Approximate Parameterized Matching

Carmit Harel Moshe Lewenstein Dina Sokol

Abstract

Two equal length strings s and s', over alphabets Σ_s and $\Sigma_{s'}$, *parameterize match* if there exists a bijection $\pi : \Sigma_s \rightarrow \Sigma_{s'}$, such that $\pi(s) = s'$, where $\pi(s)$ is the renaming of each character of s via π. *Parameterized matching* is the problem of finding all parameterized matches of a pattern string p in a text t and *approximate parameterized matching* is the problem of finding, at each location, a bijection π that maximizes the number of characters that are mapped from p to the appropriate $|p|$-length substring of t.

Parameterized matching was introduced as a model for software duplication detection in software maintenance systems and also has applications in image processing and computational biology. For example, approximate parameterized matching models image searching with variable color maps in the presence of errors.

We consider the problem for which an error threshold, k, is given and the goal is to find all locations in t for which there exists a bijection π which maps p into the appropriate $|p|$-length substring of t with at most k mismatched mapped-elements.

We show that (1) the approximate parameterized matching, when $|p|=|t|$, is equivalent to the maximum matching problem on graphs, implying that (2) maximum matching is reducible to the approximate parameterized matching with threshold k, up till an $O(\log |t|)$ factor (this can be achieved by reducing approximate parameterized matching to the problem by using a binary search on the k's). Given the best known maximum matching algorithms an $O(m^{1.5})$, where $m = |p| = |t|$, is implied for approximate parameterized matching. We show that (3) for the k threshold problem we can do this in $O(m + k^{1.5})$.

Our main result (4) is an $O(nk^{1.5} + mk \log m)$ time algorithm where $m = |p|$ and $n = |t|$.

1 Introduction

In the traditional pattern matching model [10, 18], one seeks exact occurrences of a given pattern p in a text t, i.e. text locations where every text symbol is *equal* to its corresponding pattern symbol. For two equal length strings s and s' we say that s is a *parameterized match*, *p-match* for short, of s' if there exists a bijection π from the alphabet of s to the alphabet of s' such that every symbol of s' is equal to *the image under* π of the corresponding symbol of s. In the *parameterized matching* problem, introduced by Baker [6, 8], one seeks text locations for which the pattern p p-matches the substring of length $|p|$ beginning at that location.

Baker introduced parameterized matching for applications that arise in software tools for analyzing source code. Specifically, the application is to identify duplicate code in large software systems for reuse. Here it is desirable to find not only exact matches between program fragments but also parameterized matches, namely, where the two program fragments are equal but possibly use interchangeable identifiers (representing variable, constant or function names).

When program fragments that repeat are discovered, other files of the program are searched to find repetitions of the program fragment in other locations. (It is often the case that repeating fragments have many appearances.) This exactly defines the *parameterized matching problem*, or *p-match problem* for short. It turns out that the parameterized matching problem arises in other applications such as in image processing and computational biology, see [2].

In [6, 8] an optimal, linear time algorithm was given for p-matching. However, it was assumed that the alphabet was of constant size. Amir et.al. [4] presented tight bounds for p-matching in the presence of an unbounded size alphabet.

In [7] a novel method was presented for parameterized matching by constructing *parameterized suffix trees*, which also allows for online p-matching. The parameterized suffix trees are constructed by converting the pattern string into a *predecessor string*. A *predecessor string* of a string s has at each location i the distance between i and the location containing the previous appearance of the symbol. The first appearance of each symbol replaced with a 0. For example, the predecessor string of $aabbaba$ is $0, 1, 0, 1, 3, 2, 2$. A simple and well-known fact is that:

Observation 1 *s and s' p-match iff they have the same predecessor string.*

The parameterized suffix tree is constructed in a manner that every path corresponds to a suffix, in the standard sense, but is labeled with the predecessor string of the appropriate suffix. Branching in the parameterized suffix tree occurs according to the labels of the predecessor strings. The parameterized suffix tree was further explored by Kosaraju [19] and faster constructions were given by Cole and Hariharan [11].

One of the interesting problems in web searching is searching for color images, see [1, 16, 25]. The simplest possible case is searching for an icon in a screen, a task that the Human-Computer Interaction Lab at the University of Maryland was confronted with. If the colors are fixed, this is exact 2-dimensional pattern matching [3]. However, if the color maps in pattern and text differ, the exact matching algorithm would not find the pattern. Parameterized 2-dimensional search is precisely what is needed. A nearly optimal algorithm to solve the 2-dimensional parameterized matching problem was given in [2].

In reality, when searching for an image, one needs to take into account the occurrence of errors that distort the image. Errors occur in various forms, depending upon the application. Several distance metrics have been defined to account for such errors. Two of the most classical distance metrics are the Hamming distance and the edit distance [22]. The *Hamming distance* between two equal-length strings is the number of mismatching characters when the strings are aligned. The edit distance is the minimal number of character replacements, insertions, and deletions needed to convert one string into another [22].

