Approximate Pattern Matching is Expressible in Transitive Closure Logic

Kjell Lemström*
Department of Computer Science
P.O.Box 26, 00014 University of Helsinki
FINLAND
klemstro@cs.helsinki.fi

Lauri Hella†
Department of Mathematics
P.O.Box 4, 00014 University of Helsinki
FINLAND
lauri.hella@helsinki.fi

Abstract

A sartorial query language facilitates the formulation of queries to a (string) database. One step towards an implementation of such a query language can be taken by defining a logical formalism expressing a known solution for the particular problem at hand. The simplicity of the logic is a desired property, because the simpler the logic that the query language is based on, the more efficiently it can be implemented. We introduce a logical formalism for expressing approximate pattern matching. The formalism uses properties of the dynamic programming *approach; a* minimizing path *of a dynamic programming table is expressed by using a formula in an extension of first-order logic* (FO). *We consider the well-known problems of* k mismatches *and* k differences. *Assuming first that k is given as a part of the input, those problems are expressed by using deterministic transitive closure logic* (FO(DTC)) *and transitive closure logic* (FO(TC))*, respectively. We believe that in the general case the k differences is not expressible in* FO(DTC)*, and show that solving this question in the affirmative is at least as hard as separating* LOGSPACE *from* NLOGSPACE. *We show, however, that if k is fixed, the k differences problem can be expressed by an* FO(DTC) *formula.*

1 Introduction

General approximate string matching has been well studied during the last couple of decades (for an exposition of the basic methods and algorithms we refer to [2, 6]). The algorithms have been applied to diverse research fields. Multimedia information retrieval is one active and wide area which is applying these methods for locating and retrieving information in large databases, such as biosequence (e.g. [1]), text, image (e.g. [12]) and music databases (e.g. [9]). *Edit distance* is a widely used metrics for exposing the dissimilarity between two strings. It measures how many *editing operations* are needed to convert string A to be similar to string B. The calculation of edit distance can be done, e.g., by using *dynamic programming*.

In order to express approximate string matching, we present a formalism based on *transitive closure logic* (FO(TC)). FO(TC) is obtained by adding an operation that forms the transitive closure of definable relations, to first-order logic (FO). In a special case of the approximate matching we use *deterministic transitive closure logic* (FO(DTC)). The data complexities for FO(TC) and FO(DTC) are $\mathrm{NLOGSPACE}$ and $\mathrm{LOGSPACE}$, respectively. Thus, for a fixed FO(TC) (FO(DTC)) sentence there is a *Turing machine*, which takes a finite model as an input. Moreover, the machine can decide whether or not the sentence is true in the input model by using nondeterministic (deterministic), logarithmic space. On ordered models the converse is also true; every $\mathrm{NLOGSPACE}$ ($\mathrm{LOGSPACE}$) computable property is expressible in FO(TC) (FO(DTC)).

In this study, we concentrate on the problem of finding approximate patterns in strings. In our formalism, given two strings, a *pattern* and a *text*, an FO(TC) formula expresses one of the minimizing paths that leads to an *approximate occurrence*, that is, at most k editing operations are needed to obtain a substring of the text by editing the pattern. If the only allowed editing operation is *replacement* (the k mismatches case), the formalism is straightforward and FO(DTC) can be used. However, the formalism becomes somewhat more challenging if *deletion* and *insertion* operations are also allowed (the k differences case): If k is given as a part of the input, the matching is expressed by using an FO(TC) formula. It is not certain, however, that this problem could not be expressed by FO(DTC), but proving that such a formula does not exist is at least as difficult as to show that NLOGSPACE $\neq$ LOGSPACE. On the other hand, using a fixed k, we can prove that the matching is expressible in FO(DTC).

We also derive upper and lower bounds for the complex-

*The research was supported by the Finnish Cultural Foundation.

†Senior Research Fellow at the Academy of Finland. The research was supported by grant #28139 from the Academy of Finland.

ity of the approximate pattern matching problems. In particular, we show that in the k differences case the problem can be solved by a Turing machine in $O((\log n)^2)$ space. In the case of k mismatches this is improved to an $O(\log n)$ upper bound on space complexity. These upper bounds hold in the usual complexity theoretic setting in which instances are strings that are directly used as inputs of Turing machines. Changing to our model theoretic formalism in which instances are models, we obtain a tightly matching lower bound for the k mismatches case: there is a LOGSPACE complete problem that is reducible to the k mismatches problem. However, this lower bound result is sensitive to the way it is formulated in the model theoretic setting: we show that after a small modification in the formalism, the k mismatches problem can be expressed in first-order logic with counting quantifiers, FO(COUNT), and hence the problem is in the complexity class ThC^0 (the class of problems computable by uniform sequences of polynomial size constant depth circuits with threshold gates).

Assessing the complexity of those approximate pattern matching problems and finding the simplest logics to express them, are already interesting in themselves. However, expressing the problems in a logic can be seen as a step towards constructing a string database query language (see e.g. [5]). In [11] Nykänen used *modal logic* as a starting point for expressing queries to string databases. For this purpose, he extended the basic modal logic with special features, concluding with rather expressive language.

The rest of this paper is organized as follows. We continue in the next section with a brief presentation of the concepts and tools that are essential in the formalism: edit distance, dynamic programming, first-order logic, transitive closure logic, and first-order logic with counting. In Section 3, we first present a nondeterministic version of the k differences algorithm, that will be the starting point to our formalisms, and then define our model. In Sections 4 and 5 we consider cases where k is given as a part of the input, and study how the logic tools should be used to express the k mismatches and k differences problems, respectively. Section 6 describes how the k differences problem can be expressed by FO(DTC) logic if the k is fixed in advance. Before concluding the paper in Section 8, we give estimations for the complexity of finding patterns using string matching techniques in Section 7.

2 Concepts and Tools

2.1 Edit Distance

Let $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$ be strings of an alphabet Σ, and let character λ represent the *empty sequence*, that is, the string of length zero. If a string A is of form $A = \beta\alpha\gamma$, for some β and γ, we say that α is a *factor* (substring) of A. A string A' is a *subsequence* of A if it can be obtained from A by deleting zero or more symbols, i.e., $A' = a_{i_1} a_{i_2} \dots a_{i_m}$, where $i_1 \dots i_m$ is an increasing sequence of indices in A. We denote by $pre_j(A)$ the prefix $a_1 \dots a_j$ of A. The *editing operations* are defined as follows:

E1 *Replacing* the symbol a_j by another symbol b gives: $a_1 \dots a_{j-1} b\, a_{j+1} \dots a_m$,

E2 *Inserting* a symbol b at position j gives: $a_1 \dots a_j b\, a_{j+1} \dots a_m$,

E3 *Deleting* the symbol a_j at position j gives: $a_1 \dots a_{j-1} a_{j+1} \dots a_m$,

Now the edit distance function can be defined as follows.

