External Inverse Pattern Matching

Leszek Gąsieniec * Piotr Indyk † Piotr Krysta ‡

Abstract

We consider *external inverse pattern matching* problem. Given a text $\mathcal{T}$ of length n over an ordered alphabet Σ, such that $|\Sigma| = \sigma$, and a number $m \leq n$. The entire problem is to find a pattern $\tilde{\mathcal{P}}_{MAX} \in \Sigma^m$ which is not a subword of $\mathcal{T}$ and which maximizes the sum of Hamming distances between $\tilde{\mathcal{P}}_{MAX}$ and all subwords of $\mathcal{T}$ of length m. We present optimal $O(n \log \sigma)$-time algorithm for the external inverse pattern matching problem which substantially improves the only known polynomial $O(nm \log \sigma)$-time algorithm introduced in [3]. Moreover we discuss a fast parallel implementation of our algorithm on the CREW PRAM model.

Topics: algorithms and data structures, string algorithms, parallel algorithms.

*Max-Planck Institut für Informatik, Im Stadtwald D–66123 Saarbrücken, Germany. E-mail: leszek@mpi-sb.mpg.de

†Computer Science Department, Stanford University, Gates Building, CA–94305, USA. E-mail: indyk@cs.stanford.edu

‡Institute of Computer Science, University of Wrocław, Przesmyckiego 20, PL–51151, Wrocław, Poland. E-mail: pkrysta@ii.uni.wroc.pl

Research of this author was partially supported by KBN grant 2 P301 034 07.

1 Introduction

Given a string $\mathcal{T}$ (called later a text) of length n over an alphabet Σ. The *inverse pattern matching problem* is to find a word $\mathcal{P}_{MIN} \in \Sigma^m$ (or $\mathcal{P}_{MAX} \in \Sigma^m$) which minimizes (maximizes) the sum of Hamming distances [10] between $\mathcal{P}_{MIN}$ ($\mathcal{P}_{MAX}$) and all subwords of length m in the text $\mathcal{T}$. One can also consider two variations of the problem when the optimal word is supposed to occur in the text $\mathcal{T}$ or oppositely when its occurrence in $\mathcal{T}$ is forbidden. The two variations of the problem are called respectively *internal* and *external* inverse pattern matching problems. It is assumed that in the internal inverse pattern matching desired internal pattern $\tilde{\mathcal{P}}_{MIN}$ must minimize the sum of distances, whereas in the external case optimal external pattern $\tilde{\mathcal{P}}_{MAX}$ maximizes the entire sum. As reported in [3] the inverse pattern matching appears naturally and finds applications in several fields like: information retrieval, data compression, computer security and molecular biology. For example, the external inverse pattern matching can be used in a context of intrusion or plagiarism detection (see [14]), or in the synthesis of molecular probes in genome sequencing by hybridization [2].

It was shown by Amir, Apostolico and Lewenstein in [3] that the inverse pattern matching problem can be solved in time $O(n \log \sigma)$ when no additional restriction on $\mathcal{P}_{MAX}$ (or $\mathcal{P}_{MIN}$) is assumed. However it turned out that *internal* inverse pattern matching problem appears to be significantly harder. Amir et al. in [3] presented two algorithms for this problem. The first algorithm, which is reasonably simple, has the running time $O(nm \log \sigma)$. The second one uses more sophisticated techniques (like convolutions [6]) for Hamming distance computation [1] and it runs in time $O(n\sqrt{m}\log^2 m)$. Amir et al. have shown a reduction from *all mismatches problem* (see [1]) to the internal inverse pattern matching. Any improvement in the all mismatches issue is a long standing open problem, thus it looks to be quite unlikely to get a faster algorithm for the internal inverse pattern matching. However Amir et al. show that using techniques from [12] one can get faster superlinear solution for the internal case when approximate answers are allowed.

