

# Approximate Parameterized Matching\*

Carmit Hazay<sup>†</sup> Moshe Lewenstein<sup>\*‡</sup> Dina Sokol<sup>§</sup>

## Abstract

Two equal length strings  $s$  and  $s'$ , over alphabets  $\Sigma_s$  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  $\pi$ . *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  $\pi$  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  $\pi$  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 [11, 19], 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* of  $s'$  if there exists a bijection  $\pi$  from the alphabet of  $s$  to the alphabet of  $s'$  such that every symbol of  $s'$  is equal to *the image under*  $\pi$  of the corresponding symbol of  $s$ . In the *parameterized matching* problem, introduced by Baker [7, 9], one seeks all text locations for which the pattern  $p$  parameterize matches the substring of length  $|p|$  beginning at that location.

---

\*Part of this work appeared in a preliminary version [18].

<sup>†</sup>Bar-Ilan University, Ramat Gan 52900, Israel, {harelc,moshe}@cs.biu.ac.il

<sup>‡</sup>The second author was partially supported by an IBM faculty award grant.

<sup>§</sup>Brooklyn College of the City University of N.Y., sokol@sci.brooklyn.cuny.edu; This work was done in part while the author was a postdoctoral student at Bar-Ilan University; partially supported by the Israel Science Foundation Grant 282/01.

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 application is exactly captured by the notion of the *parameterized matching problem*. It turns out that the parameterized matching problem arises in other applications such as in image processing and computational biology, see [2].

In [7, 9] an optimal, linear time algorithm was given for parameterized matching. However, it was assumed that the alphabet was of constant size. Amir et.al. [4] presented tight bounds for parameterized matching in the presence of an unbounded size alphabet.

In [8] a novel method was presented for parameterized matching by constructing *parameterized suffix trees*, which also allows for online parameterized 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  $s'$  parameterize 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 [20] and faster constructions were given by Cole and Hariharan [12].

One of the interesting problems in web searching is searching for color images, see [1, 6, 26]. 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 [23]. 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 [23].

In [10] 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 parameterize matches it. An algorithm for finding the “parameterized 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 (parameterized 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  $\pi$  defined on the alphabet of  $s$ , the  $\pi$ -mismatch is the Hamming distance between the image under  $\pi$  of  $s$  and  $s'$ . The minimal  $\pi$ -mismatch over all bijections  $\pi$  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 where 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. [16, 22], also known as pattern matching with mismatches, seem to fail. Following is an outline of the method [16] for pattern matching with mismatches.

The pattern is compared separately to each suffix of the text, beginning at locations  $1 \leq i \leq n$ . Using a suffix tree of the text and precomputed longest common ancestor (LCA) information (which can be computed once in linear time, see [17, 25]) 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 parameterize match the substring of the text that they are aligned with. However, it is possible that combined they do not form a parameterized match. See the example below. In the example  $abab$  parameterize matches  $cdcd$  followed by a mismatch and subsequently followed by  $abaa$  parameterized matching  $efee$ . However, different  $\pi$ 's are required for the local parameterized matches. This example also emphasizes why the definition of [10] is a simplification. Specifically, each local parameterized matching substring is one replacement, i.e.  $abab$  with  $cdcd$  is one replacement and  $abaa$  with  $efee$  is one more replacement. However, a more precise definition would capture the globality of the parameterized matching, not allowing, in this case  $abab$  to parameterize match to two different substrings.

$$\begin{aligned}
 p &= \quad a b a b a a b a a \dots \\
 t &= \dots c d c d d e f e e \dots
 \end{aligned}$$

In this paper we consider the problem of *parameterized matching with  $k$  mismatches*. The parameterized matching problem with  $k$  mismatches seeks all *parameterized 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 an  $O(m^{1.5})$  algorithm for the problem by using maximum matching algorithms. For the string comparison problem with threshold  $k$ , we show an  $O(m + k^{1.5})$  time algorithm. We also show that improving on either of these will lead to better maximum matching algorithms for sparse graphs - a long open question.

