Approximate Pattern Matching Using Locally Consistent Parsing

Süleyman Cenk Ṣahinalp *   Uzi Vishkin †

Abstract

A key approach in string processing algorithmics has been the labeling paradigm [KMR72], which is based on assigning labels to some of the substrings of a given string. If these labels are chosen consistently, they can enable fast comparisons of substrings. Until the first optimal parallel algorithm for suffix tree construction was given in [SV94], the labeling paradigm was considered not to be competitive with the most efficient approaches. In this paper we show that this general method can be used to obtain a linear time, deterministic algorithm for the Approximate String Matching problem. The approximate string matching problem deals with finding all substrings of a text which match a pattern "approximately", i.e., with at most m differences. The differences can be in the form of inserted, deleted, or replaced characters.

*Department of Computer Science, University of Maryland, College Park, MD 20742; and Information Sciences Center, Bell Laboratories, Murray Hill, NJ 07974.

†Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, College Park, MD 20742; and Dept. of Computer Science, Tel Aviv University, Tel Aviv, Israel;
partially supported by NSF grant CCR-9416890.

1 Introduction

As the size of electronically stored information grows rapidly, efficient methods for string processing are becoming critical. In this paper, we extend our work on parallel construction of a suffix tree [SV94] in a nontrivial manner, to obtain an efficient algorithm for the approximate string matching problem.

Given a text string T, a pattern string P, and an integer m, the approximate string matching problem deals with finding all substrings of T string that match P, with the exception of at most m differences. The differences can be in the form of inserted, deleted or replaced characters. Several deterministic algorithms for solving this problem are given in [Se80], [LV85], [LV86], [GG88], [LV88], [LV89], [GP90], and numerous others. All of these have worst case running time of $\Omega(tm)$.

A probabilistic algorithm is given in [CL90], which on the average runs in $O(mt \log p/p)$ time for random strings. The worst case running time of this algorithm is $Omega(tm)$ again.

In this paper, we provide a deterministic algorithm for the approximate string matching problem, which runs in $O(poly(m)t/p + t)$, hence achieves linear time for $poly(m) < p$.

1.1 Organization of the Paper

We start our presentation with some preliminaries. Then, we describe a new linear time string matching algorithm, which demonstrates some of the main ideas behind our approach. Finally we present our algorithm for approximate string matching.

1.2 Some Preliminaries

Throughout the paper, the text string is denoted by T and the pattern string is denoted by P. Both P and T consist of characters from a constant size alphabet $B = \{0, 1, \ldots, \beta - 1\}$. We denote stings by capital letters, P, R, S, T.

Given a string R, its size is denoted by r and its characters are denoted by $r_1, r_2, \ldots, r_r$. The substrings $r_i, r_{i+1}, \ldots, r_{i+j}$ (of successive characters with indices $1 \leq i \leq j \leq r$) of R are denoted by $R_{i,j}$. The reverse of R, is the string $r_r, r_{r-1}, \ldots, r_1$, and is denoted by $R^{\leftarrow}$, A superstring S of R is a string that includes R as a substring.

A string in which all characters are identical is called a *repetitious string*. We denote by a^l, a repetitious string composed of l repetitions of character a. Similarly, for a given substring Q, we denote by Q^l, a string obtained by concatenating l substrings that are identical to Q. A string in which no character is identical to neither its left nor its right neighbor is called a *varying string*.

Given string S, a *max-repetitious* substring of S is a substring of at least two characters which has no repetitious superstrings. Following the removal of all max-repetitious substrings of S, we get a number of strings which are called the *remaining* substrings of S.

A given string S is called *periodic* with $l > 1$ repetitions, if there exists a substring Q such that $S = Q^l Q^-$ where Q^- is a prefix of Q. We call Q a period of S. The shortest period R of S, is called *the period* of S. Given the period R, a substring U obtained by concatenation of some suffix, $R_{i+1,r}$, with its remaining prefix, $R_{1,i}$ (so that $U = R_{i+1,r}, R_{1,i}$) is called a *shifted period* of S.

Given an integer x, we denote by $\log^* x$, $\min\{i \geq 0 : \log^{(i)} x \leq 1\}$, where $\log^{(i)} x = \log(\log^{(i-1)} x)$, and $\log^{(0)} x = 0$. In the remainder of the paper, unless otherwise stated, all logarithms are base 2, and where non-integer values are referred to in integer context, their ceiling is taken.

2 A New String Matching Algorithm

In this section, we present a new linear time algorithm for the exact string matching problem. Below, we describe a slightly slower algorithm which runs in $O(t \log^* t + p)$ time. We discuss how to improve the running time to $O(t + p)$ in section 2.6.

2.1 Preview

This subsection overviews the algorithm for strings in which no substring is periodic. (This is called the *no-local-periodicity* case, and the reader should be alert that a string which is non-periodic does not necessarily fall under the no-local-periodicity case.) Our detailed presentation beginning with the next subsection deals with the general case, where periodicities are possible.

Given the pattern P and the text T, the first step of our algorithm designates some substrings of T and P as *cores*. Two cores within P (or within T) may overlap. Each core is given a unique label in a way that two cores are labeled identically if and only if the cores themselves are identical. What will make this core designation a powerful algorithmic tool is that we will be able to prove that having a long core match is a necessary condition for having a match of P in T. Therefore we first find long core matches between P and T and then extend this initial match to find full matches.

The most subtle part in our algorithm is the computation of cores. Below we describe how we compute the cores of a given string S in iterations. We note that during the execution of the string matching algorithm, we compute the cores of P and T together, guaranteeing that identical cores are identically labeled.

Initially all single character substrings of S are designated as level-0 cores. The input to a later iteration i is the cores of level-$(i-1)$ and its output is the cores of level-i. Let the level-$(i-1)$ cores of S be given in a string $S(i-1)$ whose characters are the labels of the level-$(i-1)$ cores of S in the order they appear in S. Suppose that the total number of different cores encountered in the first $i-1$ iterations is δ. We designate some of the size $O(\log^* s)$ substrings of $S(i-1)$ as *blocks*, by applying a so-called *LCP* technique, which we describe in 2.3. These blocks may overlap. Throughout the paper a given block will imply the following level-i core: Consider the labels in the block. Each label represents a core of level-$i-1$. Concatenate these cores while excluding the overlaps among successive cores j and $j+1$ from core $j+1$. For convenience, we will avoid re-mentioning this exclusion of overlaps later on. No confusion should arise.

The cores satisfy the following important properties.

Property 1 (Number of Cores) *The number of level-i cores of a given string T is upper bounded by* $t/2^i$ *and is lower bounded by* $t/3^i - 2(\log^* t + 3)$. *Similarly, the number of cores of P at level-i is upper bounded by* $p/2^i$ *and is lower bounded by* $p/3^i - 2(\log^* t + 3)$.

Property 2 (Noticeability of Cores) *Given two identical substrings Q and R (of T or P), if R is a core, then so is Q.*

The cores of a given string, the exclusion of overlaps, and the number and noticeability properties are demonstrated in Figure 1.

After computing the cores of both P and T, the algorithm designates the leftmost core in P of its highest level as the *main core* of P. We use the notion of *fat-trees* to illustrate the hierarchical structure of the cores.

Fat-Tree of a string: A fat-tree is a directed acyclic graph (DAG) which differs a standard rooted tree by the fact that a node of a fat tree can have more than one parent or no parents at all. Specifically: (i) Each core is a node in the fat tree.
(ii) The child nodes of a core at level-i are those nodes of level-i whose labels are contained in the block of this core.
The leaves of the fat-tree are the characters of the string (the level-0 cores), themselves.

Figure 2 illustrates how the fat-tree of the string S in figure 1 looks like.

Based on the core properties, we make the following observations:
The main core of P is at least of level $h = \log_3 p/2(\log^* t + 3)$.
If a substring W of T matches P, then the substring V of W which is identical to the main core of P should be a core as well.

Our algorithm works as follows. We first find all cores in T which are identical to the main core of P. Each such core implies a candidate substring S which may match P: S is the substring whose main core occurs at the same relative location as the main core of P. We derive these candidate substrings.

Due to the noticeability property, no substring of T other than these candidates can match P. Due to the number property, the number of cores in T at level h is upper bounded by $t/2^h$. Since, $h \geq \log_3 p/2(\log^* t + 3)$, the upper bound on the number of substring candidates is $t/2^{\log_3(p/2(\log^* t+3))} = t(2(\log^* t + 3)/p)^{1/\log 3}$. We demonstrate how to verify if a given candidate actually matches P in $O(\log p)$ time in section 2.5. The total time for executing the algorithm is therefore $O(t \log p(2(\log^* t + 3)/p)^{1/\log 3})$, which is linear in t.

We describe how to compute the cores for arbitrary strings in section 2.2. Then we describe the locally consistent parsing (LCP) technique in section 2.3. We state some of the properties of the LCP technique in section 2.4, and the properties of the cores with proofs in section 2.5. We finally give our string matching algorithm in detail in section 2.6.

2.2 Core Computation in the Presence of Periodicities

Given a string S, we show how to designate and label its cores. In the no-local-periodicity case (described in the preview), we assumed that no substring of the input strings were periodic. Hence not only the cores, but also the substrings of the cores were non-periodic. In the general case, we have cores which can either be periodic or non-periodic, and even the cores which are non-periodic may have substrings that are periodic.

The cores of S are again designated, and labeled in $O(\log s)$ iterations as follows. Initially, we designate each character of S as a core of level-0. For upper levels, we will designate two types of cores: (i) *replicative cores*, and (ii) *non-replicative cores*. The replicative cores are periodic strings. The non-replicative cores can either be periodic or non-periodic strings.