The best known (to our knowledge) $O(nm \log \sigma)$-time algorithm for the *external* inverse pattern matching was given by Amir et al. in [3]. They presented the idea of m-stems for the text $\mathcal{T}$, i.e. all possible words of length at most m not belonging to $\mathcal{T}$ but whose all proper prefixes form subwords of $\mathcal{T}$. It was shown in [3] that the optimal external pattern $\tilde{\mathcal{P}}_{MAX}$ can be composed of some m-stem of $\mathcal{T}$ extended by a proper size suffix of the maximal word $\mathcal{P}_{MAX}$. Unfortunately the straightforward application of the m-stem approach leads to $O(nm \log \sigma)$-time solution because one has to look for m-stems testing all text subwords of length at most m. In this paper we show how to perform tests of text subwords more efficiently. We present *new* and *optimal* $O(n \log \sigma)$-time algorithm for the external inverse pattern matching problem, showing that the internal case is a bottleneck in the inverse pattern matching. The optimality comes from the complexity of element distinction problem to which external inverse pattern matching can be reduced. The new efficient solution is a consequence of deeper analysis of relation between the maximal words $\mathcal{P}_{MAX}$, $\tilde{\mathcal{P}}_{MAX}$ and the text $\mathcal{T}$. Our main algorithm uses efficient algorithmic techniques like: compact suffix trees [16], range minimum queries [8] and lowest common ancestor queries in trees [11] supported by an on-line computations of symbol weights (defined later).

The rest of the paper is organized as follows. In section 2 we introduce notation and basic techniques used in our algorithm. Section 3 contains the main algorithm with complexity analysis and proof of correctness. In this section we also discuss a parallel implementation of our algorithm. Section 4 contains the final remarks and states some open problems in the related areas.

2 Preliminaries

Given an ordered *alphabet* Σ containing σ symbols, i.e. $|\Sigma| = \sigma$. Any sequence of concatenated symbols from Σ is called a *word* or a *string*. We use symbol $\cdot$ to denote operation of concatenation, but the symbol is omitted in cases where the use of concatenation is natural. We use a notation $w[i]$ for the i^{th} symbol of the word w, $w[i..j]$ for the substring of w which starts at position i and ends at j, ^{-}w stands for the string w without its first symbol, while symbol ε stands for the empty string. For example, let $w \in \Sigma^*$ be a string of length n, i.e. $|w| = n$. Then $w = w[1..n]$, $^{-}w = w[2..n]$, $w[i,i] = w[i]$ and $w[i,j] = \varepsilon$ when $i > j$. Any subword of w of the form $w[1..i]$, for all $i \in \{1, \ldots, n\}$, is called a *prefix* of w and a subword of the form $w[j..n]$, for all $j \in \{1, \ldots, n\}$, is called a *suffix* of w. We use notation $u \in w$ ($u \notin w$) when u is (not) a subword of string w. In case $u \notin w$ we say that the word u is external string for w. A search tree for a given set of words $S \subseteq \Sigma^*$ whose edges are labeled by symbols drawn from the alphabet Σ is called a *trie* [7] for S. Any sequence $v = v_1, \ldots, v_k$ of neighboring nodes (by parent-children relation) in a tree, such that all v_is are pairwise disjoint, is called a *path*. Each word $w \in S$ is represented in the trie as a path from the root to some leaf. Recall that a trie is a prefix tree, i.e. two words have a common path from the root as long as they have the same prefix. A path $v = v_1, \ldots, v_k$, whose all internal nodes but last have degree equal to 1, is called a *chain*.

2.1 Problem Definition

Let $\mathcal{T}$ be a text over Σ such that $|\mathcal{T}| = n$, and $\mathcal{P}$ be a string over Σ such that $|\mathcal{P}| = m \leq n$. The *Hamming distance* [10] between the word $\mathcal{P}$ and a subword of $\mathcal{T}$ starting at position i is defined as follows:

$$H(\mathcal{P}, \mathcal{T}[i..i + |\mathcal{P}| - 1]) = \sum_{j=1}^{|\mathcal{P}|} h(\mathcal{P}[j], \mathcal{T}[i + j - 1]),$$

where for any two symbols $a, b \in \Sigma$

$$h(a,b) = \begin{cases} 0, & a = b \\ 1, & a \neq b. \end{cases}$$

In other words the Hamming distance $H(\mathcal{P}, \mathcal{T}[i..i+|\mathcal{P}|-1])$ gives the number of mismatches between symbols of the aligned words $\mathcal{T}[i..i+|\mathcal{P}|-1]$ and $\mathcal{P}$. In this paper we are primarily interested in the *total Hamming distance* between the word $\mathcal{P}$ and all its alignments in the text $\mathcal{T}$, which is defined as:

$$\mathcal{H}(\mathcal{P}, \mathcal{T}) = \sum_{i=1}^{|\mathcal{T}|-|\mathcal{P}|+1} H(\mathcal{P}, \mathcal{T}[i..i + |\mathcal{P}| - 1]). \tag{1}$$