Definition 1 *The edit distance between strings A and B is the minimum number of allowed editing operations required to convert A into B.*

Another way to view the distance is to split A and B into equally many parts, where some of the parts may be empty sequences. Let $A = \alpha_1 \dots \alpha_p$ and $B = \beta_1 \dots \beta_p$ be such a splitting (also called a *trace*). Every trace suggests a way to obtain the string B from A by using *local transformations*, comprising the editing operations (E1–E3) and an identity (ID) operation (which can be used for free):

ID Identity: a_j is preserved: $a_j = b_i$.

A trace using the minimum number of editing operations (of all possible traces) gives the value for the edit distance $D(A, B)$. Moreover, the value equals the number of those required editing operations. The *Hamming distance* $D_H(A, B)$ can be calculated by allowing only editing operation E1 and the identity operation ID.

2.2 Dynamic Programming and k Similarity

In this paper, we are interested in the problem of finding *k similar* occurrences of a given pattern P ($P = p_1 \dots p_m$) within text (or source) S ($S = s_1 \dots s_n$), where k denotes the maximum allowed distance between the pattern and its occurrence in S. Typically, the pattern is very short with respect to the length of the text, that is, $m << n$.

Definition 2 *Pattern P has a k similar occurrence in S if there is a factor P' of S such that $D(P, P') \leq k$.*

The pattern matching problems associated with Hamming and edit distances are called k mismatches and k differences problems, respectively. Thus, when searching occurrences with k mismatches, only E1 and ID should be allowed, while in the case of k differences the operations E2 and E3 should be allowed, as well.

The occurrences of P within text S can be located by using dynamic programming technique [15]. The dynamic programming tabulates the edit distances between all the possible prefixes of A and any factor of B: $d_{ij} = \min\{D(pre_j(A), b_r \ldots b_i) \mid 1 \leq r \leq i+1\}$. Examples of such calculations can be found in Figs. 1 and 2 (corresponding to the cases of k mismatches and k differences, respectively). Letting $1 \leq i \leq n, 1 \leq j \leq m$, the k differences algorithm is as follows:

Algorithm 1 (**k differences**)
begin
1. **for** $j := 0$ **to** m **do** $d_{0j} = j$;
2. **for** $i := 1$ **to** n **do begin**
3. $\quad d_{i0} := 0$;
4. $\quad$ **for** $j := 1$ **to** m **do**
5. $\quad\quad d_{ij} := \min\{d_{i-1,j-1} + (\textbf{if } p_j = s_i \textbf{ then } 0 \textbf{ else } 1),$
$\quad\quad\quad\quad d_{i-1,j} + 1, d_{i,j-1} + 1\}$;
6. $\quad$ **if** $d_{im} \leq k$ **then** $\texttt{write}(d_{im}, i)$;
7. **end**
end

A *backtracking path* in a dynamic programming table starts from row m and ends at row 0 (see Figs. 1 and 2), and advances step by step (possibly not unique) according to the minimizing operation (line 5 of Algorithm 1). A *path* is a backtracking path whose direction is reversed.

Definition 3 (Minimizing path) *A Minimizing path is a path that ends in cell $d_{n'm}$ and $d_{n'm} = \min\{d_{im} \mid 1 \leq i \leq n\}$.*

Theorem 1 *[14] Let there be a minimizing path ending in cell d_{im} and starting from d_{r0}. The edit distance $D(P, s_{r+1} \ldots s_i) = d_{im}$ and $d_{im} = \min\{D(P, s_t \ldots s_i) \mid t \leq i+1\}$.*

Obviously, Theorem 1 holds for D_H distance, as well. The approximate occurrences of P within S can be found by observing the bottom row of the table. Due to Theorem 1 above, the value of a cell in that bottom row represents the minimum number of allowed editing operations used to convert P to be similar to any suffix of $pre_i(S)$. We call such a minimizing path *accepting* if it is ending in cell d_{im} and $d_{im} \leq k$.

If the starting index of an accepting path is needed, a procedure that resolves the minimizing path has to be called in the **if** statement on line 6 of Algorithm 1. Clearly, both the time and space requirements (with respect to the RAM model [17]) for this basic algorithm are $O(mn)$. In an application where the exact starting point of an occurrence is not needed, it is enough to maintain space for $m+1$ cells of the dynamic programming table; The values in the previous column are always overwritten by the values of the present column as the calculation proceeds column-by-column. The cell above the adjacent cell in the previous column (corresponding to the replacement or identity) should be handled with care, and stored to a temporary variable. Moreover, the time requirement can also be diminished; it has been subsequently refined, first to $O(kn)$ expected time [16], and then to $O(kn)$ worst-case time [8] [1].

2.3 First-Order Logic

Vocabularies $\pi, \sigma \ldots$ are finite sets consisting of *relation symbols* $R_0, R_1, \ldots$, and *constant symbols* $c_0, c_1, \ldots$. Every relation symbol R_i has a special natural number n_i that is attached to it, namely its *arity*. A *structure* (*model*) $\mathcal{A}$ of a vocabulary π is a tuple:

$$\mathcal{A} = \langle A, R_0^{\mathcal{A}}, R_1^{\mathcal{A}}, \ldots, c_0^{\mathcal{A}}, c_1^{\mathcal{A}}, \ldots \rangle,$$

where the nonempty set A is the *universe* of $\mathcal{A}$. Furthermore, for every relation symbol R_i in π, $R_i^{\mathcal{A}}$ is an n_i-ary relation on A, and for every constant c in π, $c_i^{\mathcal{A}}$ is an element of A.

The set of first-order formulas over a vocabulary π is defined, as usually, by starting with atomic formulas and closing under negations $\neg\varphi$, disjunctions $\varphi \vee \psi$, and existential quantifications $\exists x \varphi$. An atomic formula over π is an expression of the form $R(t_1, \ldots, t_n)$ or $(t_1 = t_2)$, where $R \in \pi$ is an n-ary relation symbol and $t_1, \ldots, t_n$ are either variables or constant symbols in π. Other connectives (conjunction $\wedge$, implication $\rightarrow$, equivalence $\leftrightarrow$) and the universal quantifier ($\forall$) are defined in terms of $\neg$, $\vee$ and $\exists$ in the obvious way.

The semantics of first-order logic is defined in the standard way using Tarski's truth definition. If $\varphi(\overline{x})$ is a formula over a vocabulary π, $\mathcal{A}$ is a π-structure, and $\overline{a}$ is a tuple of elements of A such that $\varphi(\overline{x})$ is true in $\mathcal{A}$ when the variables in $\overline{x}$ are interpreted by the corresponding components of $\overline{a}$, then we write $\mathcal{A} \models \varphi[\overline{a}]$.

2.4 Transitive Closure Logic

Next we give a brief review of logics obtained by augmenting first-order logic with transitive closure operators. For more detailed expositions, we refer the reader to the books [3] and [7]. We start by defining the transitive closure and deterministic transitive closure operators TC and

[1] The time complexity can be further refined by applying the hardware word-level parallelism of bit-vector operations. One of the most recent and fastest algorithms of such an approach runs in time $O(nm/w)$ [10], where w denotes the size of the computer word in bits.