Below we first describe iteration 1 of the core computation. Although this iteration has the same basic steps as the later iterations it has to be treated separately. Following this, we proceed to the description of later iterations.

Iteration 1 of the core computation: In iteration 1, we compute the level-1 cores of S. The input to iteration-1 is the string S itself.

Replicative cores of level-1: We designate every max-repetitious substring $R = a^r$ for which $r > \log^* s$ as a replicative core (as well as a block) of level-1. We label R with the following 3-tuple: (i) *character* entry, which is a, (ii) *level* entry, which is 0, (iii) *length* entry, which is r.

Non-replicative cores of level-1: We obtain the non-replicative cores as follows.

1. **Marking characters:** We first mark the following characters of S:
   (a) In each max-repetitious substring and in each remaining substring of length $\leq \log^* s + 3$, we mark the leftmost character and every second (i.e., odd indexed) character to its right. We do not mark the rightmost character.
   (b) In each remaining substring of length $r > \log^* s + 3$, we mark the characters implied by the LCP (see section 2.3).
2. **Obtaining blocks:** For each marked character x, we try to obtain a block as follows: Consider (1) the substring of size $\log^* s + 3$ to the immediate right of x and (2) the substring of size $\log^* s + 3$ to the immediate left of x. If both of these substrings exist, we append substring (1) to the right of x and substring (2) to the left of x, to obtain its block.
3. **Computing cores:** In level-1, each block is a core itself. The character x is called the *center character* of the core.
   Labeling cores: We label the non-replicative level-1 cores from left to right. The leftmost non-replicative core is labeled with the integer β. Each time we encounter a new non-replicative core, we label it with the integer j, where j denotes the number of distinct non-replicative cores encountered up to that point. (Note that the integers $0, \ldots, (\beta - 1)$ are used to label level-0 cores.) This labeling method is used in all iterations.

4. **Updating the fat-tree:** For each level-1 core, we create a level-1 node in the fat-tree. The children of such a node C will be the leaves (or the level-0 nodes) of the fat-tree, whose characters are in C.

Iteration i of the core computation: The input to iteration-i of the core computation is the string comprised of (1) the labels of non-replicative cores of level-$(i-1)$, and (2) the character entries of the labels of replicative cores of level-$(i-1)$, in the order in which their respective cores appear in S. We denote this string by $S(i-1)$.

Replicative cores of level-i: We designate two types of replicative cores of level-i.

1. We designate as a block, every max-repetitious substring $R = a^r$ of $S(i-1)$ for which $r > \log^* s$. We label R with a 3-tuple comprised of the entries: (i) the character of R, which is a, (ii) the level of character a, which is $i-1$, (iii) the length of R, which is r. The core of this block is the concatenation of the cores of the labels in it. Again we create a node in the fat-tree for this core and connect it to its children, which are all the level-$(i-1)$ cores in its block.
2. Consider each character x in $S(i-1)$ which represents a level-$(i-1)$ replicative core C. Let the level entry of x be l, and length entry of x be c. If $c \geq (2^{i-l} - 1)(\log^* s + 3) + 2^{i-1}$, we designate this single character as a *singleton block*. We use the 3-tuple label of C to label this singleton block, and C itself as its core.

We observe that for a replicative core C designated by possibility 1, there might exist a chain of parents in the fat-tree, each having a single child. Each such parent node, then, would represent the same string, C, as a core.

Non-replicative cores of level-i: We designate two types of non-replicative cores of level-i.

1. From each remaining substring Q, we remove the characters which represent a level-$(i-1)$ replicative core. The substrings of Q that remain are called the *leftover* substrings of Q. We obtain several cores out of Q as follows.
   (a) **Marking characters:** We then *mark* the following characters of $S(i-1)$:
      i. In each max-repetitious or leftover substring R of length $r \leq \log^* s + 3$, we mark the leftmost character, and every second (i.e. odd indexed) character to its right. We do not mark the rightmost character of R.
      ii. In each leftover substring of length $r > \log^* s + 3$, we mark the characters implied by LCP.
   (b) **Obtaining blocks:** For each marked character x, let y be the leftmost character to the right of x, which is designated as a singleton block. Similarly, let z be the rightmost character to the left of x, which is designated as a singleton block. Also, let k denote the number of characters between x and y, and l denote the number of characters between x and z.
   We append the leftmost $\min(k+1, \log^* s + 3)$ characters to the right of x, and the rightmost $\min(l+1, \log^* s + 3)$ characters to the left of x. Such a

substring is called the block of character x. However, we avoid designating a block for x if to the left (respectively, right) of x, there are fewer than $\log^* s + 3$ characters and no singleton block.

(c) **Computing cores:** For each block obtained, we compute its core C according to the following possibilities.
(1) If the leftmost character is a singleton block and the lowest level in which it occurred as a core (the so-called level entry) is m, then we take the rightmost $2^{i-m} \times (k+1-\log^* s - 3)$ level-m cores of the singleton block. To their right we concatenate all the cores in the block of x.
(2) If the leftmost character is not singleton block we simply concatenate all the cores in the block of x.
However, in either case we check to see whether the rightmost character is a singleton block and the level entry in which it occurred as a core is n, then we take the leftmost $2^{i-n} \times (k+1-\log^* s - 3)$ level-n cores of the singleton block. Again: (i) all concatenations exclude overlaps, and (ii) character x will be the center character of core C.

(d) **Updating fat-tree:** For each level-i core we obtained, we create a node of level-1 in the fat-tree. The children of such a node are the nodes of the characters (labels) its block covers. However, it is important to notice that we now also allow a different interpretation to an edge in the tree: an edge connecting a core C at level-i with a replicative core D at level-$(i-1)$ does not imply that D is fully included in C.

2. Finally we consider each character x which represents a level-$(i-1)$ replicative core, but is not designated as a singleton block; i.e., the level entry of x, denoted l, and length entry of x, denoted c, should satisfy $c < (2^{i-l-1} - 1)(\log^* s + 3)$. We treat x as if it is a max-repetitious substring of $S(i-1)$ of size $c/2^{i-l-1}$. Therefore, we obtain $c/2^{i-l}$ blocks that include x. These blocks and their respective cores are computed as follows.

   (a) **Obtaining blocks:** Let y be the leftmost character to the right of x which is designated as a singleton block. Similarly let z be the rightmost character to the left of x which is designated as a singleton block. Let k denote the number of characters between x and y, and l denote the number of characters between x and z.
   To obtain the leftmost block, we append x the following substrings. To its right, the substring of size $\min(k+1, \log^* s + 3 - 1)$ to the immediate right of x. To its left, the substring of size $\min(l+1, \log^* s + 3 - c/2^{i-l-1} + 1)$ to the immediate left of x. Similarly, to obtain the j^{th} block from the left, we append x the following substrings. To its right, the substring of size $\min(k+1, \log^* s + 3 - (2j+1))$ to the immediate right of x. To its left, the substring of size $\min(l+1, \log^* s + 3 - c/2^{i-l-1} + (2j+1))$ to the immediate left of x. Again we do not designate a block if to the left or to the right of x, there are fewer than the specified number of characters and no singleton block.

(b) **Computing cores:** The cores of these blocks are obtained similar to the previous case.

(c) **Updating fat-tree:** For each level-i core we obtained, we create a node of level-1 in the fat-tree, similar to the previous case.

2.3 Description of LCP

Given a leftover substring R (of a given $S(i-1)$), we mark at most $r/2$ and at least $r/3$ of its characters as follows.

1. We mark the leftmost character of R and every second (i.e. odd indexed) character to its right, till the $(\log^* s + 1)^{th}$ character.

2. For each character x to the right of the rightmost marked character so far, compute a *tag*, using a similar method to the Cole-Vishkin deterministic coin tossing [CV86], as follows.

   Let y be the left neighbor of x. The next paragraph explains why such a left neighbor always exists.
   Initially assign $tag(x) = x$.
   Compute $tag(x)$ in $\log^* s - 1$ iterations, as follows.
   Denote by i the index of the least significant bit in which $tag(x)$ is different from $tag(y)$.
   Denote by γ the value of bit i in x.
   Assign $tag(x) = 2i + \gamma$. Initially the maximum possible tag value is δ (the total number of different cores encountered in the first $i-1$ iterations). In the subsequent iterations, the maximum value of the tags would be $2\log\delta - 1$, then $2\log\log\delta$, and so on. The final possible range of values for the tags is $0, \ldots, 5$.

   Computing the tags of the leftmost $(\log^* s + 1)$ *characters.* Let us denote the indices of the leftmost characters by i, $0 \leq i \leq \log^* s$. Then, character i participates in $i-1$ rounds of the tag computation (namely, characters 0 and 1 do not participate at all). This guarantees that the tag of the left neighbor, as required in the previous paragraph always exists.

3. We mark each character whose tag is a local maxima, i.e. is greater than the tags of both its left and its right neighbors.

4. We mark each character whose tag is a local minima, i.e. is smaller than the tags of both its left and its right neighbors, and which has no neighbors that have previously been marked.

5. For each character marked above, if the closest marked character to its right is both more than three characters apart, and is in the same leftover substring, then we mark the character that is half way between these two characters.