The main result of the paper is an algorithm that solves the parameterized matching 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*  $s = s_1s_2\dots s_n$  of length  $|s| = n$  over an alphabet  $\Sigma_s$ , and a bijection  $\pi$  from  $\Sigma_s$  to some other alphabet  $\Sigma_{s'}$ , the *image* of  $s$  under  $\pi$  is the string  $s' = \pi(s) = \pi(s_1) \cdot \dots \cdot \pi(s_n)$ , that is obtained by applying  $\pi$  to all characters of  $s$ .

Given two equal-length strings  $w$  and  $u$  over alphabets  $\Sigma_w$  and  $\Sigma_u$  and a bijection  $\pi$  from  $\Sigma_w$  to  $\Sigma_u$  the  $\pi$ -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  $\pi$  such that the  $\pi$ -mismatch  $\leq k$ . The *approximate parameterized match* between  $w$  and  $u$  is the minimum  $\pi$ -mismatch (over all bijections  $\pi$ ).

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  $\pi$  yielding minimum  $\pi$ -mismatch for  $y$  at each position of  $x$ . Of course, the best  $\pi$  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  $\pi$  itself is also desired (however, here any  $\pi$  with  $\pi$ -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,  $s = s_1, s_2, \dots, s_m$  and  $s' = s'_1, s'_2, \dots, s'_m$ , and an integer  $k$ .
- **Output:** True, if there exists  $\pi$  such that the  $\pi$ -mismatch of  $s$  and  $t$  is no more than  $k$ .  
False, otherwise.

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  $\pi$ -mismatch for every possible  $\pi$ . 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 maximum weighted matching in  $B$  corresponds to a minimal  $\pi$ -mismatch, where  $\pi$  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. [13, 14, 15, 24]. For integer weights, where the largest weight is bounded by  $|V|$ , the fastest algorithm runs in time  $O(E\sqrt{V})$  [24]. 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 the case where we have a general text (i.e. of length longer than  $m$ ), which is 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.

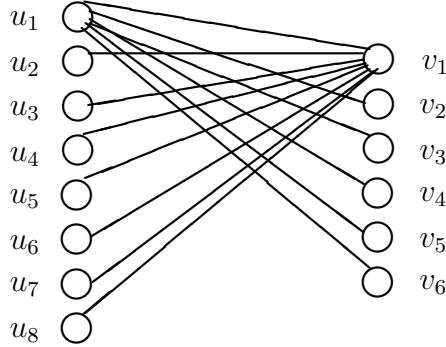
### 3.1 Reducing Maximum Matching to the String Comparison Problem

While we have seen that the maximum matching problem can be used to solve the string comparison problem, we will now see the reverse.

**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:** Let  $U = \{u_1, u_2, \dots, u_{|U|}\}$ ,  $V = \{v_1, v_2, \dots, v_{|V|}\}$  and  $E = \{e_1, \dots, e_{|E|}\}$ . Create two strings  $s_U$  and  $s_V$  of length  $|E|$ , where for edge  $e_i = \{u_j, v_l\}$  we set the  $i$ th location of  $s_U$  to be  $u_j$  and the  $i$ th location of  $s_V$  to be  $v_l$ . See example below.

We let  $s_U$  and  $s_V$  be the input of the string comparison problem and note that, by construction, the alphabet of  $s_U$  is  $U$  and the alphabet of  $s_V$  is  $V$ . Hence a bijection in the string comparison problem is a bijection from  $U$  to  $V$ , which is a matching in  $B$ . In the string comparison problem we desire to minimize the mismatches which are those edges that are not in the bijection and in the matching problem we desire to maximize the number of edges in the bijection. Since these two objective functions are equivalent the result follows.  $\square$



$$E = \{(u_1, v_1), (u_1, v_2), (u_1, v_3), (u_1, v_4), (u_1, v_5), (u_1, v_6), (u_2, v_1), (u_3, v_1), (u_4, v_1), (u_5, v_1), (u_6, v_1), (u_7, v_1), (u_8, v_1)\} \text{ and}$$

$$s_U = u_1 u_1 u_1 u_1 u_1 u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 \text{ and } s_V = v_1 v_2 v_3 v_4 v_5 v_6 v_1 v_1 v_1 v_1 v_1 v_1.$$