DTC. Let $h \geq 1$ and R be a $2h$-ary relation on a set A. Now $\mathrm{TC}(R)$ is defined as follows:

$$\begin{aligned}\mathrm{TC}(R) := \{(\overline{a}, \overline{b}) \in A^{2h} \mid & \text{ there exist } n > 0 \text{ and} \\ & \overline{d}_0, \ldots, \overline{d}_n \in A^h \text{ such that} \\ & \overline{a} = \overline{d}_0, \overline{b} = \overline{d}_n, \text{ and} \\ & \text{for all } i < n, (\overline{d}_i, \overline{d}_{i+1}) \in R\}.\end{aligned}$$

Thus, $\mathrm{TC}(R)$ consists of all pairs $(\overline{a}, \overline{b})$ such that there is an R-path from $\overline{a}$ to $\overline{b}$ (viewing R as a directed graph on h-tuples of A). The deterministic transitive closure $\mathrm{DTC}(R)$ of R is defined similarly, except that the R-path may not branch.

$$\begin{aligned}\mathrm{DTC}(R) := \{(\overline{a}, \overline{b}) \in A^{2h} \mid & \text{ there exist } n > 0 \text{ and} \\ & \overline{d}_0, \ldots, \overline{d}_n \in A^h \text{ such that} \\ & \overline{a} = \overline{d}_0, \overline{b} = \overline{d}_n, \text{ and} \\ & \text{for all } i < n, \overline{d}_{i+1} \text{ is the unique} \\ & \overline{d} \text{ for which } (\overline{d}_i, \overline{d}) \in R\}.\end{aligned}$$

Transitive closure logic, FO(TC), is the extension of first-order logic that is obtained by adding the new formula formation rule:

- if φ is a formula then $[TC_{\overline{x},\overline{y}}\varphi](\overline{s}, \overline{t})$ is a formula,

where the variables in $\overline{x}, \overline{y}$ are pairwise distinct and tuples $\overline{x}, \overline{y}, \overline{s}$ and $\overline{t}$ are all of the same length ($\overline{s}$ and $\overline{t}$ are tuples of terms). Here the operator $TC_{\overline{x},\overline{y}}$ binds all occurrences of the variables $\overline{x}$ and $\overline{y}$ in φ, whereas the variables in $\overline{s}$ and $\overline{t}$ are free in the new formula.

The semantics of FO(TC) is defined by adding the following rule to the truth definition of first-order logic:

- $\mathcal{A} \models [TC_{\overline{x},\overline{y}}\varphi][\overline{a}, \overline{b}]$, iff, $(\overline{a}, \overline{b}) \in \mathrm{TC}(\varphi^{\mathcal{A}})$, where $\varphi^{\mathcal{A}} = \{(\overline{c}, \overline{d}) \mid \mathcal{A} \models \varphi[\overline{c}, \overline{d}]\}$.

(Here we have suppressed free variables of φ other than those in $\overline{x}$ and $\overline{y}$.)

Deterministic transitive closure logic, FO(DTC), is defined in the same way as FO(TC), except that in the syntax the operator $TC_{\overline{x},\overline{y}}$ is replaced by $DTC_{\overline{x},\overline{y}}$, and the semantics is given by the deterministic transitive closure of relations.

It is a well-known fact that FO(DTC) and FO(TC) capture the complexity classes $\mathrm{LOGSPACE}$ and $\mathrm{NLOGSPACE}$, respectively (see, e.g., [3, 7]). In other words, for every FO(DTC) sentence there is a $\mathrm{LOGSPACE}$ Turing machine which takes a model as an input, and decides the truth of the sentence in the given model. Similarly, there is a $\mathrm{NLOGSPACE}$ Turing machine for every FO(TC) sentence. Furthermore, on ordered models FO(DTC) $=$ $\mathrm{LOGSPACE}$ and FO(TC) $=$ $\mathrm{NLOGSPACE}$. Thus the inverse directions are also valid if the vocabulary of the models considered contains a special binary relation symbol which is always interpreted as a linear order. Actually, it suffices that the models are equipped with a successor relation, since the corresponding linear order is easily definable by a formula of FO(DTC) from its successor relation.

2.5 First-Order Logic with Counting

Another way of increasing the expressive power of first-order logic is to add counting mechanisms to it. First-order logic with counting, FO(COUNT), is usually defined on two-sorted structures where the second sort consists of an initial segment $\{0, \ldots, n-1\}$ of the natural numbers equipped with order, addition and multiplication (see [7]). Counting power is introduced by the new formula formation rule:

- if $\varphi(x)$ is a formula then $\exists^i x\varphi(x)$ is a formula,

where x is a variable over the first sort, and i is a variable over the second (numerical) sort. Note that the counting quantifier $\exists^i x$ binds x, whereas i becomes a new free variable in the formula $\exists^i x\varphi(x)$. The semantics of the counting quantifiers is given by the rule:

- $\mathcal{A} \models \exists^i x\varphi(x)$, iff, the number of elements a in the first sort of $\mathcal{A}$ such that $\mathcal{A} \models \varphi[a]$ is at least (the interpretation of the variable) i.

Using the arithmetic functions on the numerical sort, FO(COUNT) can express all kinds of counting properties. For example, the sentence

$$\exists i(\exists j(i = j + j) \wedge \exists^i xR(x) \wedge \neg\exists^{i+1} xR(x))$$

says that the relation R contains an even number of elements.

The logic FO(COUNT) is also known to capture a complexity class; every problem expressible in FO(COUNT) is in ThC^0, and on ordered models FO(COUNT) $= \mathrm{ThC}^0$ (see [7]). Here ThC^0 is the class of all problems computable by uniform sequences of polynomial size constant depth circuits with threshold gates. The class ThC^0 is contained in LOGSPACE, but it is not known whether the containment is strict.

3 On Expressing an Occurrence

Let $P = p_1 \ldots p_m$ and $S = s_1 \ldots s_n$ be the pattern and text, respectively. Without loss of generality we assume in the sequel that $n \geq m$. We often omit parentheses in formulas when there is no danger of incorrect interpretation.

3.1 Nondeterministic k Differences Algorithm

Let us assume a nondeterministic function $\texttt{guess}(p_1, p_2)$ that gets as input the lower and upper bounds of a subset of $\mathbb{N}$. The function foresees the correct value within the given domain and returns it. The nondeterministic version of Algorithm 1 is as follows:

Algorithm 2 (nondeterministic k differences)

```
begin
    i := guess(1, n);
    count := 0; j := 0;
    repeat
        operation := guess(1, 4);
        if not (operation = 3) then j := j + 1;
        if not (operation = 4) then i := i + 1;
        if not ((operation = 1) and (p_j = s_i)) then
            count := count + 1;
    until j = m;
    if count ≤ k then write (count, i);
end
```

Firstly, $\texttt{guess}$ is a function that returns the location where an accepting path starts. The cost of required editing operations are accumulated to the variable *count* and the value of *operation* corresponds to one of the possible choices: identity, replacing, inserting and deleting corresponding to $1, \ldots, 4$, respectively. One should be careful with the **if not** statements. They are used in order to avoid **if-else** structures. Thus they make the analysis shorter.