In [9] the parameterized match problem was considered in conjunction with the edit distance. Here the definition of edit distance was slightly modified so that the edit operations are defined to be insertion, deletion and parameterized replacements, i.e. the replacement of a substring with a string that p-matches it. An algorithm for finding the "p-edit distance" of two strings was devised whose efficiency is close to the efficiency of the algorithms for computing the classical edit distance. Also an algorithm was suggested for the decision variant of the problem where a parameter k is given and it is necessary to decide whether the strings are within (p-edit) distance k.

However, it turns out that the operation of parameterized replacement relaxes the problem to an easier problem. The reason that the problem becomes easier is that two substrings that participate in two parameterized replacements are independent of each other (in the parameterized sense).

A more rigid, but more realistic, definition for the Hamming distance variant was given in [5]. For a pair of equal length strings s and s' and a bijection π defined on the alphabet of s, the π-mismatch is the Hamming distance between the image under π of s and s'. The minimal π-mismatch over all bijections π is the approximate parameterized match. The problem considered in [5] is to find for each location i of a text t the approximate parameterized match of a pattern p with the substring beginning at location i. In [5] the problem was defined and linear time algorithms were given for the case were the pattern is binary or the text is binary. However, this solution does not carry over to larger alphabets.

Unfortunately, under this definition the methods for classical string matching with errors for Hamming distance, e.g. [15, 21], also known as pattern matching with mismatches, seem to fail. Following is an outline of the method [15] for pattern matching with mismatches.

The pattern is compared separately to each suffix of the text, beginning at locations $1 <= i <= n$. Using a suffix tree of the text and precomputed longest common ancestor (LCA) information (which can be computed once in linear time, see [17, 24]) one can find the longest common prefix of the pattern and the corresponding suffix (in constant time). There must be a mismatch immediately afterwards. The algorithm jumps over the mismatch and repeats the process taking into consideration the offsets of the pattern and suffix.

When attempting to apply this technique to a parameterized suffix tree, it fails. To illustrate this, consider the first matching substring (up until the first error) and the next matching substring (after the error). Both of these substrings p-match the substring of the text that they are aligned with. However, it is possible that combined they do not form a p-match. See the example below. In the example *abab* p-matches *cdcd* followed by a mismatch and subsequently followed by *abaa* p-matching $efee$. However, different π's are require for the local p-matches. This also emphasizes why the definition of [9] is a simplification.

$$\begin{array}{rl} p = & \quad a\ b\ a\ b\ a\ a\ b\ a\ a\ ... \\ t = & ...\ c\ d\ c\ d\ d\ e\ f\ e\ e\ ... \end{array}$$

In this paper we consider the problem of *parameterized matching with k mismatches*. The parameterized matching problem with k mismatches seeks all *p-matches* of a pattern p in a text t, with at most k mismatches.

For the case where $|p|=|t| = m$, which we call the string comparison problem we show that this problem is in a sense equivalent to the maximum matching problem on graphs. An $O(m^{1.5})$ algorithm for the problem follows, by using maximum matching algorithms. A matching "lower bound" follows, in the sense that an $o(m^{1.5})$ algorithm implies better algorithms for maximum matching. For the string comparison problem with threshold k, we show an $O(m + k^{1.5})$ time algorithm and a "lower bound" of $\Omega(m + \frac{k^{1.5}}{\log m})$.

The main result of the paper is an algorithm that solves the parameterized matching problem with k mismatches problem, given a pattern of length m and a text of length n, in $O(nk^{1.5}+mk \log m)$ time.

This immediately yields a 2-dimensional algorithm of time complexity $O(n^2mk^{1.5} + m^2k\log m)$, where $|p| = m^2$ and $|t| = n^2$.

Roadmap: In section 2 we give preliminaries and definitions of the problem. In section 3 we present an algorithm that compares two equal length strings for a parameterized match with k mismatches. Also a reduction from maximum matching to the string comparison problem is given. The algorithmic techniques introduced in this section are used in the following section, section 4, where we present the algorithm for the general problem of parameterized string matching with k mismatches. Finally, in section 6, we show an extension of our algorithm for images.

2 Preliminaries and Definitions

Given a *string* $s = s_1s_2...s_n$ of *length* $|s| = n$ over an alphabet Σ_s, and a bijection π from Σ_s to some other alphabet Σ'_s, the *image* of s under π is the string $s' = \pi(s) = \pi(s_1) \cdot ... \cdot \pi(s_n)$, that is obtained by applying π to all characters of s.