**Example:** A bip. graph and its corresponding strings. The max. matching and bijection is of size 2.

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 comparison 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 parameterized 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  $\pi$  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.  $s_i = s_j$  and  $s'_i \neq s'_j$
2.  $s_i \neq s_j$  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  $S \subseteq \{1, \dots, m\}$  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  $S$ .  $\square$

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 parameterized 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  $\Pi$ -mismatch is greater than  $k$ . If  $|L| \leq k/2$  then there exists  $\Pi : \Sigma_s \rightarrow \Sigma_{s'}$  such that the  $\Pi$ -mismatch counter is less than or equal to  $k$ .

**Proof:** Combining Lemma 2 and Lemma 3 yields the proof.  $\square$

### 3.3 Parameterized Comparison of Two Equal Length Strings

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 immediately announce a match or mismatch 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. We do the search in two phases. First mismatch pairs are searched according to the first rule of definition 1, i.e.  $s_i = s_j$  but  $s'_i \neq s'_j$ . This is done by scanning the strings from left-to-right and for each character  $s_i = \sigma$  of the first string  $s$  we go to the stack designated for  $\sigma$  and do the following. If it is empty we push the character  $s'_i$  of the second string  $s'$  onto the stack along with the location  $i$ . If it is not empty we check the top of the stack designated for  $\sigma$  and compare the character at the top of the stack with  $s'_i$ . If they are the same we push  $s'_i$  and  $i$  onto the stack. If they are not the same we pop from the stack the  $\langle \text{character}, \text{location} \rangle$  pair  $\langle \tau, j \rangle$  and  $(i, j)$  is declared a mismatch pair.

In the second phase we consider the locations that have not been declared as mismatch pairs and seek for mismatch pairs that satisfy the second rule of definition 1, i.e.  $s_i \neq s_j$  but  $s'_i = s'_j$ . This is done as in phase one with the roles of  $s$  and  $s'$  reversed (and disregarding locations that have already been declared as part of mismatch pairs).

**Time Complexity:** The algorithm makes two simple scans of the strings and each character in the string is pushed/popped onto at most one stack. Hence the running time and space is  $O(m)$ , where  $m$  is the length of the strings.

### 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  $\pi$ -mismatch over all bijections  $\pi$ . The technique used is similar to the bipartite matching algorithm discussed in the beginning of the section. It is easier to comprehend the algorithm if it is assumed that the alphabets  $\Sigma_s$  and  $\Sigma_{s'}$  do not have characters in common, even though this assumption is not necessary at all.

Let  $\hat{L} \subset \{1, \dots, m\}$  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  $i \in \{1, \dots, m\} - \hat{L}$  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  $i \in \{1, \dots, m\} - \hat{L}$  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  $\{1, \dots, m\} - \hat{L}$  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, we may remove every edge with weight  $> k$ , along with their vertices. Note that we will need to account for the other edges that were removed and connected to one of these 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:** A bijection  $\pi : \Sigma_s \rightarrow \Sigma_{s'}$  is equivalent to a matching in the constructed graph  $B$  as each is a one-one mapping of the same vertex/alphabet set. In a maximum matching we desire to maximize the sum of the weights of the edges chosen and in a bijection to minimize the number of "edges" that are not selected - the mismatches. These objective functions are equivalent.

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 [24].  $\square$

**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 An Algorithm for Parameterized Matching with $k$ Mismatches

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

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

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. The output of the preprocessing phase is described in section 4.1. 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  $s'$ .*

$$\begin{array}{l}
s = \textcircled{A} \textcircled{B} \textcircled{C} \textcircled{A} B B \textcircled{A} B \textcircled{A} \\
s' = D F D F F F E F D
\end{array}$$

**Example:** A maximal set of 3 mismatch pairs. This is a 3-good witness and a 2-bad witness.

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  $t_i \dots t_{i+m-1}$ . Obviously, by Corollary 1, if there is a  $k$ -bad witness then  $p$  cannot parameterize  $k$ -match  $t_i \dots t_{i+m-1}$ . Hence, after the filter stage, it remains to verify for those locations  $i$  which have a  $k$ -good witness whether  $p$  parameterize  $k$ -matches  $t_i \dots t_{i+m-1}$ . 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 [21]. Like the KMP algorithm [19] 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:

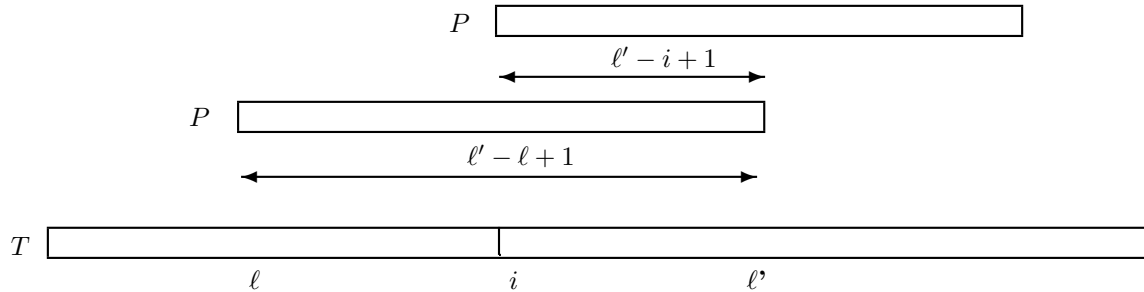
**Output of pattern preprocessing:** For each location  $i$  of  $p$ , a maximal disjoint collection of mismatch pairs  $L$  for some pattern prefix  $p_1, \dots, p_{j-i+1}$  and  $p_i, \dots, p_j$  such that either  $|L| = 3k + 3$  or  $j = m$  and  $|L| \leq 3k + 3$ .

As the first step of the algorithm we consider the first text location, i.e. when text  $t_1 \dots t_m$  is aligned with  $p_1 \dots p_m$ . 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  $p_1, \dots, p_j$  of the pattern and  $t_i, \dots, t_{i+j-1}$  we will stop our search, since this immediately implies that  $p_1, \dots, p_m$  and  $t_i, \dots, t_{i+m-1}$  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  $\ell$  which has a maximal stop location, over all locations  $l < i$ , is called a *maximal stopper* at  $i$ .

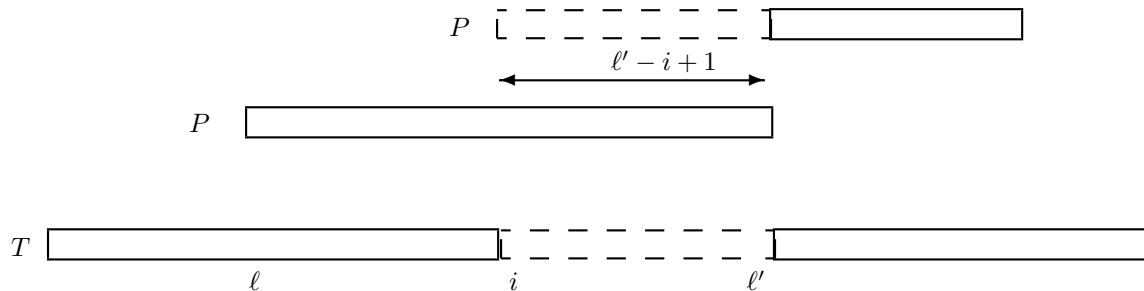
The reason for working with the maximal stopper at  $i$  is that the text beyond the stop location of the maximal stopper has not yet been scanned. If we can show that the time to compute a  $k$ -bad witness or find a maximal disjoint collection of mismatch pairs for location  $i$  up until the maximal stopper, i.e. for  $p_1, \dots, p_{\ell-i+1}$  with  $t_i, \dots, t_{\ell}$  where  $\ell'$  is the stop location of  $\ell$ , 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 Computing Up Until the Maximal Stopper