Now we are ready to introduce the entire problem:

> **Problem:** *External Inverse Pattern Matching.*
>
> Given a text string $\mathcal{T} \in \Sigma^n$, where $|\Sigma| = \sigma$, and a positive integer m, s.t. $m \leq n$. The entire problem is to find a pattern $\tilde{\mathcal{P}}_{MAX} \in \Sigma^m$, s.t. $\tilde{\mathcal{P}}_{MAX} \notin \mathcal{T}$ and $\mathcal{H}(\tilde{\mathcal{P}}_{MAX}, \mathcal{T}) \geq \mathcal{H}(\mathcal{P}, \mathcal{T})$, for all strings $\mathcal{P} \in \Sigma^m$, which do not belong to $\mathcal{T}$.

Notice that if text $\mathcal{T}$ contains all possible strings of length m then the external inverse pattern matching has no solution.

According to the definition of entire problem the i^{th} symbol of the desired pattern $\tilde{\mathcal{P}}_{MAX}$ can be aligned only with positions from i to $n - m + i$ in the text $\mathcal{T}$, since we are interested only in full alignments of the pattern $\tilde{\mathcal{P}}_{MAX}$ in $\mathcal{T}$. The latter observation defines naturally m different ranges in the text $\mathcal{T}$, s.t. the i^{th} range is associated with the i^{th} position in pattern $\tilde{\mathcal{P}}_{MAX}$. We call these m consecutive ranges *windows* $Win_1, \ldots, Win_m$. So simply $Win_i = \mathcal{T}[i..n - m + i]$. Let ϕ_i be a *frequency* function defined in the i^{th} window Win_i, i.e. $\phi_i(a)$ equals to the number of all occurrences of symbol a in Win_i, for all $a \in \Sigma$ and $i \in \{1, \ldots, m\}$. The weight w_i of symbols in the i^{th} window is defined as follows:

$$w_i(a) = (n - m + 1) - \phi_i(a), \quad \text{for all} \quad i \in \{1, \ldots, m\} \text{ and } a \in \Sigma.$$

In terms of the weights the entire problem can be viewed as looking for a string $\mathcal{P} \notin \mathcal{T}$ of length m, which maximizes the sum:

$$\sum_{i=1}^{m} w_i(\mathcal{P}[i]). \tag{2}$$

Notice that the sum (2) is maximized when the i^{th} position in the pattern $\mathcal{P}$ is occupied by the least frequent, or equivalently the heaviest symbol in the window Win_i, which corresponds to the definition of the maximal word $\mathcal{P}_{MAX}$ (the maximal solution of the general inverse pattern matching). However in the external inverse pattern matching optimal pattern $\tilde{\mathcal{P}}_{MAX}$ maximizes the sum (2) among all external strings for the text $\mathcal{T}$.

Through the rest of the paper we will use the weighted version of the external inverse pattern matching. Moreover, before the main algorithm starts, we transform the input string to one which consists of numbers from the range 1 to σ, s.t. every symbol from the alphabet Σ is substituted by a unique number. Thus from now on we assume that symbols can be treated as small numbers. Since the alphabet Σ is ordered, the transformation can be simply performed in time $O(n \log \sigma)$, which does not violate the complexity of the entire algorithm.

2.2 Basic Techniques

In the following section we recall some basic techniques used in our algorithm.

2.2.1 Compact suffix tree and compact trie

A *suffix tree* [16] of a word $w \in \Sigma^*$ is a trie which represents all suffixes of w. Notice that in the worst case the size of the suffix tree can be quadratic in the size of the input string. However, since the suffix tree has exactly $|w| = n$ leaves (corresponding to all suffixes) it can be stored in linear space as follows. Every chain in the suffix tree is represented by a pair of integers (i, j) which refers to the subword $w[i..j]$. There are exactly n leaves in the suffix tree, thus the number of internal nodes of degree greater than one and the number of chains are both not greater than n. The linear representation of a suffix tree is called *compact suffix tree* and it is a known fact [16] that for a word w, such that $|w| = n$, it can be constructed in time $O(n \log \sigma)$. In this paper we consider tries with compact description of chains. Recall that a chain is a path $v = v_1, \ldots, v_k$ whose all nodes but last have degree 1. For our purposes there are stored subwords of only one text in the trie, which means that all the chains in the trie represent substrings of the same text. All the chains are exchanged by edges labeled by pairs of indices describing a position of the corresponding subword in the text. It is important that our definition of the chain implies that each node of the trie of degree ≥ 2 has all outgoing edges of length 1. This means that these edges are labeled by single symbols.