Given two equal-length strings w and u over alphabets Σ_w and Σ_u and a bijection π from Σ_w to Σ_u the π-mismatch between w and u is $Ham(\pi(w), u)$, where $Ham(s,t)$ is the Hamming distance between s and t. We say that w *parameterized k-matches* u if there exists a π such that the π-mismatch $\leq k$. The *approximate parameterized match* between w and u is the minimum π-mismatch (over all bijections π).

For given input text x, $|x| = n$, and pattern y, $|y| = m$, the problem of *approximate parameterized matching* for y in x consists of computing the *approximate parameterized match* between y and every (consecutive) substring of length m of x. Hence, approximate parameterized searching requires computing the π yielding minimum π-mismatch for y at each position of x. Of course, the best π is not necessarily the same at every position. The problem of *parameterized matching with k mismatches* consists of computing for each location i of x whether y parameterized k-matches the (consecutive) substring of length m beginning at location i. Sometimes π itself is also desired (however, here any π with π-mismatch of no more than k will be satisfactory).

3 String Comparison Problem

We begin by evaluating two equal-length strings for a parameterized k-match as follows:

- **Input:** Two strings, $s = s_1, s_2, ..., s_m$ and $s' = s'_1, s'_2, ..., s'_m$, and an integer k.
- **Output:** True, if there exists π such that the π-mismatch of s and t is no more than k.

In standard string comparison, we simply compare the characters and count the number of mismatches in $O(m)$ time. In approximate parameterized matching the naive method of solving the problem is to check the π-mismatch for every possible π. However, this takes exponential time.

One way to solve the problem is to reduce the problem to a maximal bipartite weighted matching in a graph. We construct a bipartite graph $B = (U \cup V, E)$ in the following way. $U = \Sigma_s$ and

$V = \Sigma_{s'}$. There is an edge between $a \in \Sigma_s$ and $b \in \Sigma_{s'}$ iff there is at least one a in s that is aligned with a b in s'. The weight of this edge is the number of a's in s that are aligned to b's in s'. It is easy to see that:

Observation 2 *A maximal weighted matching in B corresponds to a minimal π-mismatch, where π is defined by the edges of the matching.*

The problem of maximum bipartite matching has been widely studied and efficient algorithms have been devised, e.g. [12, 13, 14, 23]. For integer weights, where the largest weight is bounded by $|V|$, the fastest algorithm runs in time $O(E\sqrt{V})$ [23]. This solution yields an $O(m^{1.5})$ time algorithm for the problem of parameterized string comparison with mismatches. The advantage of the algorithm is that it views the whole string at once and efficiently finds the best function. In general, for the parameterized pattern matching problem with inputs t of length n, and p of length m, the bipartite matching solution yields an $O(nm^{1.5})$ time algorithm.

3.1 Reducing Maximum Matching to the String Comparison Problem

Lemma 1 *Let $B = (U \cup V, E)$ be a bipartite graph. If the string comparison problem can be solved in $O(f(m))$ time then a maximum matching in B can be found in $O(f(|E|)$ time.*

Proof: Take an enumeration of the nodes $\{u_1, u_2, ..., u_{|U|}\}$ and $\{v_1, v_2, ..., v_{|V|}\}$ and an enumeration of the edges $\{e_1, ..., e_{|}E|\}$. Create two strings s_U and s_V of length $|E|$, where for edge $e_i = \{u_j, v_l\}$ we set the ith location of s_U to be u_j and the ith location of s_V to be v_j.

Obviously a bijection from the "alphabet" of s_U to the "alphabet" of s_V corresponds to a matching in B. Hence, the lemma follows. $\square$

Since it is always true that $|E| < |V|^2$ it follows that $|E|^{1/4} < \sqrt{|V|}$. Hence, it would be surprising to solve the string comparison problem in $o(m^{1.25})$. In fact, even for graphs with a linear number of edges the best known algorithm is $O(E\sqrt{V})$ and hence an algorithm for the string comparison problem in $o(m^{1.5})$ would have implications on the maximum matching problem as well.

Note that for the string comparison problem one can reduce the approximate parameterized matching version to the parameterized matching with k mismatches version. This is done by binary searching on k to find the optimal solution for the string comparison problem in the approximate parameterized matching version. Hence an algorithm for detecting whether the string comparison problem has a bijection with fewer than k mismatches with time $o(\frac{k^{1.5}}{\log k} + m)$ would imply a faster maximum matching algorithm.

3.2 Mismatch Pairs and Parameterized Properties

Obviously, one may use the solution of approximate parameterized string matching to solve the problem of parameterized matching with k mismatches. However, it is desirable to construct an algorithm with running time dependant on k rather than m. The bipartite matching method does not seem to give insight into how to achieve this goal. In order to give an algorithm dependent on

k we introduce a new method for detecting whether two equal-length strings have a parameterized k-match. The ideas used in this new comparison algorithm will be used in the following section to yield an efficient solution for the problem of parameterized matching with k mismatches.