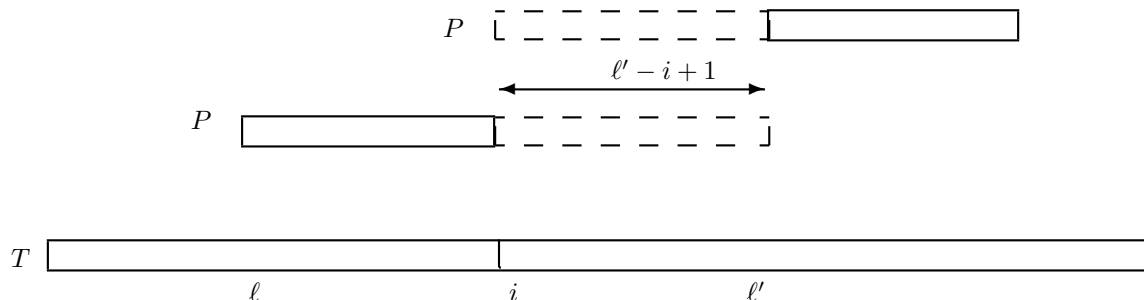
**Figure 1:** The maximal stopper and the current location.

The situation now is that we are evaluating a location  $i$  of  $t$ . Let  $\ell$  be the maximal stopper at  $i$ . 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  $\ell$  of  $t$ . Let  $\ell'$  be the stop location of  $\ell$ . See Figure 1. We would like to evaluate the overlap of  $p_1, \dots, p_{\ell' - i + 1}$  with  $t_i, \dots, t_{\ell'}$  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. See Figure 2.

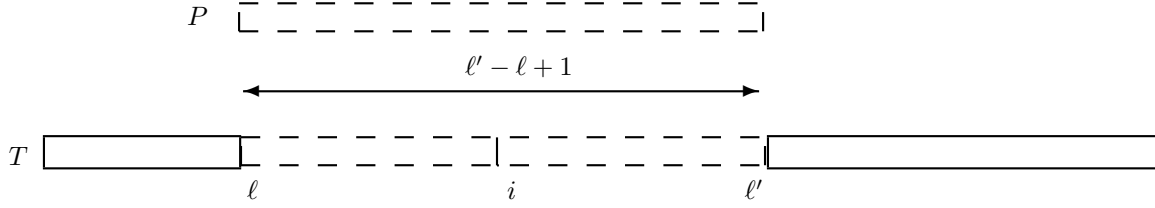


**Figure 2:** The overlap to be computed.

There are two possible cases. One possibility is that the pattern preprocessing returns  $3k + 3$  mismatch pairs for a prefix  $p_1, \dots, p_j$  and  $p_{i - \ell + 1}, \dots, p_{i - \ell + j + 1}$  where  $j \leq \ell' - i + 1$ . See Figure 3. Yet, there are  $\leq k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell' - \ell + 1}$  and  $t_\ell, \dots, t_{\ell'}$ , Figure 4.



**Figure 3:** The two cases hinge on the pattern preprocessing of this overlap.



**Figure 4:** The maximal stopper location has at most  $k + 1$  mismatches until its stop location.

**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  $s''$ .  $\square$

Set  $M$  to be  $k + 1$  and one can use lemma 5 for the case at hand, namely, there must be at least  $k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell-i+1}$  and  $t_i, \dots, t_{\ell'}$ , 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  $p_1, \dots, p_j$  and  $p_{\ell-i+1}, \dots, p_{\ell-i+j+1}$  where  $j > \ell' - i + 1$ . See, Figure 3. However, since  $\ell'$ 's stop location is  $\ell'$ , we still have an upper bound of  $k + 1$  mismatch pairs between  $p_1, \dots, p_{\ell-\ell+1}$  and  $t_{\ell}, \dots, t_{\ell'}$ . 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  $\pi : \Sigma_s \rightarrow \Sigma_{s'}$  such that  $\pi$  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  $\pi' : \Sigma_{s'} \rightarrow \Sigma_{s''}$  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 the 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.  $\square$

#### 4.1.2 Putting it All 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  $t_i, \dots, t_{i+m-1}$  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  $p = p_1, \dots, p_m$  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  $p_1, \dots, p_{j-i+1}$  and  $p_i, \dots, p_j$  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;