Clearly, the algorithm cannot execute the **repeat-until** loop more than $m + n$ times (corresponding to m deletions and n insertions). Thus, for the nondeterministic Algorithm 2 the worst-case time requirement with respect to the RAM model is $O(m + n)$, while the space requirement is reduced to $O(1)$.

3.2 Table Models

In order to obtain a model theoretic framework for expressing approximate pattern matching problems, we first have to encode their inputs (i.e., the strings P and S, and the error limit k) as models over some suitable vocabulary π. There is a standard way of encoding strings in Σ^* into models over a vocabulary consisting of a binary relation symbol Sc, and a unary relation symbol P_a for each $a \in \Sigma$: a string $B = b_1 \ldots b_n$ is encoded by the model $\mathcal{A}_B = \langle [n], Sc, (P_a^B)_{a \in \Sigma} \rangle$, where $[n] = \{1, \ldots, n\}$, $Sc = \{(i, i+1) \mid 1 \leq i < n\}$ is the successor relation on $[n]$, and $P_a^B = \{i \mid b_i = a\}$.

Note that in this encoding the vocabulary of the models $\mathcal{A}_B$ depends on the alphabet Σ. However, for our purposes it is better to use a more uniform encoding. Hence we will represent the symbols in Σ by elements of the model, which means that the strings P and S can be naturally represented as (partial) functions of the model.

More precisely, given an alphabet Σ, we fix an injective mapping $\sigma : \Sigma \to \mathbb{N}$. The **table model** $\mathcal{A}_{P,S,k}$ for pattern P, text S and error limit k is the model $\langle A, Sc, \mathrm{f_p}, \mathrm{f_s}, \mathrm{k} \rangle$ over a vocabulary with three binary relation symbols and one constant symbol, where the following hold:

- $A = \{0, 1, \ldots, r\}$ where r is the maximum of n and $\sigma(a)$, $a \in \Sigma$,
- Sc is the successor relation on A,
- $\mathrm{f_p} = \{(1, a_1), \ldots, (m, a_m)\}$ where $a_1 = \sigma(p_1), \ldots, a_m = \sigma(p_m)$,
- $\mathrm{f_s} = \{(1, b_1), \ldots, (n, b_n)\}$ where $b_1 = \sigma(s_1), \ldots, b_n = \sigma(s_n)$,
- k is the allowed maximum for edit distance.

(We assume here without loss of generality that $\mathrm{k} \leq r$.) Note that n and m are easily definable from the relations $\mathrm{f_s}$ and $\mathrm{f_p}$ in first-order logic. Moreover, the natural ordering $\leq$ of A is definable from Sc by the FO(DTC) formula $[DTC_{u,v}\, Sc(u,v)](x, y) \vee x = y$. In the sequel we will often regard Sc as a function; thus $Sc(x)$ denotes the successor of x.

Now we are ready to introduce a few abbreviations which we will use to clarify the eventual formula. Please note that we denote by *plain terms* $s, t, \ldots, z$ elements in A, while the overlined versions are used for tuples $\overline{s}, \overline{t}, \ldots, \overline{z} \in A^3$.

- $\triangle(\overline{p}, \overline{q}) \Leftrightarrow ((q_1 = p_1) \wedge (q_2 = Sc(p_2)))$,
- $\triangleleft(\overline{p}, \overline{q}) \Leftrightarrow ((q_2 = p_2) \wedge (q_1 = Sc(p_1)))$,
- $\diamond(\overline{p}, \overline{q}) \Leftrightarrow ((q_1 = Sc(p_1)) \wedge (q_2 = Sc(p_2)))$,

Thus, the intuitive meaning of $\triangle(\overline{p}, \overline{q})$ is that $\overline{p}$ is the cell immediately above the cell $\overline{q}$ in a dynamic programming table. Similarly $\triangleleft(\overline{p}, \overline{q})$ ($\diamond(\overline{p}, \overline{q})$) says that $\overline{p}$ is the previous cell on the same row (on the same diagonal, respectively) as the cell $\overline{q}$.

4 Expressing the Case of k Mismatches

Let us start our formalism with a straightforward case where the approximate occurrences of a pattern are searched by using the k mismatches approach. This case shows itself to be a special case of our formalism; it can be expressed with an FO(DTC) formula. In Fig. 1, we have given an example of k differences calculation by dynamic programming, when $P = \texttt{abc}$, $S = \texttt{ddadc}$ and $k = 1$. In

		d	d	a	d	c	
	0	0	0	0	0	0	0
a	1	1	1	0	1	1	1
b	2	2	2	2	1	2	2
c	3	3	3	3	3	1	3
	0	1	2	3	4	5	

Figure 1. Example of dynamic programming with k mismatches (k=1).

the example, there is one accepting path, which is emphasized.

In this case a combination of Algorithms 1 and 2 is to be used:

Algorithm 3 (nondeterministic k mismatches)
begin
1. $i :=$ `guess`$(1, n)$*;*
2. count $:= 0$*;*
3. **for** $j := 1$ **to** m **do**
4. *count* $:=$ *count* $+$(**if** $a_j = b_{i+j-1}$ **then** 0 **else** 1)*;*
5. **if** *count* $\leq k$ **then** `write`(*count*$, i + m - 1$);
end

Although Algorithm 3 is nondeterministic this nondeterminism is weak; the guess on line 1 can be expressed by using an existential quantifier.

Theorem 2 *There is an* $\mathrm{FO(DTC)}$ *sentence which is true in the table model $\mathcal{A}_{P,S,k}$ if and only if the corresponding pattern P has a k mismatches occurrence within the corresponding text S.*

Proof Let us start by giving an FO(DTC) sentence that is established according to Algorithm 3. After that we analyze it. The formula is as follows:

$$\Phi_{\mathrm{m}}{=}\exists\overline{x}\exists\overline{y}([DTC_{\overline{s}\overline{t}}\varphi_{\mathrm{m}}(\overline{s},\overline{t})](\overline{x},\overline{y}) \wedge (x_2{=}0) \wedge \quad (1)$$

$$(x_3{=}0) \wedge (y_2{=}m) \wedge (y_3{\leq}k)), \quad (2)$$

$$\text{where } \varphi_{\mathrm{m}}(\overline{s},\overline{t}) = \exists u_1 \exists u_2 \exists v_1 \exists v_2 ((\diamond(\overline{s},\overline{t})) \wedge \quad (3)$$

$$\mathrm{f_p}(u_1, u_2) \wedge (u_1 = t_2) \wedge \quad (4)$$

$$\mathrm{f_s}(v_1, v_2) \wedge (v_1 = t_1) \wedge \quad (5)$$

$$(((u_2 = v_2) \wedge (t_3 = s_3)) \vee \quad (6)$$

$$(\neg(u_2 = v_2) \wedge (t_3 = Sc(s_3))))). \quad (7)$$

Lines (1–2) represent the main part of the formula: it checks that the two ends of the path are in accordance with the problem setting. The transition formula $\varphi_{\mathrm{m}}(\overline{s},\overline{t})$ is composed of the *alignment* (4–5) and *action* parts (6–7). In the alignment part, the advance direction is at first bound. In this case only the southeast direction is accepted ($\diamond(\overline{s},\overline{t})$), thus the path is always advanced in the considered diagonal. Then the associated characters within P and S are bound; bounding the character in P, the horizontal alignment is used ($u_1 = t_2$), analogously ($v_1 = t_1$) aligns the corresponding character in S vertically. In the action part, the formula compares whether or not the corresponding characters u_2 of P and v_2 of S are the same and assigns a value for t depending on the result of the comparison.