When faced with the problem of p-matching with mismatches, the initial difficulty is to decide, for a given location, whether it is a match or a mismatch. Simple comparisons do not suffice, since any symbol can match (or mismatch) every other symbol. In the bipartite solution, this difficulty is overcome by viewing the entire string at once. A bijection is found, over all characters, excluding at most k locations. Thus, it is obvious that the locations that are excluded from the matching are "mismatched" locations. For our purposes, we would like to be able to decide locally, *i.e. before* π *is ascertained*, whether a given location is "good" or "bad."

Definition 1 (mismatch pair) *Let s and s' be two equal-length strings then a* mismatch pair *between s and s' is a pair of locations (i,j) such that one of the following holds,*

$$1.\ s_i = s_j \text{ and } s'_i \neq s'_j \qquad 2.\ s_i \neq s_j \text{ and } s'_i = s'_j.$$

It immediately follows from the definition that:

Lemma 2 *Given two equal-length strings, s and s', if (i, j) is a mismatch pair between s and s', then for every bijection $\Pi : \Sigma_s \rightarrow \Sigma_{s'}$ either location i or location j, or both locations i and j are mismatches, i.e. $\pi(s_i) \neq s'_i$ or $\pi(s_j) \neq s'_j$ or both are unequal.*

Conversely,

Lemma 3 *Let s and s' be two equal-length strings and let $S \subset \{1, ..., m\}$ be a set of locations which does not contain any mismatch pair. Then there exists a bijection $\pi : \Sigma_s \rightarrow \Sigma_{s'}$ that is parameterized on S, i.e. for every $i \in S, \pi(s_i) = s'_i$.*

Proof: S does not contain any mismatch pair. Hence, for any two locations i and j in S if $s_i{=}s_j$ then $s'_i{=}s'_j$, and if $s_i \neq s_j$ then $s'_i \neq s'_j$. Hence there is a bijection $\pi : \Sigma_s \rightarrow \Sigma_{s'}$ that is a parameterized match on the set of locations S. □

The idea of our algorithm is to count the mismatch pairs between s and s'. Since each pair contributes *at least* one error in the p-match between s and s' it follows immediately from Lemma 2 that if there are more than k mismatch pairs then s does not parameterize k-match s'. We claim that if there are fewer than $k/2 + 1$ mismatch pairs then s parameterize k-matches s'.

Definition 2 *Given two equal-length strings, s and s', a collection L of mismatch pairs is said to be a* maximal disjoint collection *if (1) all mismatch pairs in L are disjoint, i.e. do not share a common location, and (2) there is no mismatch pair that can be added to L without violating disjointness.*

Corollary 1 *Let s and s' be two strings, and let L be a maximal disjoint collection of mismatch pairs of s and s'. If $|L| > k$, then for every $\Pi : \Sigma_s \rightarrow \Sigma'_s$, the Π-mismatch is greater than k. If $|L| \leq k/2$ then there exists $\Pi : \Sigma_s \rightarrow \Sigma'_s$ such that the Π-mismatch counter is less than or equal to k.*

Proof: Combining Lemma 2 and Lemma 3 yields the proof. □

3.3 String Comparison Algorithm

The method uses Corollary 1. First the mismatch pairs need to be found. In fact, by Corollary 1 only $k+1$ of them need to be found since $k+1$ mismatch pairs implies that there is no parameterized k-match. After the mismatch pairs are found if the number of mismatch pairs mp is less than $k/2$ or more than k we can announce a mismatch or match immediately according to Corollary 1. The difficult case to detect is whether there indeed is a parameterized k-match for $k/2 < mp \leq k$. This is done with the bipartite matching algorithm. However, here the bipartite graph needs to be constructed somewhat differently. While the case where $mp \leq k/2$ implies an immediate match, for simplicity of presentation we will not differentiate between the case of $mp \leq k/2$ and $k/2 < mp \leq k$. Thus from here on we will only bother with the cases $mp \leq k$ and $mp > k$.

3.3.1 Find Mismatch Pairs

In order to find the mismatch pairs of equal-length strings s and s' we use a simple stack scheme, one stack for each character in the alphabet. First mismatch pairs are searched for according to the first rule of definition 1, i.e. $s_i = s_j$ but $s'_i \neq s'_j$. Then within the leftover locations mismatch pairs are seeked that satisfy the second rule of definition 1, i.e. $s_i \neq s_j$ but $s'_i = s'_j$.

