work of Frandsen et al. [19] using the classic Schützenberger-McNaughton-Papert theorem [32, 40]. Then we explain how to use the meta-theorem for the following toy problems: Disjoint Factors and Edit Distance. As usual with meta-theorems, the parametric factor in the complexity guarantees of the obtained data structures is not explicit, and typically is much higher than if one constructs the data structure "by hand". Therefore, we next show how to derive concrete data structures with concrete complexity guarantees for Disjoint Factors and Edit Distance. Finally, we discuss a methodology for lower bounds introduced by Amarilli et al. [4], and we apply it to derive lower bounds for those two problems.

A.1 A meta-theorem

We first need to recall basic knowledge on different equivalent views on regular languages. This material is standard in the area of algebraic theory of languages, so we refer an interested reader to the book of Bojańczyk [12] for a broader introduction. In particular, we explain the contemporary understanding of the material, and for appropriate references and historical remarks, we refer to [12].

The first view is through the lens of logic. Fix a finite alphabet Σ. We consider the logic $\mathsf{MSO}[\Sigma, <]$ operating on words. In this logic there are variables for single positions (denoted with small letters) and subsets of positions (denoted with capital letters). The atomic formulas are of the following form:

- equality test $x = y$;
- test $x \in X$ checking that position x belongs to position subset X;

- for every $a \in \Sigma$, test $a(x)$ checking that at position x there is symbol a; and
- test $x < y$ checking that position x appears before position y.

Formulas of $\mathsf{MSO}[\Sigma, <]$ can be obtained from atomic formulas using standard boolean connectives and quantification (both universal and existential, and applicable to both types of variables). $\mathsf{FO}[\Sigma, <]$ is a fragment of $\mathsf{MSO}[\Sigma, <]$ where we disallow variables for subsets of positions.

A *sentence* is a formula without free variables. By $w \models \varphi$ we mean that the sentence φ is satisfied in the word w. For a sentence $\varphi \in \mathsf{MSO}[\Sigma, <]$, the language *defined* by φ consists of all words w in which φ is satisfied. A language $\mathcal{L} \subseteq \Sigma^\star$ is MSO-*definable* if $\mathcal{L}$ is defined by some $\varphi \in \mathsf{MSO}[\Sigma, <]$ as above, and FO-*definable* if it is defined by some $\varphi \in \mathsf{FO}[\Sigma, <]$. It appears that regular languages exactly coincide with ones definable in MSO.

▶ **Theorem 17.** *A language of finite words over a finite alphabet is regular if and only if it is* MSO*-definable.*

The next view is through semigroup homomorphisms. Consider a language $\mathcal{L} \subseteq \Sigma^\star$. By endowing $\Sigma^\star$ with the concatenation operation we can regard it as a semigroup. For another semigroup S and a (semigroup) homomorphism $h\colon \Sigma^\star \to S$, we say that h *recognizes* $\mathcal{L}$ if there exists $A \subseteq S$ such that $\mathcal{L} = h^{-1}(A)$; in other words, whether $w \in \mathcal{L}$ can be recognized by looking at $h(w)$ and determining whether it belongs to A. It turns out that regular languages are also exactly those that are recognized by homomorphisms to finite semigroups.

▶ **Theorem 18.** *A language of finite words over a finite alphabet is regular if and only if it is recognized by a homomorphism to a finite semigroup.*

Further, it is known that if $\mathcal{L}$ is regular, then there exists a unique minimal – in terms of cardinality – semigroup S such that there is a homomorphism from $\Sigma^\star$ to S recognizing $\mathcal{L}$. This semigroup is called the *syntactic semigroup* for S.

It turns out that FO-definable languages can be characterized in terms of algebraic properties of their syntactic semigroups. Here, a semigroup is *aperiodic* (or *group-free*) if it does not contain any non-trivial group.