Note that the formula $\varphi_{\mathrm{m}}(\overline{s},\overline{t})$ determines $\overline{t}$ uniquely from $\overline{s}$: $\varphi_{\mathrm{m}}(\overline{s},\overline{t}) \wedge \varphi_{\mathrm{m}}(\overline{s},\overline{t'}) \to \overline{t} = \overline{t'}$. Thus, any φ_{m}-path is automatically non-branching.

Assume now that $\mathcal{A}_{P,S,k} \models \Phi_{\mathrm{m}}$. Then there are tuples $\overline{a}_0, \ldots, \overline{a}_m \in A^3$ such that $\overline{a}_0$ is of the form $(i, 0, 0)$, $\overline{a}_m$ is of the form (i', m, r) with $r \leq k$, and $\mathcal{A}_{P,S,k} \models \varphi_{\mathrm{m}}[\overline{a}_l, \overline{a}_{l+1}]$ holds for each $l < m$. From the above analysis we see, that if $\overline{a}_l$ is a correct entry (i, j, d_{ij}) in the dynamic programming table corresponding to $\mathcal{A}_{P,S,k}$, then $\overline{a}_{l+1}$ is the next correct entry $(i + 1, j + 1, d_{i+1,j+1})$ on the same diagonal. Since $\overline{a}_0$ is a correct entry on the top row, we conclude that the third component of $\overline{a}_m$ is $r = d_{i'm}$, which is k at the most. Thus, P has a k mismatches occurrence within S.

For the converse, assume that there is a minimizing path $d_{i,0}, d_{i+1,1}, \ldots, d_{i+m,m}$ in the dynamic programming table for P and S such that $d_{i+m,m} \leq k$. For each $l \leq m$, let $\overline{a}_l = (i + l, l, d_{i+l,l})$. Then it is clear that $\mathcal{A}_{P,S,k} \models \varphi_{\mathrm{m}}[\overline{a}_l, \overline{a}_{l+1}]$ for each $l < m$. As observed above, the φ_{m}-path $\overline{a}_0, \ldots, \overline{a}_m$ is necessarily non-branching, whence $\mathcal{A}_{P,S,k} \models ([DTC_{\overline{s}\overline{t}}\varphi_{\mathrm{m}}(\overline{s},\overline{t})][\overline{a}_0, \overline{a}_m]$. As the other conjuncts of (1) and (2) are clearly satisfied by $\overline{a}_0$ and $\overline{a}_m$, we conclude that $\mathcal{A}_{P,S,k} \models \Phi_{\mathrm{m}}$. This completes our proof. □

5 Expressing the Case of k Differences

In the k differences case all the local transformations ID, E1 – E3 are allowed. Allowing operations E2 and E3, as well, implies that in the dynamic programming table there may be several accepting paths starting from a cell d_{i0}. In Fig. 2 we give an example of a k differences dynamic programming calculation, when $P =$ `aaabbb`, $S =$ `bbaaab`, and $k = 2$. The accepting paths are emphasized. Since the path is no more unique the nondeterministic `guess` function (recall it from Algorithm 2) is used. An accepting path of k differences calculation is next expressed by using FO(TC) logic.

Theorem 3 *There is an* $\mathrm{FO(TC)}$ *sentence which is true in the table model $\mathcal{A}_{P,S,k}$ if and only if the corresponding pattern P has a k differences occurrence within the corresponding text S.*

		b	b	a	a	a	b
	0	0	0	0	0	0	0
a	1	1	1	0	0	0	1
a	2	2	2	1	0	0	1
a	3	3	3	2	1	0	1
b	4	3	3	3	2	1	0
b	5	4	3	4	3	2	1
b	6	5	4	4	4	3	2

Figure 2. Example of dynamic programming with k differences (k=2).

Proof The proof is almost identical to the proof of the previous Theorem 2. Thus, we start by giving a sentence Φ_{d} of FO(TC), and then show that it captures the idea of the nondeterministic Algorithm 2 for k differences. Let

$$\Phi_{\mathrm{d}} = \exists\overline{x}\exists\overline{y}([TC_{\overline{s}\overline{t}}\varphi_{\mathrm{d}}(\overline{s},\overline{t})](\overline{x},\overline{y}) \wedge (x_2 = 0) \wedge (x_3 = 0) \wedge (y_2 = m) \wedge (y_3 \leq k)).$$

We repartition the action part of the formula into *condition* and *expense* parts, using a subscripted ψ and δ denoting the condition and expense parts, respectively. Now the transition formula is rewritten as follows:

$$\varphi_{\mathrm{d}}(\overline{s},\overline{t}) = \exists u_1\exists u_2\exists v_1\exists v_2((\mathrm{f_p}(u_1,u_2) \wedge \tag{8}$$

$$(u_1 = t_2) \wedge \mathrm{f_s}(v_1,v_2) \wedge (v_1 = t_1)) \tag{9}$$

$$\wedge((\psi_0 \wedge \delta_0) \vee (\psi_1 \wedge \delta_1) \tag{10}$$

$$\vee (\psi_2 \wedge \delta_2) \vee (\psi_3 \wedge \delta_3))). \tag{11}$$

At lines (8–9), the vertical and horizontal alignments are fixed as before, while lines (10–11) are associated with the actions. The condition parts of the formula are defined as:

$$\psi_0 = ((\Diamond(\overline{s},\overline{t})) \wedge (u_2 = v_2)), \quad \psi_2 = (\triangleleft(\overline{s},\overline{t})),$$
$$\psi_1 = ((\Diamond(\overline{s},\overline{t})) \wedge \neg(u_2 = v_2)), \quad \psi_3 = (\triangle(\overline{s},\overline{t})).$$

Thus, the condition parts correspond to the four possible steps on the path (due to the local transformations ID, E1, E2 and E3, and due to values 1, 2, 3 and 4 of variable $operation$ in Algorithm 2). Furthermore, the use of an editing operation should be 'charged' (while the use of an identity should be free of charge). Thus, the expense formulas $\delta_0, \ldots, \delta_3$ corresponding to the local transformations ID, E1 – E3, respectively, become as follows:

$$\delta_0 = (t_3 = s_3), \quad \delta_i = (t_3 = Sc(s_3)), \text{ where } i = 1,2,3.$$

It is now clear that $\mathcal{A}_{P,S,k} \models \varphi_{\mathrm{d}}$ if and only if Algorithm 2 accepts the input (P, S, k). □

We do not know whether the k differences problem could be expressed in deterministic transitive closure logic already, but it seems to us that using FO(TC) instead of FO(DTC) is really necessary. However, it should be noted that proving that the k differences problem is not expressible in FO(DTC) is at least as hard as separating NLOGSPACE from LOGSPACE. This can be seen as follows. Recall that all LOGSPACE computable properties of ordered models are expressible in FO(DTC). Since table models have an FO(DTC)-definable ordering, non expressibility in FO(DTC) would imply that the k differences problem is not in LOGSPACE. On the other hand, Theorem 3 implies that it is in NLOGSPACE.