2.2.2 Range minimum search

Given a vector $V = V[1..n]$ of n numbers. A *range query* for a pair (i, j), where $1 \leq i \leq j \leq n$, is a question about minimum among all numbers in the range $V[i..j]$. The main goal in *range minimum search problem* is to preprocess efficiently vector V, such that the range queries can be answered as fast as possible. Gabow et al. [8] gave a linear time preprocessing algorithm for the range minima that results in constant-time query retrieval.

2.2.3 Lowest common ancestor in a tree

Let T be a rooted tree with a root r. For any node $x \in T$ let $branch(x)$ denote a path from the node x to the root r, and $depth(x)$ denote a distance (length of the path) from the node x to the root r. Given two nodes $v, w \in T$. Node v is an *ancestor* of node w iff $v \in branch(w)$. For example the root r is an ancestor of all nodes in T. A *lowest common ancestor* for any pair of nodes $v, w \in T$ is a node $u \in T$ with the greatest possible $depth(u)$, such that $u \in branch(v)$ and $u \in branch(w)$. In [11] there was shown that after a linear preprocessing of a tree T all lowest common ancestor queries can be answered in constant time. Moreover [15] introduced a simpler algorithm with the same sequential time bounds and its parallel counterpart with linear $O(\log n)$-time preprocessing and constant time queries.

3 External Inverse Pattern Matching Algorithm

We start this section recalling known facts about the external inverse pattern matching introduced by Amir et al. in [3]. The following notion of m-stems plays a crucial role in Amir et al. approach, as well as in our algorithm.

Definition 3.1 *Any string $\mathcal{R} = \mathcal{R}[1..l]$ over the alphabet Σ, for $l \in \{2, \ldots, m\}$, is called an m-stem for the text $\mathcal{T} = \mathcal{T}[1..n]$ iff the whole word $\mathcal{R} \notin \mathcal{T}[1..n-m+l]$, but $\mathcal{R}[1..l-1] \in \mathcal{T}[1..n-m+l-1]$.*

Assume that we have already computed the optimal maximal word $\mathcal{P}_{MAX}$ using techniques from [3]. Recall that the cell $\mathcal{P}_{MAX}[i]$ contains the heaviest symbol in the window Win_i. Roughly speaking construction of the word $\mathcal{P}_{MAX}$ can be done as follows. First one has to compute the frequencies of all symbols in the window Win_1, and this can be done in linear time. Since the difference between symbol frequencies in any two neighboring windows is small (only one symbol comes in and only one comes out when we change the window), we can compute the heaviest symbols in the consecutive windows $Win_2, .., Win_m$ allowing constant time for each window. Additionally we compute two arrays $\mathcal{F}_1[1..m]$ and $\mathcal{F}_2[1..m]$, where $\mathcal{F}_1[i]$ contains the second heaviest symbol in the window Win_i, and $\mathcal{F}_2[i]$ contains the difference $w_i(\mathcal{P}_{MAX}[i]) - w_i(\mathcal{F}_1[i])$. The arrays are called *tables of flips* for the pattern $\mathcal{P}_{MAX}$. More detailed description of a data structure, which gives the weights of symbols in the consecutive windows, is given in section 3.1.1.