2.4 Properties of LCP, and its implied blocks

Lemma 1 (Tag Computation) *Given any character x in a given string $S(i-1)$, $tag(x)$, is a function of at most $\log^* s$ characters to its left and one character to its right.*

Proof: In the first iteration of the tag computation, $tag(x)$, is a function of character x and its left neighbor. In each of the following iterations, $tag(x)$ becomes a function of an additional character to the left. □

Lemma 2 (Noticeability of Blocks) *Suppose that a given iteration i designates a substring R as a block. If R is a repetitious substring, then for each substring Q which is identical to R, its max-repetitious superstring will also have to be designated as a block. Otherwise (that is R is not repetitious), iteration i designates all substrings Q which are identical to R as blocks as well.*

Proof: If R is not repetitious, then it has to be obtained by marking a character x and then appending at most $\log^* s + 3$ characters to its left and $\log^* s + 3$ characters to its right. Because of Lemma 1, and the construction of blocks, all the characters that enable x to be marked are in its block, hence the same x will be marked in Q, as well. If R is repetitious, then to be designated as a block, it should be of size $> \log^* s + 3$. Therefore for any substring $Q = R$, the max-repetitious superstring of Q should be returned as a block. □

Lemma 3 (Unequal Neighbors Get Unequal Tags) *Given two successive characters of a substring on which LCP is applied, their tags are different.*

Proof: Let a character x and its right neighbor y be two successive characters of a leftover substring. This means that x and y must be unequal and so must be their initial tags. The first iteration of the tag computation produces unequal tags for successive unequal characters: To see that assume, in contradiction, that given the characters x and y an iteration produces $tag(y) = tag(x)$, and denote by z the left neighbor of x. The index of the least significant bit in which x and y are different should be the same as the index in which z and x are different. For this reason the parts of $tag(x)$ and of $tag(y)$ which include the different bit values must be different. This is a contradiction. The same argument applies to each of the later iterations of the tag computation. □

Given a substring S we say that a subset of its characters form a *2-ruling set* if every character outside the subset is adjacent to a character in the subset (i.e., the character in the subset is either predecessor, or a successor of the character outside the subset).

Lemma 4 (Marked Characters in Leftover Substrings Form 2-Ruling Sets ([CV86])) *Suppose that some of the characters of a string S are marked. Then, the marked characters in any leftover substring S form a 2-ruling set.*

A proof of the claim follows by observing that the tags are chosen from the integer range $\{0, \ldots, 5\}$. Lemma 4 implies the following two lemmas.

Lemma 5 (Overlap Between Neighbor Blocks) *Suppose that x and y are two marked characters of a given string $S(i-1)$, such that x is to the left of y, and there is no singleton block, replicative block, or a marked character between them. Denote by S the block of x, and denote by R, the block of y. If the rightmost character of R is not a replicative core, then the characters of S which are not in R are either the two or the three leftmost characters of S. Similarly, if the leftmost character of S is not a replicative core, then the characters of R which are not in S are either the two or the three rightmost characters of R.*

Lemma 6 (Number of Blocks) *Given a string S, the number of its blocks is at most $s/2$.*

Lemma 7 (Margin Blocks) *For any given string S, every character except possibly the rightmost $\log^* s + 3$ and the leftmost $\log^* s + 3$ characters is included in at least one block.*

Proof: If a given character x is in a remaining substring, or is in a max-repetitious substring of size $\leq \log^* s + 3$, then lemma 5 implies that it is at most two characters apart from its closest marked character (which is to its right or to its left). If x is not one of the rightmost (or the leftmost) $\log^* s + 3$ characters of S, the block obtained for the marked character to the left (right) of x which includes x itself would be returned by LCP. If x is in a max-repetitious substring of size $> \log^* s + 3$, then this substring is returned as a block. Hence the proof follows.

□

2.5 Properties of Cores

The following two properties are implied by lemma 5.

Lemma 8 (Fat Tree) *The fat tree obtained during the computation of the cores of a given string S has $\leq \log s$ levels, and the number of parents that each node can have is at most $\log^* s + 3$.*

The following property states upper and lower bounds on the total number of cores at a given level-i using a weighted sum $W(i, s)$: Each non-replicative core C at level-i contributes 1 to the sum. Each replicative core D contributes $k/2^{i-l}$ to the sum, where l denotes the level entry, and k denotes the length entry of D.

Lemma 9 (Number of Cores) *$W(i, s)$ is upper bounded by $s/2^i$ and is lower bounded by $s/(3^i - 2(\log^* s + 3))$.*

The following property is a corollary to lemma 2.

Lemma 10 (Noticeability of Cores) *Given a string S, let R be a level-i core of S, and let Q be a substring of S identical to R. If R is a non-replicative core, then Q is a non-replicative core of level-i. If R is a replicative core, then there exists a superstring U of Q which is a replicative core with same level and character entries with that of R.*

Definition 1 (Compact Representation of S**)** *The string obtained by concatenating the labels of the cores of S which do not have any parents, in the order they appear in S, is called the compact representation of S.*

We demonstrate the compact representation of the string in Figure 2 in Figure 3.

The *compact representation* of a given substring R of S is defined as follows. The part of the fat-tree of S which pertains to R consists of all cores which fully fall with the boundaries of R. Removing all other nodes from the fat-tree of S will provide a reduced fat-tree for R, which gives the compact representation of R.
Caveat: The compact representation of a substring R should be derived from the fat tree of the full string S with care since the margin nodes (relative to R) at any level may represent different substrings than their respective nodes in the fat tree of S. A simple top-down construction works.

The lemma 10 leads us to the most important property of cores.

Theorem 11 (Compactability of Cores) *1. Let Q and R be two substrings of S, and consider the reduced fat trees of Q and R and their compact representations. The compact representation of Q is identical to the compact representation of R if and only if Q and R are identical.*
2. As per the preview, the algorithm builds the cores of of T and P together guaranteeing that identical cores are identically labeled. Let Q be a substring of P and R be a substring of T. Then the claims of Part 1 applies to them as well.

The following lemma is implied by lemma 7.

Lemma 12 (Size of Compact Representation) *The compact representation of S is composed of at most $2(\log^* s + 3) \times h$ core labels, where $h \leq \log s$ is the highest level in the fat-tree of S.*

Given the fat-tree of S, the computation of the compact representation of any substring R takes $O(\log^* s \log s)$ time as implied by lemma 12.

Suppose a given string S is periodic. One of the interesting properties of cores is that the periodicity of S is *preserved* through the iterations of constructing the cores . (We remind that in each such iteration, one level of the fat tree is constructed.) This is articulated in the following lemma.

Lemma 13 (Preservation of the Periodicity) *Suppose S is periodic with period R, with l repetitions; i.e., $S = R^l R^-$ for some prefix R^- of R.*
If for any given level-i, $l(r-2) \geq (3+\log^* \beta)(3^{i+1}-1)$, then the following conditions hold.
(1) $S(i)$ is periodic.
(2) Denote by W, the period of $S(i)$, so that $S(i) = W^k W^-$ for some integer k and some prefix W^- of W. Integers w and k can be bounded as follows:
$r/3^i \leq w \leq r/2^i$, and $l - 2(3+\log^* \beta)(3^{i+1}-1)/2r \leq k \leq l - 2(\log^* \beta)(2^{i+1}-1)/r$.

Proof: The proof follows inductively. Suppose $S(i-1)$ is a periodic string. $S(i-1)$ can then be written as $S(i-1) = Q^k, Q^-$, where Q is the period of $S(i-1)$, k is the number of repetitions, and Q^- is a prefix of Q. Then it follows from lemma 10 that $S(i)$ is also periodic with a period P, if the length of $S(i-1)$, $s(i-1) > 2(q+\log^* \beta)$. The size of P will satisfy $q/3 \leq p \leq p/2$. $S(i)$ will consist of at most $k - 2(\log^* \beta)/p$ and at least $k - 2(3+\log^* \beta)/p$ of its period P. Some simple arithmetic is needed to derive the lemma. □

The following lemmas are implied by lemma 10. The first one helps to determine the period of a given string S out of any periodic $S(i)$. The second one implies that by only considering the string of a given level-i and a few margin characters in S, one can determine if a string S is periodic or not.

Lemma 14 (Determination of the Period) *Given a string S and an integer i, suppose that both S and $S(i)$ are periodic. Then the string obtained by (i) concatenating the cores of the leftmost period of $S(i)$, and (ii) removing the characters which also belong to the second occurrence of the period, is a shifted period of S.*

Explanation: To see how lemma 14 works for level-1, consider $S(1)$ the string of level-1 core labels. If $S(1)$ is periodic (it might not be), then consider its leftmost occurrence of its period, A (which is a string of level-1 core labels). Now consider the second leftmost occurrence of the period of $S(1)$, B (hence A is identical to B). Each label in $S(1)$ represents a core (which is a substring of S). Concatenate the cores of A, omitting overlaps between cores. From this substring, remove the characters that are also included in any of the cores of B. This substring is a shifted period of S.

Lemma 15 (Margins of Periodic Strings) *Given a periodic string S, if for a given level-i, $S(i)$ is periodic, then the prefix of S which is not covered by any of its level-i cores can not be longer than the length of the period of S. Similarly, the suffix of S which is not covered by any of its level-i cores can not be longer than the length of the period of S.*

2.6 The String Matching Algorithm

The properties of the cores enable us to obtain the following algorithm for string matching.

1. **Computation of Cores:** Compute the cores of both P and T.

2. **Computation of the Main Core:** Find the main core M of P. Let $M = P_{i,j}$.

3. **Obtaining the Candidate Substrings:** We have three cases depending on P and M:
(1) M is a non-replicative core.
(2) M is a replicative core but P is a non-periodic string.
(3) M is a replicative core and P is a periodic string with period R.

   If case (1) applies, find all cores N of T which have the same label as M. For each such core N, append its right the substring of size $p - j$ to the immediate right of N, and append its left the substring of size $i - 1$ to the immediate left of N. This gives a *candidate substring* of size p in T.