Algorithm: Find mismatch pairs(s,s')

1. Construct a stack S_σ for each $\sigma \in \Sigma_s$.
2. For $i = 1$ to m do
   if empty(S_{s_i}) then push($S_{s_i}, < s'_i, i >$)
   else if top(S_{s_i}) $=< s'_i, * >$ then push($S_{s_i}, < s'_i, i >$)
   else $< s'^*, j > \leftarrow$ pop(S_{s_i}); add (i, j) to the mismatch pairs
3. Set $\bar{L}$ to be all locations of $\{1, ..., m\}$ that do not appear in the mismatch pairs.
4. Reverse the roles of s and s' and do step 2 while scanning items only from $\bar{L}$
   (i.e. here step 2 begins - "For $j \in \bar{L}$ do")

Time Complexity: The algorithm that finds all mismatch pairs between two strings of length m runs in $O(m)$ time. Each character in the string is pushed/popped onto at most one stack.

3.3.2 Verification

The verification is performed only when the number of mismatch pairs that were found, mp, satisfies $mp \leq k$. Verification consists of a procedure that finds the minimal π-mismatch over all bijections π. The technique used is similar to the bipartite matching algorithm discussed in the beginning of the section.

Let $\hat{L} \subset \{1, ..., m\}$ be the locations that appear in a mismatch pair. We say that a symbol $a \in \Sigma_s$ is *mismatched* if there is a location $i \in \hat{L}$ such that $s_i = a$. Likewise, $b \in \Sigma_{s'}$ is *mismatched* if there is a location $i \in \hat{L}$ such that $s'_i = b$. A symbol $a \in \Sigma_s$, or $b \in \Sigma_{s'}$, is *free* if it is not mismatched.

Construct a bipartite graph $B = (U \cup V, E)$ defined as follows. U contains all mismatched symbols $a \in \Sigma_s$ and V contains all mismatched symbols $b \in \Sigma_{s'}$. Moreover, U contains all free symbols $a \in \Sigma_s$ for which there is a location $i \in \{1, ..., m\} - \hat{L}$ such that $s_i = a$ and s'_i is a mismatched symbol. Likewise, V contains all free symbols $b \in \Sigma_{s'}$ for which there is a location $i \in \{1, ..., m\} - \hat{L}$ such that $s'_i = b$ and s_i is a mismatched symbol. The edges, as before, have weights that correspond to the number of locations where they are aligned with each other.

Lemma 4 *The bipartite graph B is of size $O(k)$. Moreover, given the mismatch pairs it can be constructed in $O(k)$ time.*

Proof: This lemma follows by observing that for any $a \in \Sigma_s$ all appearances of a in locations of $\{1, ..., m\} - \hat{L}$ are aligned with a given symbol $b \in \Sigma_{s'}$ by definition of mismatch pairs. Symmetrically, the same is true for any $b \in \Sigma_{s'}$. On the other hand, $\hat{L}$ is of size $O(k)$. $\square$

While the lemma states that the bipartite graph is of size $O(k)$ it still may be the case that the edge weights may be substantially larger. However, if there exists an edge $e = (a, b)$ with weight $> k$ then it must be that $\pi(a) = b$ for otherwise we immediately have $> k$ mismatches. Thus, before computing the maximum matching, we drop every edge with weight $> k$, along with their vertices. What remains is a bipartite graph with edge weights between 1 and k.

Theorem 1 *Given two equal-length strings s and s', with $mp \leq k$ mismatch pairs. It is possible to verify whether there is a parameterized k-match between s and s' in $O(k^{1.5})$ time.*

Proof: Since the size of the bipartite graph B is $O(k)$, by Lemma 4, it follows that the maximum weighted bipartite matching can be solved in $O(k^{1.5})$ time algorithm [23].

We need to show that its solution solves our problem. The matching M clearly yields a one-to-one function $\pi : \Sigma_s \to \Sigma'_s$, since at most one edge is chosen per character in s and s'. Since the matching is maximal, it is the optimal function, and every character not included in the matching is an error. If the sum of the edge weights in $E - M$ is more than k then there is no parameterized k-match between s and s'. $\square$

Time Complexity: Given two equal-length strings s and s', it is possible to determine whether s parameterized k-matches s' in $O(m + k^{1.5})$ time, $O(m)$ to find the mismatch pairs and $O(k^{1.5})$ to check these pairs with the appropriate bipartite matching.

4 The Algorithm

We are now ready to introduce our algorithm for the problem of parameterized pattern matching with k mismatches:

- **Input:** Two strings, $t = t_1, t_2, ..., t_n$ and $p = p_1, p_2, ..., p_m$, and an integer k.
- **Output:** All locations i in t where p parameterized k-matches $t_i, \ldots, t_{i+m-1}$.

Our algorithm has two phases, the *pattern preprocessing phase* and the *text scanning phase*. In this section we present the text scanning phase and will assume that the pattern preprocessing phase is given. In the following section we describe an efficient method to preprocess the pattern for the needs of the text scanning phase.