If $\mathcal{P}_{MAX} \notin \mathcal{T}$ then we take $\mathcal{P}_{MAX}$ as desired pattern $\tilde{\mathcal{P}}_{MAX}$ and the external inverse pattern matching is solved. Otherwise if $\mathcal{P}_{MAX}$ is a subword of $\mathcal{T}$, then the following fact holds:

Fact 3.1 [3] *If $\tilde{\mathcal{P}}_{MAX}$ is the solution of the external inverse pattern matching problem, then $\tilde{\mathcal{P}}_{MAX} = \alpha \cdot \beta$, where α is some m-stem for the text $\mathcal{T}$ and $\beta = \mathcal{P}_{MAX}[|\alpha| + 1..m]$.*

According to the Fact 3.1 searching for the optimal pattern $\tilde{\mathcal{P}}_{MAX}$ has been reduced to testing (weights) of all $O(nm\sigma)$ possible words of the form $\alpha \cdot \beta$. In fact Amir et al. in [3] reduced the number of words for testing to $O(nm)$, skipping over non-reasonable solutions. More precisely they build a trie for all the substrings of $\mathcal{T}$ of length m and they traverse it (node by node) in BFS order, testing a maximal external string leaving the trie at a current node. Since the size of the trie is $O(nm)$, their approach gives an algorithm with running time $O(mn \log \sigma)$. In this paper we show how to search the nodes of the trie more efficiently.

Let v be a node in the trie on depth k. Let $C(v) = \{c_1, \ldots, c_l\}$ be set of children of the node v and $X(v) = \{x_1, \ldots, x_l\}$ be set of symbols on first positions of edges $e_1, \ldots, e_l$ connecting the node v to its children respectively. Let s be a string represented by a path from the root of the trie to v, see Figure 1. Moreover let y be the heaviest symbol in the window Win_{k+1} which is not in $X(v)$, i.e. $w_{k+1}(y) \geq w_{k+1}(z)$, for all $z \in \Sigma \setminus X(v)$. The following lemma shows the advantage of m-stem approach.

Lemma 3.1 *The string $s \cdot y \cdot \mathcal{P}_{MAX}[k+2..m]$ is the heaviest possible external string leaving the trie at the node v.*

Proof: Since the weight of the string s is fixed and the suffix $\mathcal{P}_{MAX}[k+2..m]$ is the heaviest possible extension (see the definition of $\mathcal{P}_{MAX}$), thus the string $s \cdot y \cdot \mathcal{P}_{MAX}[k+2..m]$ is the heaviest possible external string leaving trie at the node v. □

Let $\bar{X}(v) = \bar{x}_1, \ldots, \bar{x}_{l'} \subset X(v)$ be a set of symbols on first positions in the edges $\bar{e}_1, \ldots, \bar{e}_{l'}$ ($\bar{e}_p = e_q$ iff $\bar{x}_p = x_q$), such that $w_{k+1}(\bar{x}_i) \leq w_{k+1}(y)$ for all $i \in \{1, \ldots, l'\}$.

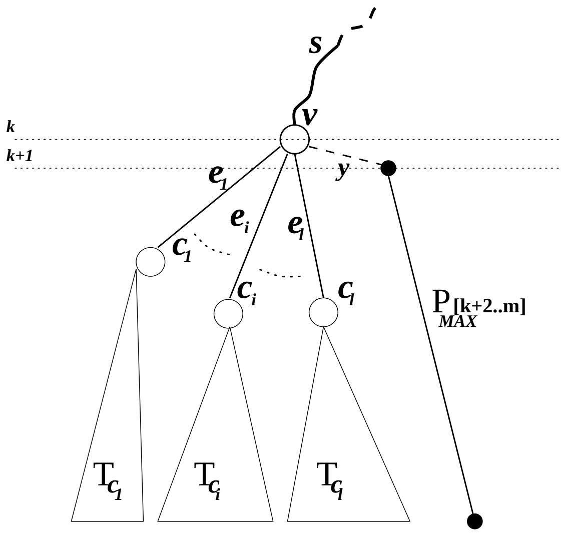

Figure 1: Heaviest external string leaving the trie at the node v.

Lemma 3.2 *Maximal external string $s \cdot y \cdot \mathcal{P}_{MAX}[k+2..m]$ leaving the trie at the node v is heavier than all strings of the form $s \cdot \bar{x}_i \cdot \mathcal{P}_{MAX}[k+2..m]$ for all $i \in \{1, \ldots, l'\}$, and there is no need to test external strings going out of the trie* at *and* below *the edges $\bar{e}_1, \ldots, \bar{e}_{l'}$.*

Proof: Notice that all external strings passing through the node v have the same prefix s. Since the suffix $y \cdot \mathcal{P}_{MAX}[k+2..m]$ is heavier than all possible suffixes starting from symbols of the set $\bar{X}(v)$, the results follows. □

Lemma 3.2 has interesting consequences if the maximal external symbol $y = \mathcal{P}_{MAX}[k+1]$.