   If either case (2) or case (3) applies, consider the label of M (which is a 3-tuple consisting of (i) the character entry, (ii) the level entry, and (iii) the length entry). Find all cores N of T satisfying the following criteria on its entries: (i) N is in the same level (of the fat-tree) with M, (ii) N has the same character and level entries, and (iii) N's length entry is $\geq j - i$.

   If case (2) applies, consider the rightmost character of N, denoted a. Append to the right of a, the substring of size $p - j$ to the immediate right of a. Then to the left of this string, append the substring of size $j - 1$ to the immediate left of a. This gives a candidate substring of size p in T.

   If case (3) applies, consider the leftmost character of N, denoted a, and the rightmost character of N, denoted b. Append to the left of a, the substring of size $i - 1$ to the immediate left of a. Then append to the right of this substring, the substring of size $p-i$ to the immediate right of a. This gives a first candidate. Now append to the right of b, the substring of size $j - 1$ to the immediate left of b. Then append to the right of this substring, the substring of size $p - j$ to the immediate right of b. This gives a second candidate. If between a and b there are more characters than $2r$, then output as a match the substrings of N whose leftmost characters are $r, 2r, 3r, \ldots$, characters to the right of a, and which are of size p. (These substrings are later referred the *guaranteed matches* of P.)

4. **Verification of Candidate Substrings for non-periodic P:** Among the candidate substrings, output (as occurrence of P in T) the ones whose compact representations are identical to that of P.

Lemma 16 (Correctness) *A substring S of T matches P if and only if it is output by the algorithm.*

Proof: Suppose that a substring S of T matches P.
(1) If M, the main core of P, is a non-replicative core, then by lemma 10, S must have a corresponding core that matches M, so that S must be determined as a candidate.
(2) If M is a replicative core, but P is a non-periodic string, then by lemma 10, one of the following cases apply: (i) S has a corresponding core that is identical to M,

(ii) S has a prefix, which matches M, and which is a suffix of a longer replicative core,
(iii) S has a suffix, which matches M, and which is a prefix of a longer replicative core.
In all three cases S must be determined as a candidate by construction.
(3) If M is a replicative core, and P is a periodic string, then by lemma 10, S must have a substring which is identical to M and is a substring of a longer replicative core. Hence S either is determined as a left or a right candidate, or is output as a guaranteed match.
Once S is determined as a candidate, it must be output as a match by lemma 11.

Suppose that a substring S of T is output by the algorithm.
Then either (1) it is output as a guaranteed match, or (2) it is output after it becomes a candidate and its compact representation is identical to that of P.
If case (1) applies, then by lemma 15 it must match P.
If case (2) applies, then by lemma 11, it must match P. $\square$

Lemma 17 (Running Time) *The algorithm described above runs in $O((t+p)\log^* t)$ time.*

Proof: Let the highest level of cores in the fat-tree of P be h. Then by lemma 9 the number of level-h cores in T, hence the number of candidate substrings are at most $t/3^h$. It takes $O(\log h)$ time to verify each candidate, so the total time for candidate designation and verification is $O(ht/3^h)$. The core computation runs in $O(\log^* t)$ time per character per iteration. As there are $O(\log t)$ iterations, and in each iteration i there are at most $(t+p)/2^i$ characters, the total time for the core computation becomes $O(\log^* t(t+p))$. $\square$

Lemma 18 (Improved Running Time) *The running time for the algorithm can be improved to $O(t+p)$.*

Proof: The time for core computation can be improved by using the fact that the number of characters in iteration i is at most β^{3i}, where β is a constant. Hence the tag computation in a given iteration i takes at most $\log^* \beta^{3i} = O(\log^* i)$ rounds per character. As the number of characters in iteration i is at most $(t+p)/2^i$, the total running time per iteration of the core computation is $O((t+p)\log^* i/2^i)$, which implies that the total running time of all the iterations is $O(t+p)$. Hence the proof follows. $\square$

3 Approximate String Matching

Given a text T, a pattern P, and an integer m, our approximate string matching algorithm finds all occurrences of P in T "with at most m differences": A substring Q of T is said to match P with $k \leq m$ differences, if we can obtain P by applying k of the following operations to Q:

- **Replacement:** Replace a character of Q with another character.
- **Insertion**: Insert a character in any location of Q.

- **Deletion**: Delete any character of Q.

In other words, Q is said to match P with k differences, if there exist (possibly empty) substrings $S_P(1) = S_Q(1), S_P(2) = S_Q(2), \ldots, S_P(k+1) = S_Q(k+1)$, and (possibly NULL) characters: $c_P(1) \neq c_Q(1), c_P(2) \neq c_Q(2), \ldots, c_P(k) \neq c_Q(k)$, such that: $P = S_P(1), c_P(1), S_P(2), c_P(2), \ldots, c_P(k), S_P(k+1)$, and $Q = S_Q(1), c_Q(1), S_Q(2), c_Q(2), \ldots, c_Q(k), S_Q(k+1)$. Henceforth, we also refer to the latter two strings as a *mapping* of the match.

We first describe an approximate string matching algorithm for the no-local-periodicity case (i.e., neither P, nor T have any substrings which are periodic) in section 3.1. We then present an algorithm for the general case in section 3.2.

3.1 Algorithm for Approximate String Matching for the No-Local-Periodicity Case

In this section we present our approximate string matching algorithm for the no-local-periodicity case (i.e. for strings which have no substrings that are periodic). The algorithm is similar to our exact string matching algorithm: We again map some of the cores of P to identical cores in T and then extend each such mapping towards left and right:

General structure of the algorithm: **Computing the cores:** Compute the cores of P and T.
Finding initial mappings: For some level-h, which is depicted in lemma 19, we use each of the level-h cores of P to obtain initial mappings between P and T. (This is unlike the exact string matching algorithm where the main core of P is used to compute initial mappings between P and T.)
Extending initial mappings: Given an initial mapping, we use an "Approximate String Identity (ASI) procedure" (which is described later) for trying to extend it to the left and to the right, in a way that the total number of differences does not exceed m.

Lemma 19 (Pigeonhole Principle) *Suppose there exists a mapping between P and a substring of T, denoted Q, with $k \leq m$ differences, and let $h = \log_3((p-m)/2(\log^* t + 3)m)$. In this mapping, at least one level-h core of P must be mapped to an identical core of Q.*

Proof: If P maps to Q with $k \leq m$ differences, then by definition one must be able to parse P as $P = S_P(1), c_P(1), S_P(2), c_P(2), \ldots, c_P(k), S_P(k+1)$, and parse Q as $Q = S_Q(1), c_Q(1), S_Q(2), c_Q(2), \ldots, c_Q(k), S_Q(k+1)$, such that for all $1 \leq i \leq k+1$:
(1) $S_P(i)$ and $S_Q(i)$ are identical substrings (with a possibility of both being empty substrings), and
(2) $c_P(i)$ and $c_Q(i)$ are non-identical characters (with a possibility of at most of them being null).
Among these substrings there must exist at least one substring $S_P(j)$ whose size is at least p/m, by the pigeonhole principle. The lemma follows from the fact that a given substring

$S(l)$ whose size is at least $(p-m)/m$, must include a core of level $\log_3(p-m)/2(\log^* t+3)m$, or higher. $\square$

Detailed description of the algorithm: First, we map each level-h core, $P_{i,j}$, of P, to each identical core of T. Let $T_{i+k,j+k}$ be such a core. We then extend this initial mapping to obtain a full mapping in three steps: As a first step, we apply the ASI procedure (as described next) to the input strings $P_{j,p}$ and $T_{j+k,p+k+m}$. The output of ASI will be the array $A[p-m:p+m]$ for which $A[l]$ denotes the minimum number of differences in a mapping between $P_{j,p}$ and $T_{j+k,l}$. As a second step we apply the ASI procedure to the input strings $P_{\overleftarrow{1,i}}$ (the reverse of $P_{1,i}$) and $T_{\overleftarrow{1+k-m,i+k}}$, to compute the minimum number of differences in a mapping between $P_{1,i}$ and $T_{l,i+k}$ for all $i \in \{1+k-m,\dots,1+k+m\}$. As a third step we compute the minimum total number of differences in matches between P and substrings of $T_{1+k-m,P+k+m}$ by simply matching all the outputs of the first step whose addition to an output of the second step does not exceed m. We demonstrate an approximate match between P and a substring of T in figure 4.

The Approximate String Identity Procedure (ASI): ASI is based on the ideas introduced in [Uk83] and [LV89]. Given input strings S and R, and an integer m, we determine all prefixes R which match S with at most m differences. We also compute the minimum number of differences in each such match.

Let $D[i,l]$ denote the minimum number of differences in a match between the prefixes $S_{1,i}$ and $R_{1,l}$. Illustratively, row i and column l are indices of the two-dimensional array D. The values of $D[i,l]$ can easily be computed via dynamic programming:
(1) if $S_i = R_l$, then $D[i,l] = D[i-1,l-1]$,
(2) if $S_i \neq R_l$, then $D[i,l] = \min(D[i-1,l-1], D[i-1,l], D[i,l-1]) + 1$. Notice that we are only interested in the values of $D[i,l]$ for $i \in \{1,\dots,s\}$ and $l \in \{i-m,\dots,i+m\}$. Hence the running time of this dynamic programming algorithm is $O(s \times m)$.

To improve the efficiency, rather than constructing D in full, we construct a two dimensional array L, which is defined as follows. Instead of referring the locations of D with row i and column l, we can use row i and diagonal $i-l$. Given a diagonal $d \in \{-m,\dots,m\}$ and a number of differences (or "errors") $e \in \{-m,\dots,m\}$, let $L[d,e]$ denote the largest row for which the value of D is e.