▶ **Theorem 19** (Schützenberger-McNaughton-Papert Theorem, [32, 40])**.** *A regular language* $\mathcal{L}$ *is* FO*-definable if and only if its syntactic semigroup is aperiodic.*

With these standard tools recalled, we can proceed to the setting of dynamic data structures.

Fix a finite alphabet Σ and consider a language $\mathcal{L} \subseteq \Sigma^\star$. The *word problem* for $\mathcal{L}$ is to design a data structure that maintains a dynamic word $w \in \Sigma^\star$ and supports the following operations:

- $\mathsf{init}(w)$: Initialize the data structure with the given word w.
- $\mathsf{update}(i, a)$: Update w by replacing the symbol at position i by symbol $a \in \Sigma$.
- $\mathsf{query}()$: Determine whether $w \in \mathcal{L}$.

The complexity guarantees of such a data structure is typically measured in terms of $n := |w|$. Note that this value is fixed upon initialization and then stays the same throughout the life of the data structure.

We can also consider the word problem for semigroups. Suppose S is a semigroup. Then the word problem for S is defined as above for words over S (that is, words $w \in S^\star$), where query is redefined as follows: Output the (left-to-right) product of all the symbols in w.

Observe that the word problem for a regular language $\mathcal{L} \subseteq \Sigma^\star$ easily reduces to the word problem for its syntactic semigroup S. Indeed, if $h\colon \Sigma^\star \to S$ is the homomorphism recognizing $\mathcal{L}$, say $\mathcal{L} = h^{-1}(A)$ for some $A \subseteq S$, then in the reduction we can map symbols $a \in \Sigma$ to their images $h(a) \in S$, and whether $w \in \mathcal{L}$ can be deduced by checking whether $h(w) \in A$.

Frandsen et al. [19] proposed an efficient dynamic data structure for the word problem in aperiodic semigroups.

▶ **Theorem 20** ([19]). *Let S be a finite aperiodic semigroup. Then there is a data structure for the word problem for S with initialization time $\mathcal{O}(n)$, worst-case update time $\mathcal{O}(\log \log n)$, and worst-case query time $\mathcal{O}(1)$.*

By combining Theorems 19 and 20 using the reduction presented above, we obtain the following.

▶ **Theorem 21.** *Let Σ be a finite alphabet and suppose $\mathcal{L} \subseteq \Sigma^\star$ is FO-definable. Then there is a data structure for the word problem for $\mathcal{L}$ with initialization time $\mathcal{O}(n)$, worst-case update time $\mathcal{O}(\log \log n)$, and worst-case query time $\mathcal{O}(1)$.*

A few remarks are in order. First, the proof of Theorem 20 relies on induction on the Khron-Rhodes decomposition of the semigroup S, where in each step of the induction one applies van Emde Boas trees [42]. The induction has depth bounded by the size of S, so one can view this data structure as $\mathcal{O}(|S|)$ van Emde Boas trees stacked "on top of each other". Consequently, the constants hidden in the $\mathcal{O}(\cdot)$ notation in Theorem 20 depend on S, but not horribly: they are polynomial in $|S|$. However, there is a much more significant complexity blow-up hidden in Theorem 18. Specifically, if a regular language $\mathcal{L}$ is defined by an $\mathsf{FO}[\Sigma, <]$ sentence φ, then the syntactic semigroup of $\mathcal{L}$ has size bounded by a function of $|\varphi|$, but this function is in general non-elementary – it is basically a tower of height equal to the quantifier rank of φ. This non-elementary dependence is known to be unavoidable [41]. Therefore, whenever one applies Theorem 21 in order to obtain data structures for a problem based on its description in FO, one should bear in mind that the constants hidden in the $\mathcal{O}(\cdot)$ notation depend non-elementarily on the length of the description.

Second, recently Amarilli et al. [4] gave a characterization of regular languages for which data structures with guarantees as in Theorem 20 exist. This characterization renders the tractability region to be a bit broader than just FO-definability, for instance the languages "on every even position there is symbol a" or "in total there is an even number of symbols a" are not FO-definable, but admit data structures for the word problem with constant update time. The characterization is expressed in algebraic terms and we could not find natural examples of parameterized string problems that would not be FO-definable, but fall under the characterization. So we refrain from giving more details and point an interested reader to [4] instead.