Corollary 3.1 *If the maximal external symbol $y = \mathcal{P}_{MAX}[k+1]$, then the string $s \cdot \mathcal{P}_{MAX}[k+1..m]$ is the heaviest possible external string passing through the node v, and there is no need to traverse the trie below the node v.*

The advantage of the Corollary 3.1 becomes more clear when it is used in the context of the nodes of some chain in the trie. The following lemma plays a crucial role in our efficient searching of chains in the trie of the text subwords.

Lemma 3.3 *Let $u = u[1..r]$ be a string which is represented by the only chain ρ_u going out from a node v (v has degree one). If $u[1..r] \neq \mathcal{P}_{MAX}[k+1..k+r]$ then:*

A. *the string $s \cdot \mathcal{P}_{MAX}[k+1..m]$ is the heaviest possible external string passing through the node v and there is no need to search the trie below v, otherwise*

B. *let j be the position in the word $u[1..r]$ for which the corresponding difference in the flip table $\mathcal{F}_2[k+j]$ is the smallest in range $k+1, \ldots, k+r$, then the word $s \cdot \mathcal{P}_{MAX}[k+1..k+j-1] \cdot \mathcal{F}_1[k+j] \cdot \mathcal{P}_{MAX}[k+j+1..m]$ is the heaviest possible external string among all external strings leaving the trie at nodes of the chain ρ_u. In this case a part of the trie below the chain ρ_u is a subject for further search.*

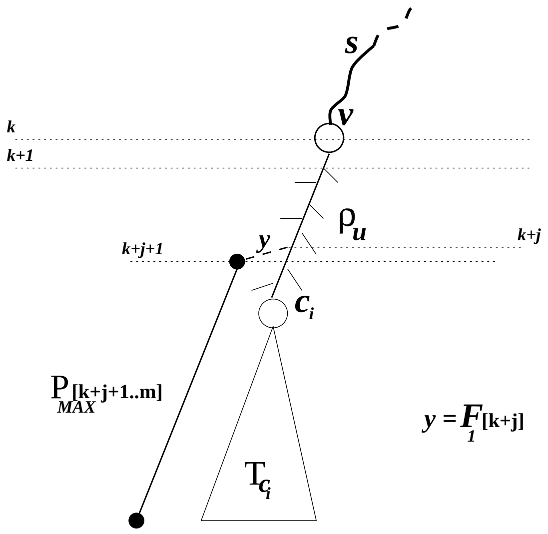

Figure 2: Heaviest external string leaving the trie at the chain ρ_u.

Proof:

ad A. The string $s \cdot \mathcal{P}_{MAX}[k+1..m]$ is an external string for $\mathcal{T}$ and it is the heaviest possible external string which passes through the node v in the trie.

ad B. It is enough to change only one symbol in the word u to create an external string which leaves the trie at some node of the chain ρ_u, see Figure 2 and Fact 3.1. According to the definition of the tables of flips, the position j in u gives the minimal lose of weight among all possible swaps of one symbol in u. Since it is still possible that the maximal external string leaves the trie below the chain ρ_u, the part of the trie hanged below ρ_u is a subject of further search.

□

Now we are ready to present our main algorithm.

3.1 Algorithm

The algorithm consists of two stages. The first one, called *preprocessing*, contains a construction and initialization of all data structures used later during the actual search. The second stage, called *searching phase*, consists of an actual construction of the desired optimal external solution $\tilde{\mathcal{P}}_{MAX}$.

3.1.1 Preprocessing

First of all, we find the maximal pattern $\mathcal{P}_{MAX}$ using techniques from [3]. If $\mathcal{P}_{MAX}$ is an external string for the text $\mathcal{T}$ (which can be checked by any string matching algorithm, e.g. see [13]) then we are done, otherwise instead of the full trie of text subwords we build a compact trie $T_{\mathcal{T}}$. It is reconstructed from a compact suffix tree for the text $\mathcal{T}$ by cutting all deep paths (from the root to leaves) on depth m, and skipping all shallow paths (shorter than m). At every node v of $T_{\mathcal{T}}$ we keep information about the string s (subword of the text $\mathcal{T}$) which is represented by the path from root of the trie to the node v. Additionally we build a common suffix tree T^* for the text $\mathcal{T}$ and the maximal word $\mathcal{P}_{MAX}$ (i.e. a suffix tree for the word $\mathcal{T}\$\mathcal{P}_{MAX}$) and we preprocess it for LCA queries. Construction of all the trees can be done in time $O(n \log \sigma)$ as well as the preprocessing for LCA queries.