6 Fixed k

Next we will show that even k differences problem can be expressed by using FO(DTC), if k is fixed. Thus, in this section k is not considered as a part of the input anymore. This reflects slightly on the model we use; the model $\mathcal{A}_{P,S}$ is otherwise identical to $\mathcal{A}_{P,S,k}$, but k is no more given as a constant. Note however, that since k is now a fixed integer, it can be defined by an FO formula.

In the resulting formula we harness the fact that the accepting path must be in a diagonal band [15]. The width of such a band is restricted by k; the smaller k is, the narrower the band is. The expressibility of this case in FO(DTC) is a consequence of the fact, that the width of the band is known beforehand.

		b	b	a	a	a	b
	0	0	0	0	0	0	0
a	1	1	1	0	0	0	1
a	2	2	2	1	0	0	1
a	3	3	3	2	1	0	1
b	4	3	3	3	2	1	0
b	5	4	3	4	3	2	1
b	6	5	4	4	4	3	2

Figure 3. Covering the band with L-polies.

We will cover the band with successive L-shaped polygons (see Fig. 3), which we call L-polies. We assume that $k \leq m$. The polygons are symmetric due to identical 'cost' of the editing operations, and the number of the elements in an L-poly is $2k + 1$. The 'corner' of an L-poly (the element #$k + 1$) is called the *pivot* of the L-poly, the elements within the ranges $1 - k$ and (k+2) – (2k+1) belong to the right and left tails of the L-poly, respectively.

The pivot of the first L-poly is always assigned (by an existential quantifier) to the position where the accepting path starts. As can also be seen in Fig. 3, either one of the two tails of an L-poly may exceed the border of the dynamic programming table. In order to avoid such overflows we use the following two additional successor functions (clearly they are definable in FO from the relations $\mathrm{f_s}$ and $\mathrm{f_p}$):

$$Sc_m(x) = \begin{cases} Sc(x), & \text{if } x < m; \\ x, & \text{else.} \end{cases} \tag{12}$$

$$Sc_n(x) = \begin{cases} Sc(x), & \text{if } x < n; \\ x, & \text{else.} \end{cases}$$

By using the functions above we might get a degenerate form of such a polygon, which we still call L-poly.

Moreover, in order to avoid the problems with complicated indices, we change the notation slightly: the 'coordinates' of a dynamic programming table cell are denoted by a pair α, β (corresponding to x_1 and x_2 of the previous notation), respectively, while the value $d_{\alpha\beta}$ in that particular location is denoted by γ (corresponding to the previous x_3). Furthermore, let vectors with a 'hat' (e.g. $\hat{x}$) denote tuples of the form $((\alpha_1^{\hat{x}}, \beta_1^{\hat{x}}, \gamma_1^{\hat{x}}), \ldots, (\alpha_{2k+1}^{\hat{x}}, \beta_{2k+1}^{\hat{x}}, \gamma_{2k+1}^{\hat{x}}))$.

Lemma 1 *Let k be fixed. There is an* FO *formula $\varphi_L(\hat{x})$ such that $\mathcal{A}_{P,S} \models \varphi_L[\hat{a}]$ if and only if $\hat{a}$ is an L-poly in $\mathcal{A}_{P,S}$.*

Proof Let $\varphi_L(\hat{x})$ be the formula

$$\varphi_L(\hat{x}) = (\bigwedge_{i=1}^{k} (\alpha_i^{\hat{x}} = \alpha_{i+1}^{\hat{x}}) \wedge (\alpha_{k+i+1}^{\hat{x}} = Sc_n(\alpha_{k+i}^{\hat{x}}))$$

$$\wedge (\beta_i^{\hat{x}} = Sc_m(\beta_{i+1}^{\hat{x}})) \wedge (\beta_{k+i+1}^{\hat{x}} = \beta_{k+i}^{\hat{x}})).$$

Clearly it satisfies the claim. □

Let us next define what we mean by a successor of an L-poly in our model. Consider Fig. 3. As the first L-poly is set, the pivots of the successive L-polies always lie in the same diagonal, at the adjacent southeast cells. The two tails are prevented to exceed outside the dynamic programming table by using the successor relations Sc_m and Sc_n.

Definition 4 *Let $\hat{a}$ and $\hat{b}$ be L-polies. L-poly $\hat{b}$ is a successor of $\hat{a}$ in $\mathcal{A}_{P,S}$ if and only if $\alpha_i^{\hat{b}} = Sc_n(\alpha_i^{\hat{a}})$ and $\beta_i^{\hat{b}} = Sc_m(\beta_i^{\hat{a}})$ hold for each i, $1 \le i \le 2k+1$.*

Please note that the definition above binds only the α and β values of a successor L-poly. Thus successors of L-polies are not unique. In order to pick a unique representative from the successors of an L-poly, also the γ values will be bound in the proof of Theorem 4.

Lemma 2 (L-polies' successor relation) *Let k be fixed and let $\hat{a}$ and $\hat{b}$ be L-polies. There is an* FO *formula $\varphi_{Sc}(\hat{x}, \hat{y})$ such that $\mathcal{A}_{P,S} \models \varphi_{Sc}[\hat{a}, \hat{b}]$ if and only if $\hat{b}$ is the successor of $\hat{a}$. Moreover, $((\alpha_1^{\hat{b}}, \beta_1^{\hat{b}}), \ldots, (\alpha_{2k+1}^{\hat{b}}, \beta_{2k+1}^{\hat{b}}))$ is the unique successor of $((\alpha_1^{\hat{a}}, \beta_1^{\hat{a}}), \ldots, (\alpha_{2k+1}^{\hat{a}}, \beta_{2k+1}^{\hat{a}}))$.*

Proof Let $\varphi_{Sc}(\hat{x}, \hat{y})$ be the formula

$$\varphi_{Sc}(\hat{x}, \hat{y}) = (\bigwedge_{i=1}^{2k+1} (\alpha_i^{\hat{y}} = Sc_n(\alpha_i^{\hat{x}})) \wedge (\beta_i^{\hat{y}} = Sc_m(\beta_i^{\hat{x}}))).$$

Since each $\alpha_i^{\hat{x}}$ and $\beta_i^{\hat{x}}$ has a unique successor $Sc_n(\alpha_i^{\hat{x}})$ and $Sc_m(\beta_i^{\hat{x}})$, the formula states that, with respect to tuples of (α, β) pairs, $\hat{y}$ is the unique successor of $\hat{x}$. □