The values of L can be computed as follows. Suppose that for all $x < e$ and all y, $L[y,x]$ was already computed. Then we compute $L[d,e]$ as follows. First we assign $j := \max\{(L[d,e-1]+1), (L[d-1,e-1]), (L[d+1,e-1]+1)\}$. Then, by using the fat-trees of S and R, we compute the maximum k for which $S_{j,k} = R_{j+d,k+d}$. The number k is the value of $L[d,e]$.

Lemma 20 (Correctness of ASI) *The values $L[d,e]$ are correctly computed.*

Proof: The proof is a simple extension on the proof of the correctness claim given in [LV89], section 2.2, together with theorem 11 of our exact string matching algorithm. $\square$

Lemma 21 (Running Time of ASI) *The ASI procedure runs in time $O(m^2 \log s)$ time.*

Proof: We compute each $L[d,e]$ for $-m \leq d \leq m$ and $0 \leq e \leq m$. In each such computation we need to compare the compact representations of two strings which are of length at most s. Hence the lemma follows. □

Improved Running Time for ASI: For improving the efficiency, the ASI procedure can be updated as follows. The algorithm computes each $L_{d,e}$ for $-m \leq d \leq m$ and $0 \leq e \leq m$. For this, it compares the compact representations of substrings, which introduces a logarithmic factor to the running time. We can improve the running time as proposed in [LV89]: we can build the suffix tree of the string obtained by concatenating T and P, and use appropriate data structures to compute in constant time, the lowest common ancestor of any two desired substrings. This improves the running time of ASI to $O(m^2)$.

Running Time of Approximate String Matching Algorithm: As there are at most $t/2^h$ cores in level-h of the fat-tree of T, there can be at most $mt/2^h = mt/2^{\log_3((p-m)/m2(\log^* t+3))} = mt(m2(\log^* t+3)/(p-m))^{1/\log 3}$ initial mappings all together. The ASI procedure runs in $O(m^2)$ time. Hence the total running time of the algorithm is $O(tm^{3+1/\log 3}(\log^* t/p)^{1/\log 3})$, which is linear in t for $m = O((p/\log^* t)^{1/(1+3\log 3)})$.

3.2 The General Version of The Approximate String Matching Algorithm

Our algorithm for the general case is based on the ideas we presented in previous sections, as well as some new "arithmetic" properties of strings that we present in the next section. These properties enable us to derive mappings of P and T by first mapping each of them to a third "yardstick" string.

The algorithm has two main stages similar to the algorithm for the no-local-periodicity case. In the first stage, we again map some substrings of P to substrings of T. This is done via the methods presented in the previous section as well as the use of the yardstick strings. In the second stage, we check whether each initial mapping extends to a full approximate match by using ASI procedure. The following lemma generalizes lemma 19 and guides us in deriving the initial mappings for the first stage.

Lemma 22 (Pigeonhole Principle for the General Case) *Suppose P matches Q (a substring of T), with $k \leq m$ differences and let $h = \log_3((p-m)/(2(\log^* t+3)m)$. Then, in any mapping between P and Q, there exists level-h cores C of P and D of Q, which satisfy one of the following:*

- *C and D are identical and are mapped to each other.*

- *(1) C and D are replicative with identical level and character entries, and, (2) C and D have identical substrings of length $> 2^h$, which include at least one period of C and D respectively, and are mapped to each other.*

Proof: The proof is a simple extension of the proof of lemma 19 by using lemma 10. $\square$

We preview our algorithm below, and give the details in section 3.2.2.

Preview of The Algorithm: We consider each level-h core, $P_{i,j}$, of P to obtain the initial mappings. Depending on whether $P_{i,j}$ is a replicative core or not, we proceed as follows.

$P_{i,j}$ **is non-replicative:** This case works identical to the algorithm for the no-local-periodicity case. We first map $P_{i,j}$ to all level-h cores $T_{k,l}$ which are identical to $P_{i,j}$. Then, we extend each such initial mapping by using the ASI procedure.

$P_{i,j}$ **is replicative:** We first find all level-h cores $T_{k,l}$ in T which have the same character and level entries with that of $P_{i,j}$. Then, we search for possible initial mappings for P in the "vicinity" (as explained in section 3.2.2) of $T_{k,l}$. This is done as follows.

Let G be the period of $P_{i,j}$ (and hence $T_{k,l}$) and suppose $P_{i,j} = G^z G^-$ for some z and G^-. We have two cases depending on whether $z \leq m$:

$z \leq m$: We map each occurrence of G in $P_{i,j}$ to each occurrence of G in $T_{k,l}$.

$z > m$: This case has to be treated separately, as trying to map each occurrence of G in $P_{i,j}$ to each occurrence of G in $T_{k,l}$ would increase the running time beyond our desired complexity bound. We observe that any mapping of P to a given substring S of T can be described as a composite mapping of P to a third string E, and then mapping E to S. We use the string $E = G^{(p+m)/g}$ (which is slightly longer than P), to compute the mappings of P to substrings of T (in the "vicinity" of $T_{k,l}$). We handle this case in two stages:

1. **Transitive Mappings:** A composite mapping is transitive if the total number of differences in the mapping of P to E and the mapping of E to S is $leqm$ (see lemma 23). To find the transitive mappings, we first compute δ, the total number of differences δ in a mapping between P and E, and if $\delta \leq m$, we compute the substrings of T (in the "vicinity" of $T_{k,l}$) which can be mapped to E with $\leq m - \delta$ differences. We output such substrings (as matches of P).
2. **Non-transitive Mappings:** A composite mapping is non-transitive if the total number of differences in the mapping of P to E and the mapping for E to S is $> m$. In such a mapping there must exist a character c of P which satisfies the following properties (see lemma 25):
(1) c is mapped to a non-identical character d in E, which again is mapped to a non-identical character f in S (c and f can be identical),
(2) c is among the leftmost m characters in P which is mapped to a non identical character in E; d is also among the leftmost m characters in E which is mapped to a non identical character in S.

To find the non-transitive mappings, we first compute all mappings of prefixes of P to E with $leqm$ differences, and then all mappings of prefixes of E to S with $leqm$ differences. We then map the characters of P which are mapped to non-identical characters of E, to characters of S which are mapped to non-identical characters of E. We finally extend such mappings by the use of the ASI procedure.

In the following section, we state some of the properties of approximate mappings between two given strings. Based on these properties, we present the full description of the approximate string matching algorithm in section 3.2.2.

3.2.1 Properties of Approximate Matches

Lemma 23 (Triangle Inequality for Strings) *Let S, R and Q be strings, and let the minimum number of differences in a match between S and R be l and the minimum number of differences in a match between S and Q be k. Suppose $k \leq l$. Then n, the minimum number of differences in a match between Q and R is at least $l - k$, and at most $l + k$.*

Proof: If $n < l - k$, then one can map R to Q with n differences and then map Q to S with k differences, obtaining a composite mapping of R to S with $n + k < l$ differences: A contradiction. If $n > l + k$, a similar contradiction can be reached. □

Definition 2 (Left-Justified and Right-Justified Mappings) *Let Q and R be two strings. A mapping of Q and R is left-justified, if for each character c (of Q and R) which maps to NULL, the following holds. Consider the leftmost character d (of the same string) to the right of c which does not map to NULL. The value of the character that d maps to must be different than c.*

Similarly, a mapping of Q and R is right-justified, if for each character c (of Q and R) which maps to NULL, the rightmost character d (of the same string) to the left of c which does not map to NULL, maps to a character different than c.

Example 1 *Let $Q = ab$, and $R = aab$. The following mapping between Q and R is left-justified: $q_1 \rightarrow r_1, NULL \rightarrow r_2, q_2 \rightarrow r_3$. In contrast, the following mapping is NOT left-justified: $NULL \rightarrow r_1, q_1 \rightarrow r_2, q_2 \rightarrow r_3$.*

Lemma 24 (Existence of Left-Justified and Right-Justified Mappings) *If for two given strings R and Q, the minimum number of differences in a match between Q and R is m, then there exists a left-justified and a right-justified mapping between R and Q with m differences.*

Proof: Given any mapping between R and Q with m differences, we inductively derive a left-justified mapping with m differences as follows. Let c be the leftmost character (in either Q or R) which is mapped to NULL in the original mapping. Without loss of generality, we can assume that c belongs to Q. Consider d, the leftmost character in Q to the right of c which is not mapped to the NULL character. If the character mapped to d is identical to c, then map it to c, and map d to the NULL character. The total number of differences in this new mapping would still be m. Continue with re-mapping characters inductively, starting from d. The derivation of a right-justified mapping can be done similarly. □

Lemma 25 (Non-transitive Equality) *Let S, Q, and R be strings, and suppose that n, k, and l, respectively, are the minimum number of differences in a match between Q and R, R and S, and Q and S. If $k + l > n$, then for any left-justified mapping M_1 of Q and R, with n differences, there exist (i) a left-justified mapping M_2 of Q and S, and (ii) a left-justified mapping M_3 of S and R, such that:*
(1) The composition of M_2 and M_3 gives M_1.
(2) There exists a character c of Q such that, if M_2 maps c maps to e, and M_3 maps e to d (hence M_1 maps c to d), then e is different than both c and d.

Proof: We construct the mappings M_2 and M_3 and designate the character c using the following inductive process.