We now explain how to use Theorem 21 in practice on two examples: Disjoint Factors and Edit Distance. In each case, the task boils down to defining the problem in $\mathsf{FO}[\Sigma, <]$ for an appropriate alphabet Σ. We start with Disjoint Factors.

▶ **Lemma 22.** *Let $k \in \mathbb{N}$ and $\Sigma_k = [k]$. There is a sentence $\varphi_k \in \mathsf{FO}[\Sigma_k, <]$, computable from k, such that for every $w \in \Sigma_k^\star$, w is a yes-instance of* Disjoint Factors *for parameter k if and only if $w \models \varphi_k$.*

Proof. In the sentence φ_k, we first make a disjunction over all permutations $\pi\colon [k] \to [k]$. For each such π, we verify that there exist positions $x_1 < y_1 < x_2 < y_2 < \ldots < x_k < y_k$ such that for each $i \in [k]$, both at position x_i and at y_i there is symbol $\pi(i)$. It is straightforward to express this condition using an $\mathsf{FO}[\Sigma_k, <]$ sentence. ◀

By applying Theorem 21 to the language defined by sentence φ_k provided by Lemma 22, we obtain the following.

▶ **Corollary 23.** *There is a data structure for the dynamic* Disjoint Factors *problem with initialization time $\mathcal{O}_k(n)$, worst-case update time $\mathcal{O}_k(\log\log n)$, and query time $\mathcal{O}(1)$.*

Note here that the query time can be a constant independent of k, as we can always recompute the answer to the query following every update.

For Edit Distance, the formula is more complicated. For two words $u, v \in \Sigma^\star$, by $u \otimes v$ we denote the word over $(\Sigma \cup \{\bot\})^2$, where $\bot$ is a symbol not present in Σ, defined as follows:

- The length of $u \otimes v$ is $\max(|u|, |v|)$.
- For each $1 \leqslant i \leqslant \min(|u|, |v|)$, we put $u \otimes v[i] = (u[i], v[i])$.
- For each $\min(|u|, |v|) < i \leqslant \max(|u|, |v|)$, we put $u \otimes v[i] = (u[i], \bot)$ or $u \otimes v[i] = (\bot, v[i])$, depending on whether $\max(|u|, |v|) = |u|$ or $\max(|u|, |v|) = |v|$.

▶ **Lemma 24.** *Let $k \in \mathbb{N}$ and Σ be a finite alphabet. There is a sentence $\psi_{k,\Sigma} \in \mathsf{FO}[(\Sigma \cup \{\bot\})^2, <]$, computable from k and Σ, such that for all $u, v \in \Sigma^\star$, we have $\mathsf{ed}(u, v) \leqslant k$ if and only if $u \otimes v \models \psi_{k,\Sigma}$.*

Proof. Denote $\Gamma = (\Sigma \cup \{\bot\})^2$ for brevity. Note that for two words $u', v' \in \Sigma^\star$ we have $\mathsf{ed}(u', v') \leqslant k$ if and only if there exist integers $a, b, c \geqslant 0$ with $a + b + c \leqslant k$ such that one can remove a positions from u' and b positions from v' so that the resulting strings have equal length and differ on exactly c positions. In such case, we will call the pair (u', v') (a, b, c)-*editable*.