Theorem 4 *Let k be fixed. There is an* $\mathrm{FO(DTC)}$ *sentence which is true in the table model $\mathcal{A}_{P,S}$ if and only if the corresponding pattern P has a k differences occurrence within the corresponding text S.*

Proof (Sketch) The desired sentence is as follows

$$\begin{aligned} \Phi_{\mathrm{f}} \;=\; & \exists \hat{x} \exists \hat{y} ([DTC_{\hat{s}\hat{t}} \varphi_{\mathrm{f}}(\hat{s}, \hat{t})](\hat{x}, \hat{y}) && (13) \\ & \wedge (\beta_{k+1}^{\hat{x}} = 0) \wedge (\gamma_{k+1}^{\hat{x}} = 0) && (14) \\ & \wedge \varphi_L(\hat{x}) \wedge \varphi_L(\hat{y}) && (15) \\ & \wedge (\bigvee_{i=1}^{2k+1} ((\beta_i^{\hat{y}} = m) \wedge (\gamma_i^{\hat{y}} \le k))) && (16) \end{aligned}$$

Line 14 of the formula Φ_{f} is almost identical to that of the formula Φ_{d}; only the indexing system has been changed. In the next line (15) the formula of the Lemma 1 is used first, to ascertain that $\hat{x}$ and $\hat{y}$ are L-polies. After that it is checked whether the accepting condition is met by one of the elements of the L-poly $\hat{y}$. Let us next give the formula $\varphi_{\mathrm{f}}(\hat{s}, \hat{t})$ inside the DTC operator:

$$\begin{aligned} \varphi_{\mathrm{f}}(\hat{s}, \hat{t}) = & \exists u_1 \ldots \exists u_{2k+2} \exists v_1 \ldots \exists v_{2k+2} && (17) \\ & ((\bigwedge_{i=1}^{k+1} \mathrm{f_p}(u_i, u_{k+i+1}) \wedge \mathrm{f_s}(v_i, v_{k+i+1})) && (18) \\ & \wedge (\varphi_{Sc}(\hat{s}, \hat{t}) \wedge (\alpha_{k+1}^{\hat{s}} < n) \wedge (\beta_{k+1}^{\hat{s}} < m) && (19) \\ & \wedge \bigwedge_{i=1}^{2k+1} \kappa_i && (20) \end{aligned}$$

The first two lines of the formula bind the corresponding subparts of the pattern and string, in an analogue with the previous formulas φ_{m} and φ_{d}. Line (19) makes sure that the L-poly $\hat{t}$ does not 'fall outside the borders' of the corresponding dynamic programming table. Finally, the formulas κ_i, $1 \leq i \leq 2k+1$ describe how the third components $\gamma_i^{\hat{y}}$ of $\hat{y}$ are computed from the third components of $\hat{x}$ and the 'characters' u_j and v_j, $k+2 \leq j \leq 2k+2$, in accordance with the priority order of local transformations given in Algorithm 1. (Note that this computation is done inductively along the tails of the L-poly, starting from the pivot.) This can be done using subformulas similar to the formulas ψ_j and δ_j in φ_{d}.

To handle degenerate L-polies, the formulas κ_i also say that $\gamma_i^{\hat{t}} = Sc(k)$ if either $1 \leq i \leq k$ and $\beta_{i+1}^{\hat{t}} = m$, or $k+2 \leq i \leq 2k+1$ and $\alpha_{i-1}^{\hat{t}} = n$. This prevents Φ_{f} from accepting paths that end outside the dynamic programming table. □

7 On the Complexity of the Pattern Matching Problems

7.1 Upper Bounds

We compare now the efficiency of the approximate pattern matching algorithms that we have considered in the paper. Since Algorithm 2 is nondeterministic, a direct comparison to the deterministic Algorithm 1 is actually not possible. However, for space complexity we can make such a comparison indirectly by using Savitch's theorem, which relates nondeterministic space to deterministic space:

Theorem 5 (Savitch [13]) *A nondeterministic $s(n)$-space bounded Turing machine can be simulated by a deterministic $(s(n))^2$-space-bounded Turing machine, provided $s(n) \geq \log_2 n$.*

Recall that with respect to the RAM model of computation, the space requirement of Algorithm 2 is $O(1)$. However, before applying Theorem 5 we have to find the space complexity with respect to the Turing model of computation. This is straightforward: Storing the values of the variables i, j and *count* requires $O(\log n)$ space, $O(\log m)$ space and $O(\log k)$ space, respectively. Hence the total space requirement is $O(\log n)$ (recall that we are assuming that $k, m \leq n$).

By Theorem 5, Algorithm 2, which includes both of the k differences cases (k is part of the input, and fixed k), can be simulated deterministically in $O((\log n)^2)$ space. This improves the upper bound $O(m)$ on the required space of Algorithm 1 for inputs in which the length of the pattern is relatively large compared to the length of the text, e.g., when $m \geq n^{\epsilon}$ for some $\epsilon > 0$.

In the case of the k mismatches problem, we can get even better space efficiency. To be specific, it is easy to modify Algorithm 3 so that it becomes deterministic: just replace the guess (on line 1) by a repeat-until loop. Clearly, the space requirement of this modified algorithm is the same as that of Algorithm 2, and hence we obtain the space bound $O(\log n)$.

7.2 Lower Bounds

In the upper bound results above the pattern matching problems were considered in the usual complexity theoretic setting in which instances are strings that are directly used as inputs to Turing machines. Here we change to a model theoretic setting; instances will now be (table) models that encode the strings P and S. It is important to note that this can make a difference for the complexity of the problems, since the natural order of the strings is no more available for free in the Turing machine computations; it has to be computed from the input model.

We will show that the intrinsic complexity of the k mismatches problem is sensitive to the way it is formulated in the model theoretic setting. In the present formulation, strings are represented in table models by a successor relation. Another natural way of representing strings is to use linear order as a basic relation instead of successor relation. We denote the corresponding versions of the k mismatches problem by $\mathcal{K}$ and $\mathcal{K}^{\leq}$, respectively. That is, $\mathcal{K}$ is the set of all models $\mathcal{A}$ isomorphic with some table model $\mathcal{A}_{P,S,k}$ such that P has a k mismatches occurrence within S. Similarly, $\mathcal{K}^{\leq}$ is the set of all models $\mathcal{A}^{\leq}$ such that $\mathcal{A} \in \mathcal{K}$, where $\mathcal{A}^{\leq}$ is the model obtained from $\mathcal{A}$ by replacing the successor relation with the corresponding linear order.

Consider the problem $\mathcal{K}$ first. Let $\mathcal{K}_{\mathrm{ord}}$ be the set of models $\langle A, Sc, a, b \rangle$ such that Sc is successor relation on A, and $a < b$ with respect the corresponding linear order $\leq$. Note that the problem $\mathcal{K}_{\mathrm{ord}}$ is the restriction of deterministic reachability 'given $R \subseteq A^2$ and $a, b \in A$, is $(a,b) \in \mathrm{DTC}(R)$' to the case of successor relations. In [4], Etessami proved that $\mathcal{K}_{\mathrm{ord}}$ is LOGSPACE complete with respect to (quantifier free) first order reductions. We show that $\mathcal{K}_{\mathrm{ord}}$ is reducible to the problem $\mathcal{K}$ via a first-order reduction. It follows that $\mathcal{K}$ is also LOGSPACE complete.