Let $Q_{1,\iota-1}$ denote the longest prefix of Q which M_1 maps to an identical prefix $R_{1,\iota-1}$. Let $S_{1,\chi-1}$ denote a prefix of S which matches $Q_{1,\iota-1}$ with the smallest number of differences $d(\iota-1)$, and let M_4 be a left-justified mapping of $Q_{1,\iota-1}$ to $S_{1,\chi-1}$ with $d(\iota-1)$ differences. Construct M_2 in a way that $Q_{1,\iota-1}$ maps $S_{1,\chi-1}$ identical to M_4. Similarly, construct M_3 in a way that $R_{1,\iota-1}$ maps $S_{1,\chi-1}$ identical to M_4.

If $d(\iota-1) > 0$, then assign $c = q_\phi$, where q_ϕ is the leftmost character in Q which is not mapped to an identical character in S. If however $d(\iota-1) = 0$, we have two cases.

1. If the following two conditions apply: (i) M_1 maps q_ι to r_ι, and (ii) s_χ is different from both q_ι and r_ι;
   then assign $c = q_\iota$ and $d = r_\iota$.
2. If case 1 does not apply, then we defer the the selection of c and d, as can be seen next.

Continue the construction of M_2 and M_3. We have two cases:

1. If q_ι maps to NULL in M_1, and r_ι is identical to s_χ, then: (i) map q_ι to NULL in M_2, (ii) map $q_{\iota+1}$ to s_χ in M_2, and (iii) map r_ι to s_χ in M_3. Notice that as M_1 is left justified, r_ι can not be identical to q_ι for this case, hence both M_2 and M_3 will stay left justified.
2. If case 1 does not apply, then: (i) map q_ι to s_χ in M_2, and (ii) map s_χ to the character M_1 maps q_ι to in M_3. Obviously both M_2 and M_3 will stay left justified.

Apply the procedure which was applied to Q, R and S above, to the strings $Q_{\iota+1,q}$, the suffix of R which M_1 maps to $Q_{\iota+1,q}$, and $S_{\chi+2,s}$ respectively. Assign a character to c only if it has not been assigned to any characters previously. If at the end of the implied inductive process no values are assigned to c and d, then obviously $k + l \leq m$; a contradiction.

□

Definition 3 (Anchor Characters) *Let Q be a string which includes at least one replicative core, $Q_{i,j}$. Let D be the period of $Q_{i,j}$; hence $Q_{i,j} = D^{(j-i+1)/d}D^{-}$ for some D^{-}. Denote by E the string $D^{(q+m)/d}$. Consider all left-justified mappings of the prefixes of $Q_{i,q}$ and E with less than or equal to m differences. A character c of $Q_{i,q}$ is defined to be a* right anchor character *of Q with respect to $Q_{i,j}$, if in at least one of these left-justified mappings, c is among the leftmost m characters of $Q_{i,q}$ that are not identical to the character they map. Similarly, consider all right-justified mappings of the prefixes of $Q_{1,j}$ and E with less than or equal to m differences. A character c of $Q_{1,j}$ is defined to be a* left anchor character *of Q with respect to $Q_{i,j}$, if in at least one of these right-justified mappings, c is among the rightmost m characters of $Q_{1,j}$ that are not identical to the character they maps.*

Example 2 *Let $m = 1$. Consider the string $Q = baaaaaaaaba$. The substring $Q_{2,9} = aaaaaaaa$ is a replicative core with period $D = a$, hence the string E can be obtained as $E = aaaaaaaaaa$. The following mapping of $Q_{2,11} = aaaaaaaaba$ to $E_{1,9}$ is a left justified mapping with $m = 1$ difference: $Q_{2,9} \to E_{1,8}$, $q_{10} \to NULL$, $q_{11} \to e_9$. Therefore, $q_{10} = b$ is a right anchor character.*

Lemma 26 (Number of Anchor Characters) *The total number of left anchor characters and right anchor characters in a given string Q with respect to any of its replicative cores is at most $4m^2$.*

Proof: Given a replicative core $Q_{i,j}$, consider the execution of the ASI procedure for computing all mappings between $Q_{i,q}$ and any of the prefixes of E with less than m differences. Also consider the execution of the ASI procedure for computing all mappings between $Q^{\leftarrow}_{1,j}$ and $E^{\leftarrow}$ with less than m differences. ASI computes the entries of the array D, for $2m$ of its "diagonals", as long as the entries are $\leq m$. In each of these diagonals, there are at most m locations where the number of differences is increased by one. In a left-justified or a right-justified mapping, a character can be mapped to a non-identical character only if it is in a location where the number of differences increase by one. Hence the proof follows. □

3.2.2 Description of the Approximate String Matching Algorithm

Our algorithm works as follows. Again we first compute the cores of P and T. For each level-h core $P_{i,j}$ of P, we do the following depending on whether $P_{i,j}$ is a replicative core.

$P_{i,j}$ **is non-replicative:** We first map all cores of T, that are identical to $P_{i,j}$. We extend each such initial mapping towards left and right by using the ASI procedure as in the algorithm for the no-local-periodicity case. We output every match.

$P_{i,j}$ **is replicative:** We first find all replicative cores of T whose character and level entries are identical to that of $P_{i,j}$. Let $T_{k,l}$ be such a core and let G be the period of $T_{k,l}$ (and hence $P_{i,j}$). Suppose $P_{i,j} = G^z G^-$, and $T^{i,k} = G^w G^{-\prime}$ for some w, z, G^-, and $G^{-\prime}$. We seek initial mappings between P and T as follows. We have two cases depending on whether $z \leq m$:

$z \le m$: We map each occurrence of G in $P_{i,j}$ to every occurrence of G in $T_{k,l}$. For every initial mapping, we apply the ASI procedure to obtain full mapping, as in the algorithm for the no-local-periodicity case. We output every match.

$z > m$: This case is handled in two independent stages:

1. **Non-Transitive Mappings:** We determine the anchor characters of P with respect to $P_{i,j}$ and the anchor characters of $T_{l-j-m,k+p-i+m}$ with respect to $T_{k,l}$. We map each anchor character of $P_{i,p}$ to each anchor character of $T_{k,k+p-i+m}$. Similarly, we map each anchor character of $P_{1,j}$ to each anchor character of $T_{l-j-m,l}$. For every initial mapping, we apply the ASI procedure to obtain full mapping, as in the algorithm for the no-local-periodicity case. We output every match.
2. **Transitive Mappings:** We output all substrings S of T which can be mapped to P such that no anchor character of P is mapped to any of the anchor characters of S. This is done as follows.
   We first give some definitions and illustrate them in figure 5. Let $E = G^{(p+m)/g}$ and denote by δ_r the minimum number of differences in a match between $P_{i,p}$ and any of the prefixes of E. Denote by e_r the length of the shortest prefix of E which matches $P_{i,p}$ with at most δ_r differences. Similarly denote by δ_l the minimum number of differences in a match between $P_{1,j}$ and any of the prefixes of E. Denote by e_l the length of the shortest prefix of E which matches $P_{1,j}$ with at most δ_l differences. Let δ denote $\delta_r + \delta_l$.
   For any given $n \in \{0, 1, \ldots, m - \delta + 1\}$, let $f_r(n)$ denote the length of the longest prefix of E which matches any prefix of $T_{k,t}$ with at most n differences. Also let $S_r(n)$ denote the longest prefix of $T_{k,t}$ which matches the $f_r(n)^{th}$ prefix of E with at most n differences. Similarly let $f_l(n)$ denote the length of the longest prefix of E which matches any prefix of $T_{1,l}$ with at most n differences. Also let $S_l(n)$ denote the longest prefix of $T_{k,t}$ which matches the $f_l(n)^{th}$ suffix of E with at most n differences.
   For all $n \in \{0, 1, \ldots, m - \delta + 1\}$, we compute the integer $x(n)$ for which the following inequality is satisfied: $g.x(n) + e_r < f_r(m - \delta - n) \le g.(x(n)+1) + e_r$ (there can be at most one such $x(n)$). For each n we consider all $n' \in \{0, 1, \ldots, m - \delta - n\}$. Denote by γ, $m - \delta - n - n'$. We compute the integer $y(n')$ for which the following inequality is satisfied: $g.y(n') + e_l < f_l(\gamma) \le g.(y(n') + 1) + e_l$ (there can be at most one such $y(n)$). Then for all τ such that $\max(0, z - x(n) - y(n') - w) \le \tau < z - x(n) - y(n' - 1) - w$, we output the locations $k - i - g.(y(n') - \tau) + \xi$ for all $\xi \in \{-\gamma, \ldots, +\gamma\}$.

Lemma 27 (Correctness of Approximate Matching Algorithm) *A substring S of T matches P with less than m differences if and only if it is computed by the algorithm.*

Proof: Suppose that a substring S of T maps to P with $\le m$ differences. Then, by lemma 22, for any mapping M_1 of S to P with $\le m$ differences, there exists a level-h core of P denoted C, and a level-h core of S, denoted D such that one of the following conditions

apply:
(a) C and D are identical, and M_1 maps C to D. (b) C and D are replicative cores with identical level and character entries, and C has a substring of size $\geq 2^h$, which maps to an identical substring of D (by M_1), and which includes at least one period of C.

Suppose case (a) applies. The algorithm first maps each level-h core of P to each identical level-h core of T. Then it computes if each such initial mapping extends to a full mapping with $\leq m$ differences. Hence we are guaranteed that the algorithm will map C to D and then will compute the full mapping between S and P.