For all $(a, b, c) \in \{0, 1, \ldots, k\}^3$ with $a+b+c \leqslant k$ and $s, t \in \{-k, \ldots, k\}$, we shall construct a formula $\alpha_{s,t,a,b,c}(x, y)$ that satisfies the following: for two positions $1 \leqslant x \leqslant y \leqslant \max(|u|, |v|)$, we have

$$u \otimes v \models \alpha_{s,t,a,b,c}(x, y) \qquad \text{if and only if} \qquad (u[x : y], v[x + s : y + t]) \text{ is } (a, b, c)\text{-editable.}$$

Here, we use the convention that if the specified range $[x : y]$ or $[x + s : y + t]$ does not fit into the corresponding word, or makes no sense due to $x > y + 1$ or $x + s > y + t + 1$, then $\alpha_{s,t,a,b,c}(x, y)$ should be false (this can be easily recognized in $\mathsf{FO}[\Gamma, <]$). If we achieve the above, the formula $\psi_{k,\Sigma}$ can be defined as the disjunction of all formulas $\alpha_{0,t,a,b,c}(1, n_u)$ for a, b, c as above, where 1 is the first position, n_u is the last position of u, and $t \in \{-k, \ldots, k\}$ is such that $n_u + t$ is the last position of v (all these are easily definable from $u \otimes v$ in $\mathsf{FO}[\Gamma, <]$).

The construction is by induction on $a + b$. For the base case $a = b = 0$, we may define $\alpha_{s,t,0,0,c}(x, y)$ as follows: if $s \neq t$ then the formula is always false, and otherwise it checks whether there are exactly c different positions z such that $x \leqslant z \leqslant y$ and $u[z] \neq v[z + s]$. This can be checked by comparing the first coordinate of $u \otimes v[z]$ with the second coordinate of $u \otimes v[z + s]$. Since $|s| \leqslant k$ by assumption, it is straightforward to formulate this assertion in $\mathsf{FO}[\Gamma, <]$.

We proceed to the induction step. So assume $a + b > 0$, say $a > 0$; the construction in the case $b > 0$ is analogous, so we omit it. The idea is that we guess, by existential quantification, the first position in $u[x : y]$ that gets removed, and use simpler formulas given by the induction assumption. More precisely, $\alpha_{s,t,a,b,c}(x, y)$ can be defined as the conjunction of formulas

$$\exists z.\, (x \leqslant z \leqslant y \wedge \alpha_{s,r,0,b_1,c_1}(x, z - 1) \wedge \alpha_{r-1,t,a-1,b_2,c_2}(z + 1, y))\,,$$

for all integers $b_1, b_2 \geqslant 0$ with $b_1 + b_2 = b$, $c_1, c_2 \geqslant 0$ with $c_1 + c_2 = c$, and $r \in \{-k+1, \ldots, k\}$. Here, $z - 1$ and $z + 1$ are a syntactic sugar for the predecessor and the successor of z, respectively, which are easily definable in $\mathsf{FO}[\Gamma, <]$. It is straightforward to see that the construction of $\alpha_{s,t,a,b,c}(x, y)$ as above satisfies the required properties. ◀

Similarly as before, by combining Theorem 21 with Lemma 24 we obtain the following.

▶ **Corollary 25.** *There is a data structure for the dynamic* Edit Distance *problem with initialization time $\mathcal{O}_{k,\Sigma}(n)$, worst-case update time $\mathcal{O}_{k,\Sigma}(\log \log n)$, and query time $\mathcal{O}(1)$.*

B Omitted proofs

Proof of Lemma 11. We assume the familiarity with the proof of Lemma 7, as our data structure will extend the one proposed there. Recall that in the context of Lemma 11, we maintain the data structure of Lemma 7 for the dictionary $\widetilde{\mathcal{S}}$ and $\tilde{o}$ is the maintained word.