The reduction of $\mathcal{K}_{\mathrm{ord}}$ to $\mathcal{K}$ goes as follows. Let $\langle A, R, a, b \rangle$ be an instance of the problem $\mathcal{K}_{\mathrm{ord}}$. We may assume that there are unique R-minimal and R-maximal elements $\min$ and $\max$, and that each element $x \neq \max$ ($x \neq \min$) has a unique R-successor (R-predecessor); otherwise $\langle A, R, a, b \rangle$ is simply rejected. Let c and d be the R-successors of a and b, respectively. Define a new relation $R^{a,b}$ from R by:

$$R^{a,b} = (R \setminus \{(a,c),(b,d)\}) \cup \{(\max, c),(b, \min)\}.$$

Furthermore, define F_{p} and F_{s} to be the constant function with value $\min$: $F_{\mathrm{p}} = F_{\mathrm{s}} = \{(x, \min) \mid x \in A\}$.

We claim now that $\langle A, R, a, b\rangle \in \mathcal{K}_{\rm ord}$ if and only if $\langle A, R^{a,b}, F_{\rm p}, F_{\rm s}, \min\rangle \in \mathcal{K}$. Indeed, if R is a successor relation and $a < b$ with respect to the corresponding order, then $R^{a,b}$ is also a successor relation. Since the pattern P and string S corresponding to $F_{\rm p}$ and $F_{\rm s}$ are identical, P has a k mismatches occurrence within S irrespective of k. On the other hand, if R is not a successor relation, or if $b \leq a$ in the corresponding ordering, then the relation $R^{a,b}$ contains a cycle. Hence, $\langle A, R^{a,b}, F_{\rm p}, F_{\rm s}, \min\rangle$ cannot be isomorphic to any table model.

Note that allowing the insertion and deletion operations E2 and E3 does not change anything in the argument above. Thus $\mathcal{K}_{\rm ord}$ is reducible to the k differences problem (with the successor formulation) as well; moreover, this holds even for the case of fixed k.

Consider then the alternative formulation of the k mismatches problem in terms of table models with linear order. It turns out that there is an intimate connection between the class $\mathcal{K}^{\leq}$ and the counting extension FO(COUNT) of first-order logic. The first observation is that $\mathcal{K}^{\leq}$ is definable in FO(COUNT). Intuitively, this is because, for deciding whether a model $\mathcal{A}$, that encodes pattern P and text S, is in $\mathcal{K}^{\leq}$, it suffices to count the number of mismatches between P and each factor of S; it is straightforward to express such counting in FO(COUNT). A detailed proof will appear in the complete version of the paper. As a consequence, $\mathcal{K}^{\leq}$ is in the complexity class ThC^0. This is in sharp contrast with $\mathcal{K}$ which, being LOGSPACE complete, is not in ThC^0, unless LOGSPACE collapses to ThC^0.

Moreover, on ordered models the converse of the preceding definability result is also true in the following sense. Let $\mathrm{FO}(Q_{\mathcal{K}^{\leq}})$ be the extension of first-order logic by the generalized quantifier $Q_{\mathcal{K}^{\leq}}$ corresponding to the class $\mathcal{K}^{\leq}$ (we refer the reader to [3] for a definition of logics with generalized quantifiers). In the complete version of the paper we will show that, in the presence of a linear order, the counting quantifiers $\exists^i x$ can be simulated by the quantifier $Q_{\mathcal{K}^{\leq}}$. The idea is that $\exists^i x\varphi(x)$ holds if and only if P does not have an $i-1$ mismatches occurrence in S, where P and S are identical, except that $p_x \neq s_x$ for each x satisfying φ. Hence, on ordered models containing also addition and multiplication, the logic $\mathrm{FO}(Q_{\mathcal{K}^{\leq}})$ has exactly the same expressive power as FO(COUNT), and consequently $\mathrm{FO}(Q_{\mathcal{K}^{\leq}})$ captures the complexity class ThC^0.

8 Conclusion

We have considered the problem of expressing approximate pattern matching in strings by using transitive closure logics. Our formalism simulates the calculation along an accepting path of dynamic programming approach.

When k is given as a part of the input, the basic problem of k mismatches can be expressed straightforwardly by FO(DTC). When proceeding to a more challenging problem of k differences, we have also proceeded to more expressive logic, that is FO(TC). It is not certain, however, that this problem could not be expressed by FO(DTC), but proving that such a formula does not exist is at least as difficult as to show that NLOGSPACE $\neq$ LOGSPACE.

Moreover, we have proven that even the k differences problem can be expressed by FO(DTC), if the k is fixed beforehand. In that formalism we harnessed the fact that the minimizing path has to be in a diagonal band whose width is restricted by k, and cover the band with successive symmetrical polygons.

We also derived (with respect to the Turing model of computation) the upper bound $O((\log n)^2)$ for the space complexity of the k differences problem. For the case of the k mismatches problem, this upper bound was improved to $O(\log n)$. On the other hand, we proved that in our model theoretic formalism, the latter cannot be improved, since there is a LOGSPACE complete problem that is reducible to the corresponding class of models. However, this lower bound is sensitive to the way it is formulated: we proved that after a small modification in the formalism, the k mismatches problem becomes expressible in the logic FO(COUNT), whence the problem is already in ThC^0.

Acknowledgments

The authors are indebted to Professor Esko Ukkonen and to Matti Nykänen for their useful remarks on this paper.

References

[1] A. Brazma, I. Jonassen, E. Ukkonen, and J. Vilo. Discovering patterns and subfamilies in biosequences. In *Proc. 4th International Conference on Intelligent Systems for Molecular Biology, ISMB '96*, pages 34–43. AAAI Press, 1996.

[2] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[3] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, Berlin Heidelberg, 1997.

[4] K. Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54:400–411, 1997.

[5] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'94)*, pages 303–312, 1994.

[6] D. Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press, 1997.

[7] N. Immerman. *Descriptive Complexity*. Springer-Verlag, New York, 1999.

[8] G. M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computers and Systems*, 37:63–78, 1988.

[9] K. Lemström and P. Laine. Musical information retrieval using musical parameters. In *Proceedings of the 1998 International Computer Music Conference*, pages 341–348, Ann Arbor, MI, 1998.

[10] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In *Proceedings of Combinatorial Pattern Matching*, pages 1–13, Piscataway, N.J., 1998.

[11] M. Nykänen. *Querying String Databases with Modal Logic*. PhD thesis, University of Helsinki, Department of Computer Science, 1997. Report A-1997-3.

[12] E. G. M. Petrakis. *Image representation, indexing and retrieval based on spatial relationships and properties of objects*. PhD thesis, University of Crete, Department of Computer Science, 1993.

[13] W. J. Savitch. Relationship between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[14] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[15] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[16] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[17] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume A: Algorithms and complexity*, pages 3–66. Elsevier, Amsterdam, 1990.