Suppose case (b) applies. Let $C = G^z G^-$, where G is the period of C, and let $D = G^w G^{-\prime}$, and let $E = G^{(p+m)/g}$, for some z, w, G^-, and $G^{-\prime}$ Also let δ_l be the minimum number of differences in a match between the prefix of P to the left of C and any suffix of the string E. Similarly let δ_r be the minimum number of differences in a match between the suffix of P to the right of C and any prefix of the string E. Let $\delta = \delta_l + \delta_r$.

We again have two cases depending on whether $z \leq m$.

Suppose $z \leq m$. For each level-h replicative core of P, such as C, the algorithm finds all cores of T which have identical character and level entries, such as D. Then it maps each occurrence of G in C to each occurrence of G in D. Hence we are guaranteed that it would find an initial mapping and will extend it to compute desired match.

Suppose $z > m$. Also let σ_l be the minimum number of differences in a match between the prefix of S to the left of D and any of the suffixes of string E. Similarly let σ_r be the minimum number of differences in a match between the suffix of S to the right of D and any of the prefixes of string E. Let $\sigma = \sigma_l + \sigma_r$. We can have two cases depending on whether $\sigma + \delta \leq m$.

Suppose $\sigma + \delta \leq m$. By the execution of the stage for the transitive mappings, we are guaranteed that the algorithm will output S as a match.

Suppose $\sigma + \delta > m$. By lemma 25, in any mapping of S and P with $\leq m$ differences, there exists an anchor character of S which is mapped to an anchor character of P. The algorithm maps each anchor character of P to each anchor character of S and extends this initial mapping to a full mapping, hence it compute the match between S and P.

Suppose that a substring S of T is output by the algorithm. Then either it is output as a guaranteed match, or it is output after extending an initial mapping. If S is output as a guaranteed match, then by lemma 23 it must match P with $leqm$ differences. If S is output after a match between the compact representations of S and P is computed, then by lemma 20 it must match P. □

Running Time of Approximate String Matching Algorithm: As there are at most $t/2^h$ cores in level-h of the fat-tree of T, there can be at most $mt/2^h = mt/2^{\log_3(p/m2(\log^* t+3))} = mt(m2(\log^* t+3)/p)^{1/\log 3}$ cores of T which are considered for deriving initial mappings. By Lemma 26, the number of the anchor characters can be at most m^2 for P or any substring of T it matches. Hence the number of possible initial matches can be at most $m^5 t(m2(\log^* t + 3)/p)^{1/\log 3}$. The ASI procedure runs in $O(m^2)$ time. Therefore the total running time of the algorithm is $O(tm^{8+1/\log 3}(\log^* t/p)^{1/\log 3})$, which is linear in t for $m = O((p/\log^* t)^{1/(1+8\log 3)})$.

References

[AILSV88] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber and U. Vishkin, Parallel Construction of a Suffix Tree with Applications, *Algorithmica*, 3, (1988):347-365.

[BM77] R. Boyer and J. Moore, A Fast String Searching Algorithm, *Communications of the ACM*, 20, (1977):762-772.

[CL90] W. Chang and E. Lawler, Approximate String Matching in Sublinear Expected Time, Proceedings of *IEEE Symposium on Foundations of Computer Science*, (1990).

[CV86] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades, Micro and Macro Techniques for Designing Parallel Algorithms, Proceedings of *ACM Symposium on Theory of Computing*, (1986).

[FM96] M. Farach and M. Muthukrishnan, An optimal logarithmic time, randomized parallel string matching algorithm, Proceedings of *International Colloquium on Automata, Languages and Programming* , (1996).

[GG88] Z. Galil and R. Giancarlo, Data Structures and Algorithms for Approximate String Matching, *Journal of Complexity*, 4, (1988):33-72.

[GP90] Z. Galil and K. Park, An Improved Algorithm for Approximate String Matching, *SIAM Journal on Computing*, 19, (1990):989-999.

[Ha94] R. Hariharan, Optimal Parallel Suffix Tree Construction, Proceedings of *ACM Symposium on Theory of Computing*, (1994).

[Ja90] J. Ja'Ja', Introduction to Parallel Algorithms, *Addison-Wesley*, (1990).

[KMR72] R. Karp, R. Miller and A. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, Proceedings of *ACM Symposium on Theory of Computing*, (1972).

[KMP77] D. Knuth, J. Morris and V. Pratt, Fast Pattern Matching in Strings, *SIAM Journal on Computing*, 6, (1977):323-350.

[LV85] G. Landau and U. Vishkin, Efficient String Matching in the Presence of Errors, Proceedings of *IEEE Symposium on Foundations of Computer Science*, (1985).

[LV86] G. Landau and U. Vishkin, Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm, Proceedings of *ACM Symposium on Theory of Computing*, (1986).

[LV88] G. Landau and U. Vishkin, Fast String Matching with k Differences, *Journal of Computer and System Sciences*, 37, (1988):63-78.

[LV89] G. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, 10, (1989):158-169.

[MM90] U. Manber and G. Myers, Suffix Arrays: A New Method for On-Line String Searches, *SIAM Journal on Computing*, 22, (1993):935-948.

[Mc76] E. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *Journal of the ACM*, 23, (1976):262-272.

[SV94] S. C. Sahinalp and U. Vishkin, Symmetry Breaking for Suffix Tree Construction, Proceedings of*ACM Symposium on Theory of Computing*, (1994).

[SV96] S. C. Sahinalp and U. Vishkin, Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm, Proceedings of*IEEE Symposium on Foundations of Computer Science*, (1996).

[Se80] P. Sellers, The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, 1, (1980):359-373.

[Uk83] E. Ukkonen, On Approximate String Matching. Proceedings of *Conference on Foundations of Computation Theory*, (1983).

[We73] P. Weiner, Linear Pattern Matching Algorithms, Proceedings of *IEEE Symposium on Foundations of Computer Science*, (1973).

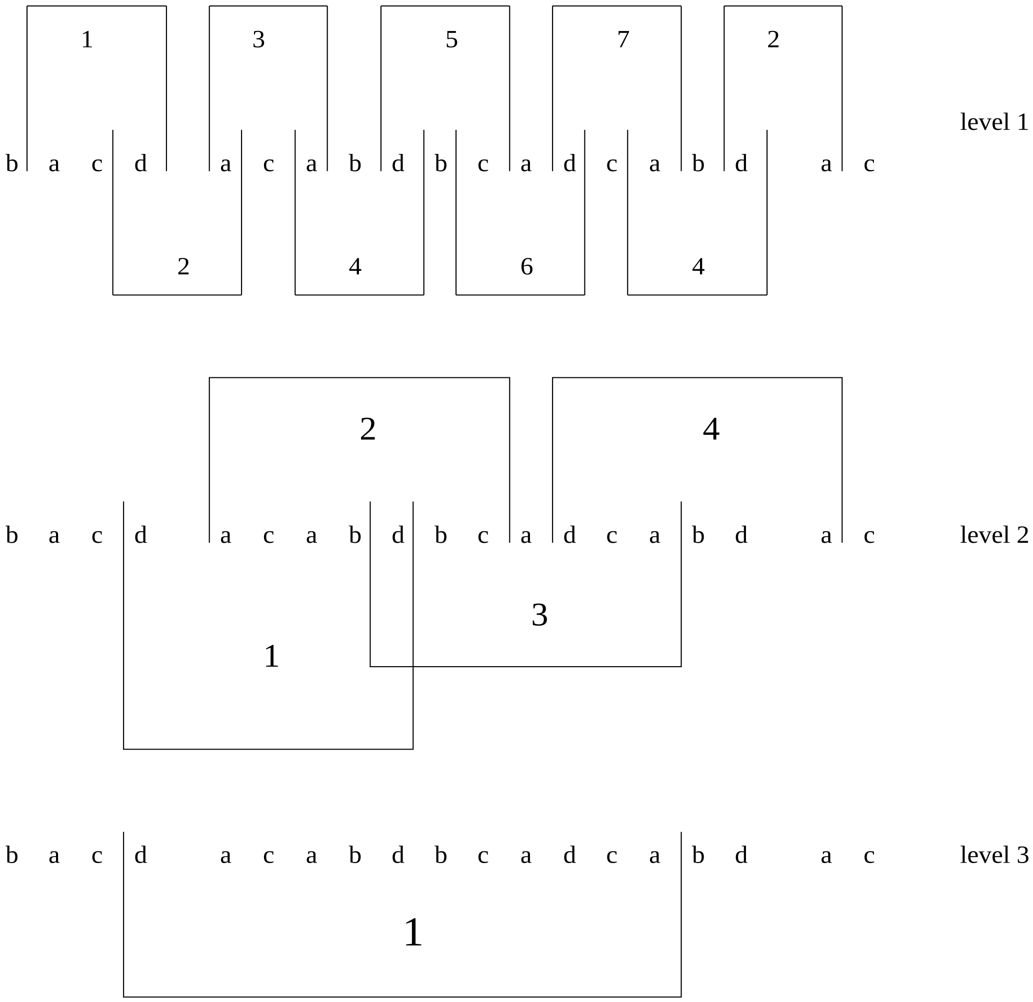

Figure 1: The cores of a given string $S = bacdacabdbcadcabdac$. The level-0 cores are the characters themselves. The cores of levels 1, 2 and 3 are demonstrated. As an example, consider core 1 of level-2. It consists of cores 234 of level-1 and is composed in the following manner: Core 2 contributes da. Then, out of aca of core 3, the first a is excluded and core 3 contributes ca. Finally out of abd of core 1 a is excluded and core 4 contributes bd. This figure is for demonstration purpose of the noticeability and number properties only and is misleading otherwise: the cores computed by LCP cannot be of length 2 or 3. In fact they are never shorter than 9.