Definition 3 *Let s and s' be two-equal length strings. Let L be a collection of disjoint mismatch pairs between s and s'. If L is maximal and $|L| \leq k$ then L is said to be a k-*good witness *for s and s'. If $|L| > k$ then L is said to be a k-*bad witness *for s and s'.*

The text scanning phase has two stages the (a) *filter stage* and the (b) *verification stage*. In the filter stage for each text location i we find either a k-good witness or a k-bad witness for p and $t_i \ldots t_{i+m-1}$. Obviously, by Corollary 1, if there is a k-bad witness then p cannot parameterize k-match $t_i \ldots t_{i+m-1}$. Hence, after the filter stage, it remains to verify for those locations i which have a k-good witness whether p parameterize k-matches $t_i \ldots t_{i+m-1}$. The verification stage is identical to the verification procedure in Section 3.3.2 and hence we will not dwell on this stage.

4.1 The Filter Stage

The underlying idea of the filter stage is similar to [20]. Like the KMP algorithm [18] one location after another is evaluated utilizing the knowledge accumulated at the previous locations combined with the information gathered in the pattern preprocessing stage. The information of the pattern preprocessing stage is:

Output of pattern preprocessing: For each location i of p, a maximal disjoint collection of mismatch pairs L for some pattern prefix $p_1, \ldots, p_{j-i+1}$ and $p_i, \ldots, p_j$ such that either $|L| = 3k+3$ or $j = m$ and $|L| \leq 3k+3$.

As the first step of the algorithm we consider the first text location, i.e. when text $t_1 \ldots t_m$ is aligned with $p_1 \ldots p_m$. Using the method in Section 3.3.1 we can find all the mismatch pairs in $O(m)$ time, and hence find a k-good or k-bad witness for the first text location. When evaluating subsequent locations i we maintain the following invariant: *For all locations $l < i$ we have either a k-good or k-bad witness for location l.*

It is important to observe that, when evaluating location i of the text, if we discover a maximal disjoint collection of mismatch pairs of size $k+1$, i.e. a k-bad witness, between a prefix $p_1, ..., p_j$ of the pattern and $t_i, \ldots, t_{i+j-1}$ we will stop our search, since this immediately implies that $p_1, \ldots, p_m$

and $t_i, \ldots, t_{i+m-1}$ have a k-bad witness. We say that $i+j$ is a *stop location* for i. If we have a k-good witness at location i then $i+m$ is its *stop location.* When evaluating location i of the text, each of the previous locations has a stop location. The location ℓ which has a maximal stop location, over all locations $l < i$, is called a *maximal stopper for* i.

The reason for working with the maximal stopper for i is that the text beyond the maximal stopper has not yet been scanned. If we can show that the time to compute a k-bad witness or find a maximal disjoint collection of mismatch pairs for location i up until the maximal stopper, i.e. for $p_1, \ldots, p_{\ell'-i+1}$ with $t_i, \ldots, t_{\ell'}$, is $O(k)$ then we will spend overall $O(n+nk)$ time for computing, $O(nk)$ to compute up until the maximal stopper for each location and $O(n)$ for the scanning and updating the current maximal collection of mismatch pairs.

4.1.1 Up to the Maximal Stopper

The situation now is that we are evaluating a location i of t. Let ℓ be the maximal stopper for i. We utilize two pieces of precomputed information; (1) the pattern preprocessing for location $i-\ell+1$ of p and (2) the k-good witness or k-bad witness (of size $k+1$) for location ℓ of t. Let ℓ' be the stop location of ℓ. We would like to evaluate the overlap of $p_1, \ldots, p_{\ell'-i+1}$ with $t_i, \ldots, t_{\ell'}$ and to find a maximal disjoint collection L of mismatch pairs on this overlap, or, if it exists, a maximal disjoint collection L of mismatch pairs of size $k+1$ on a prefix of the overlap.

There are two cases. One possibility is that the pattern preprocessing returns $3k+3$ mismatch pairs for a prefix $p_1, \ldots, p_j$ and $p_{\ell-i+1}, \ldots, p_{\ell-i+j+1}$ where $j \leq \ell' - i + 1$. Yet, there are $\leq k+1$ mismatch pairs between $p_1, \ldots, p_{\ell'-\ell+1}$ and $t_\ell, \ldots, t_{\ell'}$.