An on-line computation of the weights of symbols in the consecutive windows plays a crucial role in the preprocessing and the searching phase. According to the need of the algorithm a data structure which represents the weights of symbols must keep also the current order between the weighted symbols. The data structure is represented by an array $M = M[1..n]$, s.t. the i^{th} cell of the array contains a pointer to a (double-linked) *horizontal* list of all symbols having weight i. Moreover all non-empty cells of the array are connected into a (double-linked) *vertical* list. The non-empty cell in the array M with the largest index (which contains a list of the heaviest symbols) is accessible directly by a variable max. Symbols in the lists are also accessible directly by the *symbol index*. The construction (initialization) of the data structure, which corresponds to Win_1, can be simply done in linear $O(n)$ time, since we assumed that symbols from the alphabet Σ are substituted by unique numbers from the range $1, \ldots, \sigma$. The data structure supports the following three operations.

The *first* operation gives the weight of any symbol $a \in \Sigma$ in the current window. The weight of the symbol a corresponds to the position of a horizontal list containing a in the array M. Since the symbols in the horizontal lists are accessible by the symbol index thus this operation works in constant time.

The *second* operation is needed when the algorithm changes a window from Win_i to Win_{i+1}, for all $i \in \{1, \ldots, n-m\}$. When the window is changed, only two symbols change slightly their weights, i.e. $\mathcal{T}[i]$ comes out and the symbol $\mathcal{T}[n-m+i]$ comes into the window. We find the weight $w_i(\mathcal{T}[i])$ using the symbol index, then we exclude the symbol $\mathcal{T}[i]$ from the list linked at $M[w_i(\mathcal{T}[i])]$, and then the symbol $\mathcal{T}[i]$ is inserted at the beginning of the list linked at $M[w_i(\mathcal{T}[i]) - 1]$. In the meantime the pointers to the neighbors of $M[w_i(\mathcal{T}[i])]$ and $M[w_i(\mathcal{T}[i]) - 1]$ in the vertical list are modified if necessary. Finally the symbol index is decreased by one at the position $\mathcal{T}[i]$. Similar operation is performed when the symbol $\mathcal{T}[n-m+i]$ increases by one its level in the array M. Thus the whole step can be implemented in constant time.

The *third* operation is performed when we look for the heaviest symbol in a window Win_i, not belonging to the given set of symbols $X(v) = \{x_1, .., x_l\} \subset \Sigma$. After a sequence of l deletions in the horizontal lists and at most l deletions in the vertical list, the desired symbol is accessible at $M[max]$. Finally the current structure of symbol weights is restored by the reverse sequence of insertions to the horizontal lists and the vertical list. And the whole step can be implemented in time $O(l)$.

Using the on-line computation of weights we compute in time $O(m)$ the flip tables: $\mathcal{F}_1[1..m]$ which contains the second heaviest symbols in the consecutive windows and $\mathcal{F}_2[1..m]$ whose i^{th} cell contains the difference $w_i(\mathcal{P}_{MAX}[i]) - w_i(\mathcal{F}_1[i])$. The second table $\mathcal{F}_2$ is preprocessed in linear time for the minimum range queries. Finally we compute the weights of all suffixes of the maximal pattern $\mathcal{P}_{MAX}$ and we store them in a table $\mathcal{S}[1..m]$ also in time $O(m)$.

3.1.2 Searching phase

The searching phase consists of two rounds. During the first search of $T_{\mathcal{T}}$ for every node v we compute the heaviest external string, called a *candidate*, which leaves the trie at a node v or at a chain which is placed under the node v. Finally, if there is any candidate, the trie is searched again to find the maximal external pattern $\tilde{\mathcal{P}}_{MAX}$. Otherwise the entire problem has no solution.