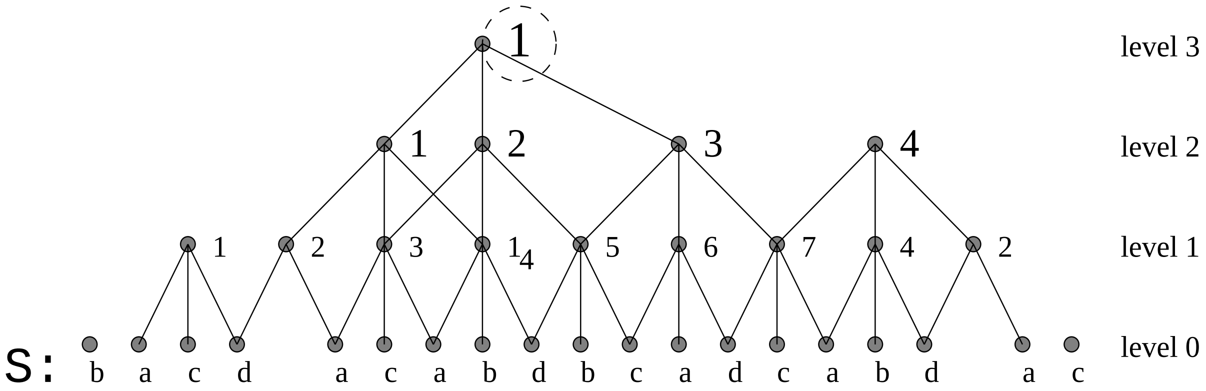

Figure 2: The fat-tree of an example string $S = bacdacabdbcadcabdac$ and its cores. LCP procedure is not demonstrated. The main core is marked with a surrounding circle. We represent the labels of increasing levels with larger characters.

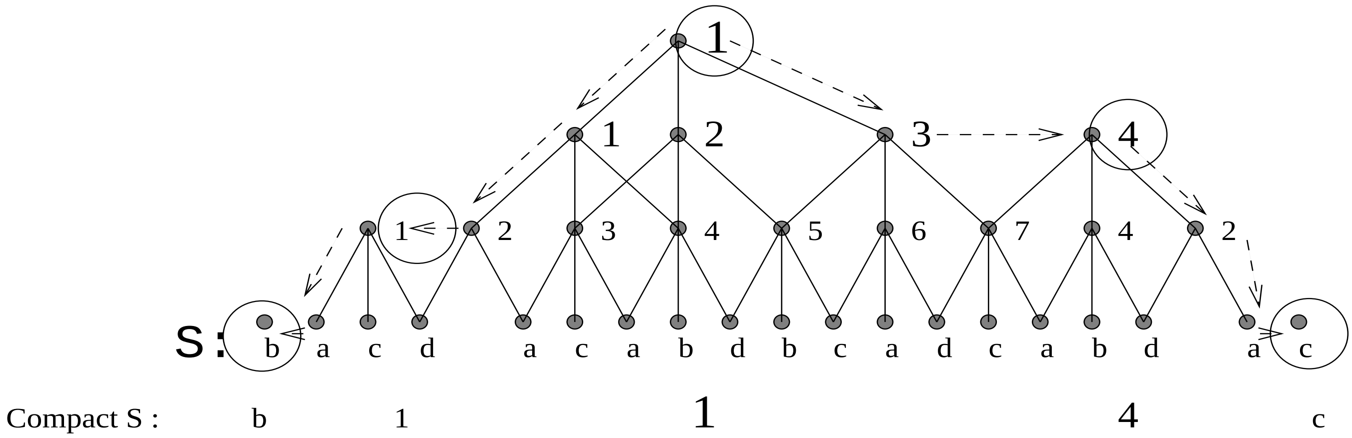

Figure 3: The compact representation of the full string S is the string of core labels: b 1 **1** 4 c. It is obtained via the traversal of the fat-tree of S. Notice that, among all the traversed nodes, only those whose labels are surrounded by a circle are of interest to us.

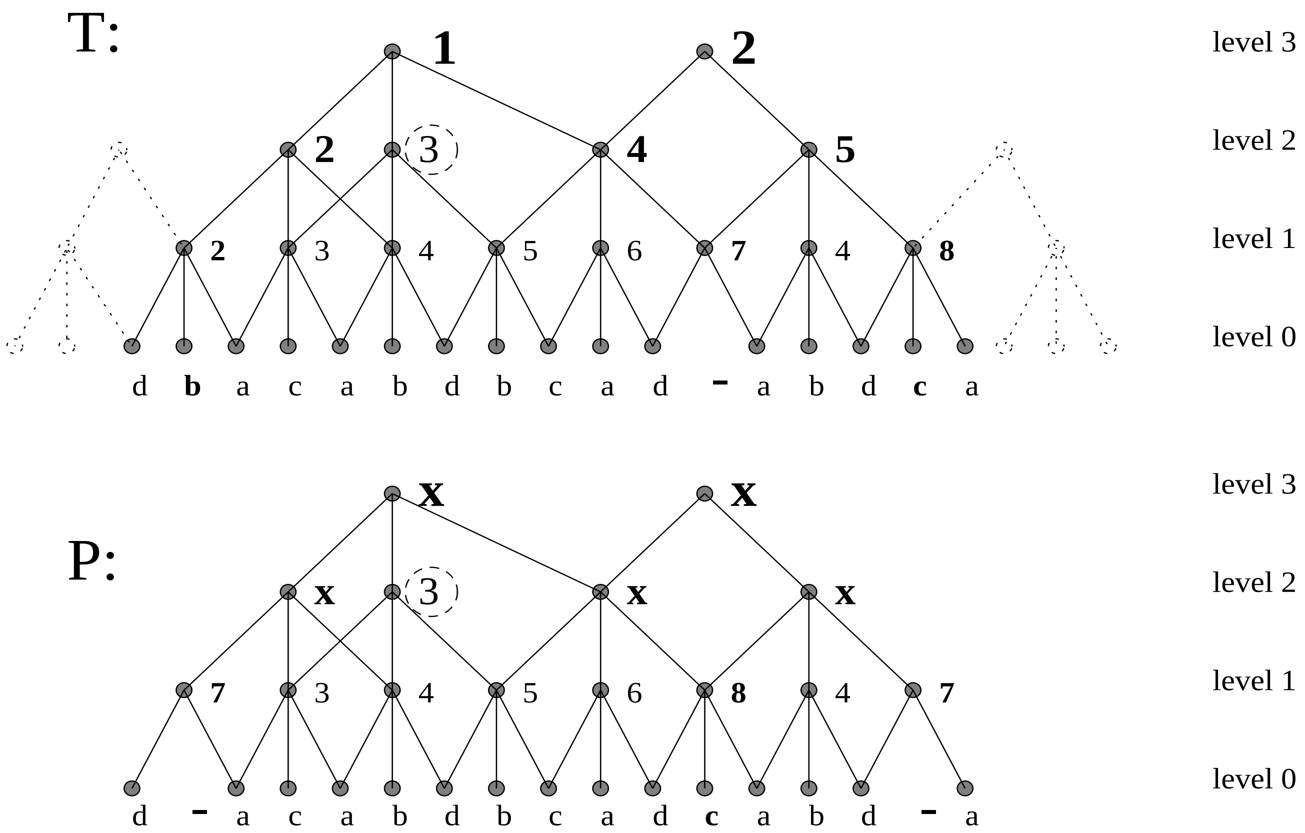

Figure 4: We demonstrate how the fat-trees of P and a substring of T which matches P with m differences look like. Notice that the cores in P which do not have identical cores in T are marked with **X**. A mapping with m differences between P and this substring can be obtained by (1) mapping the cores marked with circles, and (2) extending this initial mapping by the use of the ASI procedure.

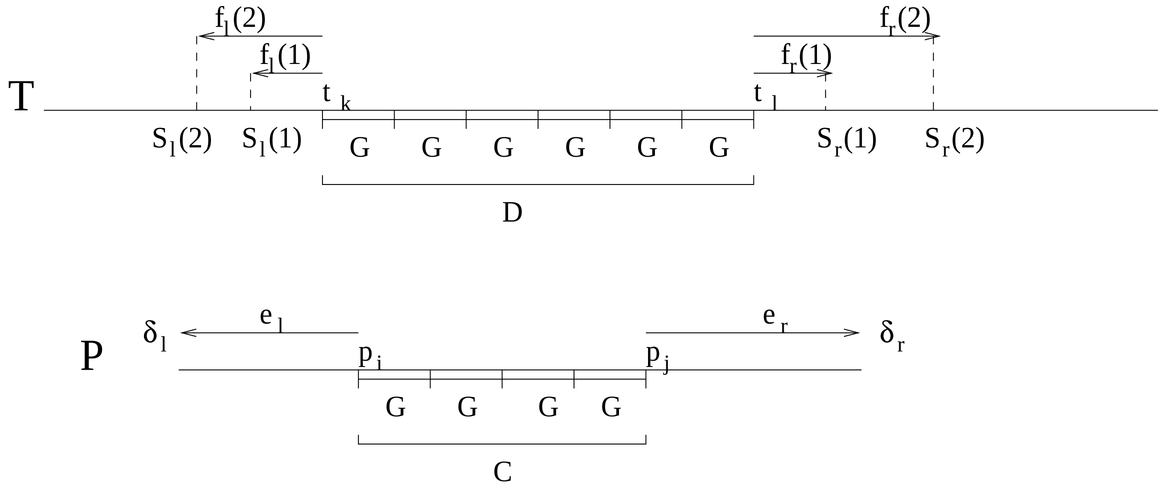

Figure 5: Illustration of definitions for non-transitive mappings.