The data structure of Lemma 7 stores, for every $\tilde{s} \in \widetilde{\mathcal{S}}$, the following set:

$$\Delta(\tilde{o}, \tilde{s}) = \{i \in [L] \mid \tilde{o}[i] \neq \tilde{s}[i]\}.$$

As, we have a fixed coloring $\pi \colon [L] \to [16d]$, we can maintain a table of counters

$$\mathcal{T}(\tilde{s}, c) \coloneqq |\{i \in [L] \mid \pi(i) = c \text{ and } i \in \Delta(\tilde{o}, \tilde{s})\}| \qquad \text{for all } \tilde{s} \in \widetilde{\mathcal{S}} \text{ and } c \in [16d].$$

Upon initialization, we explicitly compute $\mathcal{T}(\tilde{s}, c)$ for every $\tilde{s} \in \tilde{S}$ and $c \in [16d]$ in $\mathcal{O}(nL + nd)$ time. During an update to the position $i \in [L]$ of a word $\tilde{s} \in \widetilde{\mathcal{S}}$ we check if $\tilde{o}[i] \neq \tilde{s}[i]$ and update the number of colors in $\mathcal{T}(\tilde{s}, c)$ based on $\pi(i)$ accordingly. It may happen that the update to $\tilde{s}$ at position i triggered a modification of the word $\tilde{o}$ at position i. Then we need to change $\mathcal{T}(\tilde{s}, \pi(i))$ for every $\tilde{s} \in \widetilde{\mathcal{S}}$. Note that this alone requires $\mathcal{O}(|\mathcal{S}|)$ time. However, recall that in the proof of Lemma 7 we argued that before update when the position i of the word o changes, there were at least $|\widetilde{\mathcal{S}}|/4$ updates to that position where $\tilde{o}$ was not modified (recall here that we work over the binary alphabet). Therefore, as in the proof of Lemma 7, we may charge the running time $\mathcal{O}(|\mathcal{S}|)$ to those previous updates to argue that the amortized update time is $\mathcal{O}(1)$.

Observe that based on the table $\mathcal{T}$, we can compute the set $\mathsf{colors}_{\tilde{o},\pi}(\tilde{s}) = \{\pi(i) \mid i \in [L] \text{ and } \tilde{s}[i] = \tilde{o}[i]\}$ for any given $\tilde{s}$ in time $\mathcal{O}(d)$, because it is enough to iterate through all $c \in [16d]$ and check whether $\mathcal{T}(\tilde{s}, c) > 0$. Now we describe how to maintain the sets

$$\Phi(C) = \{\tilde{s} \in \widetilde{\mathcal{S}} \mid \mathsf{colors}_{\tilde{o},\pi}(\tilde{s}) = C\} \qquad \text{for every } C \subseteq [16d].$$

Each set $\Phi(C)$ is stored as a doubly-linked list of pointers to words from $\widetilde{\mathcal{S}}$. Upon initialization, we iterate over every $\tilde{s} \in \tilde{S}$, lookup the value of $C_{\tilde{s}} \coloneqq \mathsf{colors}_{\tilde{o},\pi}(\tilde{s})$ and add a pointer to $\tilde{s}$ to the list $\Phi(C_{\tilde{s}})$. Observe, that this operation can be implemented in total time $2^{\mathcal{O}(d)} + nL$ time, as we can compute sets $C_{\tilde{s}}$ for all $\tilde{s} \in \widetilde{\mathcal{S}}$ in total time $\mathcal{O}(nL)$.

Next, when a word $\tilde{s}$ is updated on some position and is changed to $\tilde{s}'$, we compute the previous set of colors $C \coloneqq \mathsf{colors}_{\tilde{o},\pi}(\tilde{s})$ and the new set of colors $C' \coloneqq \mathsf{colors}_{\tilde{o},\pi}(\tilde{s}')$. As argued, this operation can be done in $\mathcal{O}(d)$ time. Next, we delete the pointer to the word $\tilde{s}$ from the list $\Phi(C)$ and append a pointer to $\tilde{s}'$ to list $\Phi(C')$. Alongside $\tilde{s}$ we store the pointer to its list entry in the list $\Phi(C')$ in order to be able to remove it efficiently. Both of these operations can be implemented in $\mathcal{O}(1)$ time. During a query $C \subseteq [16d]$ we return any element from the list $\Phi(C)$ or assert that it is empty in $\mathcal{O}(1)$ time. ◀