Lemma 5 *Let s, s' and s'' be three equal-length strings such that there is a maximal collection of mismatch pairs $L_{s,s'}$ between s and s' of size $\leq M$ and a maximal collection of mismatch pairs $L_{s',s''}$ between s' and s'' of size $\geq 3M$. Then there must be a maximal collection of mismatch pairs $L_{s,s''}$ between s and s'' of size $\geq M$.*

Proof: Since each mismatch pair is composed of 2 locations, there must be at least M mismatch pairs between s' and s'' which do not participate in mismatch pairs between s and s'. It follows from the definition of mismatch pairs that these M mismatch pairs are also mismatch pairs between s and s''. □

Set M to be $k+1$ and one can use the Lemma for the case at hand, namely, there must be at least $k+1$ mismatch pairs between $p_1, \ldots, p_{\ell'-i+1}$ and $t_i, \ldots, t_{\ell'}$, which defines a k-bad witness.

The second case is where the pattern preprocessing returns fewer than $3k+3$ mismatch pairs or it returns $3k+3$ mismatch pairs but it does so for a prefix $p_1, \ldots, p_j$ and $p_{\ell-i+1}, \ldots, p_{\ell-i+j+1}$ where $j > \ell' - i + 1$. However, we still have an upper bound of $k+1$ mismatch pairs between $p_1, \ldots, p_{\ell'-\ell+1}$ and $t_\ell, \ldots, t_{\ell'}$. So, we can utilize the fact that:

Lemma 6 *Given three strings, s, s', s'' such that there are maximal disjoint collections of mismatch pairs, $L_{s,s'}$ and $L_{s',s''}$ of sizes $O(k)$. In $O(k)$ time one can find a k-good or k-bad witness for s and s''.*

Proof: Let $\bar{L}_{s,s'}$ be the indices that are not in any mismatch pair between s and s'. By Lemma 3 there exists a bijection $\pi : \Sigma_s \to \Sigma_{s'}$ such that π defines a parameterized match on $\bar{L}_{s,s'}$. Likewise, if $\bar{L}_{s',s''}$ are the indices that are not in any mismatch pair between s' and s'', then there is a bijection $\pi' : \Sigma_{s'} \to \Sigma_{s''}$ that defines a parameterized match on $\bar{L}_{s',s''}$.

The set of indices that participates in no mismatch pair in either of the pairs is $\bar{L}_{s,s'} \cap \bar{L}_{s',s''}$. On this set $\pi' \cdot \pi()$ defines a parameterized match. However, the number of locations participating in mismatch pairs is bounded by k. Using linked lists for each character one can now construct the, at most $O(k)$ mismatch pairs in $O(k)$ time. □

4.1.2 Putting it Together

The filter stage takes $O(nk)$ time and the verification stage takes $O(nk^{1.5})$. Hence,

Theorem 2 *Given the preprocessing stage we can announce for each location i of t whether p parameterized k-matches $t_i, ..., t_{i+m-1}$ in $O(nk^{1.5})$ time.*

5 Pattern Preprocessing

In this section we solve the pattern preprocessing necessary for the general algorithm.

- **Input:** A pattern $p = p_1, \ldots, p_m$ and an integer k.
- **Output:** For each location i, $1 \leq i \leq m$ a maximal disjoint collection of mismatch pairs L for the minimal pattern prefix $p_1, \ldots, p_{j-i+1}$ and $p_i, \ldots, p_j$ such that either $|L| = 3k + 3$ or $j = m$ and $|L| \leq 3k + 3$.

The naive method takes $O(m^2)$, by applying the method from Section 3.3.1. However, this does not exploit any previous information for a new iteration. Since every alignment combines two previous alignments, we can get a better result. Assume, w.l.o.g., that m is a power of 3. We divide the set of alignments into $\log_3 m + 1$ sequences as follows;

$R_1 = [2, 3], R_3 = [4, 5, 6, 7, 8, 9], ..., R_i = [3^{i-1} + 1, ..., 3^i], ..., R_{\log_3 m} = [3^{\log_3 m - 1} + 1, ..., 3^{\log_3 m}]$ $1 \leq i \leq \log_3 m$.

At each step, we compute the desired mismatches for each set of alignments R_i. Step i uses the information computed in steps $1...i-1$. We further divide each set into two halves, and compute the first half followed by the second half. This is possible since each R_i contains an even number of elements, as $3^i - 3^{i-1} = 2 * 3^{i-1}$. We split each sequence R_i into two equal length sequences $R_i^1[3^{i-1} + 1, ..., 2 * 3^{i-1}]$ and $R_i^2 = [2 * 3^{i-1} + 1, \ldots, 3^i]$. For each of the new sequences we have that:

Lemma 7 *If $r_1, r_2 \in R_i^1$ (the first half of set R_i) or $r_1, r_2 \in R_i^2$ (the second half of R_i) such that $r_1 < r_2$ then $r_2 - r_1 \in R_j$ for $j < i$.*

Proof: We assume that $r_1, r_2 \in R_i^1$ (the proof for the other case is symmetric). Obviously $r_2 \leq 2 * 3^{i-1}$ and $r_1 \geq 3^{i-1} + 1$, then $r_2 - r_1 \leq 2 * 3^{i-1} - 3^{i-1} - 1 = 3^{i-1} - 1$. □