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

$$1 \leq i \leq \log_3 m.$$

At each step, we compute the desired mismatches for each set of alignments  $R_i$ . Step  $i$  uses the information computed in steps  $1 \dots i - 1$ . 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  $R_i^1[3^{i-1} + 1, \dots, 2 * 3^{i-1}]$  and  $R_i^2 = [2 * 3^{i-1} + 1, \dots, 3^i]$ . 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  $r_2 - r_1 \leq 2 * 3^{i-1} - 3^{i-1} - 1 = 3^{i-1} - 1$ .  $\square$

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  $R_1, \dots, R_{\log_3 m}$ ,

1. Compute a maximal disjoint collection of mismatch pairs  $L$  for some pattern prefix  $p_1, \dots, p_{j+1}$  and  $p_{f_i}, \dots, p_{f_i+j}$  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  $p_{m_i}, \dots, p_{m_i+j}$ .
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  $O(km \log_3 m)$ .  $\square$

**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

Two-dimensional parameterized matching has applications in image search with variable colors and errors. In this section we present an extension of our algorithm for 2-dimensional approximate parameterized matching.

- **Input:** Two images, text  $T[1\dots n][1\dots n]$ , and pattern  $P[1\dots m][1\dots m]$ , 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  $n$ . 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.*

## 7 Future Work

The first obvious question to be asked is whether faster solutions for the problem exist. For the one-dimensional case the reduction to bipartite matchings seem to make this a somewhat challenging problem to answer. However, the 2-dimensional case leaves room for improvement.

Another interesting question, suggested by an anonymous referee, is to consider the problem of enumerating all the bijections of the solution. Recall that the solution is approximate and hence the bijection that maximizes matches need not be unique. In this case, would the time complexity still be independent of the size of the alphabets? This can be further explored asking for all bijections matching within threshold  $k$ , i.e. all bijections that are within  $k$  mismatches of the maximal approximate match.

Naturally, one may ask what happens if you consider the edit distance? Recall the discussion in the introduction which pointed to the work of Baker [10]. However, as alluded to in the introduction the correct way for defining the edit distance problem, in our opinion, is to allow the operations and then apply the edit distance. This gives a global definition which makes for another interesting problem.

## 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)*, 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)*, 929–942, 2003.
- [3] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. on Computing*, 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] G.P. Babu, B.M. Mehtre, and M.S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, 1995.
- [7] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation (STOC)*, 71–80, 1993.
- [8] B. S. Baker. Parameterized pattern matching: Algorithms and applications. *J. of Computer and System Sciences*, 52(1):28–42, 1996.
- [9] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. on Computing*, 26(5):1343–1362, 1997.
- [10] B. S. Baker. Parameterized diff. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 854–855, 1999.
- [11] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.
- [12] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, 407–415, 2000.
- [13] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *J. of the ACM*, 34(3):596–615, 1987.
- [14] H. N. Gabow. Scaling algorithms for network problems. *J. of Computer and System Sciences*, 31:148–168, 1985.
- [15] H. N. Gabow and R.E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. on Computing*, 18:1013–1036, 1989.
- [16] Z. Galil and R. Giancarlo. Improved string matching with  $k$  mismatches. *SIGACT News*, 17(4):52–54, 1986.
- [17] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *J. of Computer and System Sciences*, 13:338–355, 1984.

- [18] C. Hazay, M. Lewenstein, D. Sokol. Approximate Parameterized Matching. *Proc. of 12th Annual European Symposium on Algorithms (ESA)*, 414–425, 2004.
- [19] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6:323–350, 1977.
- [20] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. *Proc. 36th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 631–637, 1995.
- [21] G.M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching. *Proc. 18th ACM Symposium on Theory of Computing (STOC)*, 220–230, 1986.
- [22] G.M. Landau and U. Vishkin. Fast string matching with  $k$  differences. *J. of Computer and System Sciences*, 37(1):63–78, 1988.
- [23] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
- [24] 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.
- [25] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. on Computing*, 17:1253–1262, 1988.
- [26] M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.