This gives a handle on how to compute our desired output. Denote $f_i = min\{j \in R_i^1\}$ and $m_i = min\{j \in R_i^2\}$ the representatives of their sequences. We compute the following two stages for each group R_i, in order $R_1, \ldots, R_{\log_3 m}$,

1. Compute a maximal disjoint collection of mismatch pairs L for some pattern prefix $p_1, \ldots, p_{j+1}$ and $p_{f_i}, \ldots, p_{f_i+j}$ such that $|L| = (3k+3)3^{\log_3 m - i}$ or $f_i + j = m$ and $|L| \leq (3k+3)3^{\log_3 m - i}$. Do the same with $p_{m_i}, \ldots, p_{m_i+j}$.
2. Now for each $j \in R_i^1$ apply the algorithm for the text described in the previous section on the pattern p, the pattern shifted by f_i (R_i^1's representative) and the pattern shifted by j. Do the same with m_i for R_i^2.

The central idea and importance behind the choice of the number of mismatch pairs that we seek is to satisfy Lemma 5. It can be verified that indeed our choice of sizes always satisfies that there are 3 times as many mismatch pairs as in the previous iteration. Therefore,

Theorem 3 *Given a pattern p of length m. It is possible to precompute its $3k+3$ mismatch pairs at each alignment in $O(km \log_3 m)$ time.*

The time complexity for each group R_i with $O(3^{i-1})$ members is $3^{\log_3 m - i + 1}(3k+3)$ for finding the mismatches, multiply by $O(3^{i-1})$ members, which is $O(mk)$ for a group, plus $O(m)$ for the first and the middle elements. All together it is $O(km \log_3 m)$. □

Corollary 2 *Given a pattern p and text t, we can solve the parameterized matching with k mismatches problem in $O(nk^{1.5} + km \log m)$ time.*

6 2-d Parameterized Matching with k Mismatches

The applications for approximate parameterized matching, such as the image search with variable color maps and errors, is 2-dimensional. So, in this section we present an extension of our algorithm to 2-dimensional approximate parameterized matching.

- **Input:** Two images, text $T[1...n][1...n]$, and pattern $P[1...m][1...m]$, and an integer k.
- **Output:** all locations (i,j) in T such that P parameterized k-matches the $m \times m$ image whose upper left corner is $T[i,j]$.

We begin with the assumption that the text has exactly m rows, and then multiply the resulting time complexity by n/m. Both the text and the pattern are linearized column-by-column. Our string matching algorithm of section 4 can be applied to the linear pattern and text. Care must be taken that only those locations whose index is divisible by m will be considered. This can clearly be accomplished in $O(n)$ time. Thus, the time complexity for each m rows of the text is $O(nmk^{1.5})$.

Theorem 4 *Given an $m \times m$ image P and an $n \times n$ image T, we can solve the parameterized matching with k mismatches problem in $O(n^2mk^{1.5} + m^2k \log m)$ time.*

References

[1] K. W. Church A. Amir and E. Dar. Separable attributes: a technique for solving the submatrices character count problem. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 400–401, 2002.

[2] A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications and a lower bound. In *Proc. of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 929–942, 2003.

[3] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

[4] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.

[5] A. Apostolico, P. Erdős, and M. Lewenstein. Parameterized matching with mismatches. *manuscript.*

[6] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation*, pages 71–80, 1993.

[7] B. S. Baker. Parameterized string pattern matching. *J. Comput. Systems Sci.*, 52, 1996.

[8] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Computing*, 26, 1997.

[9] B. S. Baker. Parameterized diff. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 854–855, 1999.

[10] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[11] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. In *Proc. 32nd ACM STOC*, pages 407–415, 2000.

[12] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimizations algorithms. *J. of the ACM*, 34(3):596–615, 1987.

[13] H. N. Gabow. Scaling algorithms for network problems. *J. of Computer and System Sciences*, 31:148–168, 1985.

[14] H. N. Gabow and R.E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. on Computing*, 18:1013–1036, 1989.

[15] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[16] B.M. Mehtre G.P. Babu and M.S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4), 1995.

[17] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[18] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[19] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. *Proc. 36th IEEE FOCS*, pages 631–637, 1995.

[20] G.M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching. *Proc. 18th ACM Symposium on Theory of Computing*, pages 220–230, 1986.

[21] G.M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37, 1988.

[22] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[23] W.K. Sung M.Y. Kao, T.W. Lam and H.F. Ting. A decomposition theorem for maximum weight bipartite matching. *SIAM J. on Computing*, 31(1):18–26, 2001.

[24] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17:1253–1262, 1988.

[25] M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.