During the first round the algorithm traverses the tree $T_{\mathcal{T}}$ in the BFS-like order, s.t. children are inserted into a waiting list according to their depth in the tree. Assume that the algorithm just took from the waiting list a node v of depth k in $T_{\mathcal{T}}$. It is assumed recursively that the weight of a string s, which is represented by a path from the root of $T_{\mathcal{T}}$ to the node v, has been already computed and the on-line weight data structure is currently set to answer queries in the window Win_{k+1}.

If the node v is of degree ≥ 2, all edges coming out of v are labeled by single symbols (definition of the compact trie, see section 2.2.1). We find the maximal external symbol y using the on-line weight data structure. The weight of the word $s \cdot y \cdot \mathcal{P}_{MAX}[k+2..m]$ is clearly composed of weights of: the string s (stored at the node v), the symbol y (described by weight function in the current window) and the suffix of pattern $\mathcal{P}_{MAX}$ (stored in the table $\mathcal{S}$). The weight of the word $s \cdot y \cdot \mathcal{P}_{MAX}[k+2..m]$ is stored at the node v. According to Lemma 3.2 all *light* edges (and corresponding subtrees hanged under them) with symbols lighter than y can be ignored. For the rest of edges we update at their ending nodes information about the weight of a string which is represented by the path coming from the root, to fulfill the recursive assumption. The weight of the string is composed of the weight of the string s (stored at v) and the weight of a symbol placed on the edge (given by the weight function). All nodes below *heavy* edges are inserted into the waiting list on level $k+2$.

If the node v is a first node (its degree is 1) of a chain ρ_u, we check if a string $u = u[1..r]$ represented by the chain symbols, is a subword of the pattern $\mathcal{P}_{MAX}$, i.e. if $u[1..r] = \mathcal{P}_{MAX}[k+1..k+r]$. This can be done by asking for a lowest common ancestor of the proper suffix of $\mathcal{T}$ (string s and its extension in $\mathcal{T}$) and the pattern suffix $\mathcal{P}_{MAX}[k+1..m]$. If the lowest common ancestor for both suffixes is placed on a level $< r$ in T^*, then we know that $u[1..r] \neq \mathcal{P}_{MAX}[k+1..k+r]$ and according to part A of Lemma 3.3 we have only one candidate $s \cdot \mathcal{P}_{MAX}[k+1..m]$, and we do not search the trie below the chain ρ_u. Otherwise, when the equality holds, we recover the candidate from the flip tables $\mathcal{F}_1$ and $\mathcal{F}_2$. First we ask a minimum range query in $\mathcal{F}_2[k+1..k+r]$, getting index of a position j whose change gives the smallest lose of weight, and getting the candidate $s \cdot \mathcal{P}_{MAX}[k+1..j-1] \cdot \mathcal{F}_1[k+j] \cdot \mathcal{P}_{MAX}[k+j+1..m]$. The information about the weight of a string represented by path coming from the root to the node under the chain ρ_u is updated with a help of the table $\mathcal{S}$.

In both cases the time at node v is proportional to degree of the node v, thus searching of the whole trie $T_{\mathcal{T}}$ can be done in time proportional to the size of the trie, i.e. in time $O(n)$. At last the trie $T_{\mathcal{T}}$ is searched again to find the maximal weight, which is the weight of the maximal external pattern $\tilde{\mathcal{P}}_{MAX}$.

Theorem 3.1 *The external inverse pattern matching problem can be solved in optimal time $O(n \log \sigma)$.* □

3.2 Parallel Approach

In this section we discuss shortly a parallel implementation of our external inverse pattern matching algorithm on the CREW PRAM model. Most of the steps in our algorithm can be easily parallelized when we allow for superlinear work and space.

Theorem 3.2 *The external inverse pattern matching algorithm can be implemented in time $O(\log n)$ and work $O(n \log n + m\sigma \log \sigma)$ on the CREW PRAM.*

Proof: Both trees $T_{\mathcal{T}}$ and T^* can be computed in $O(\log n)$-time and $O(n \log n)$ work when subquadratic space is available, see [4]. Moreover the tree T^* can be preprocessed in time $O(\log n)$ and linear work for LCA queries, see [15]. Since we can not use on-line computation of weights in all windows at the same time we have to compute the whole table of weights which is of size $O(m\sigma)$. The table of weights can be easily computed in time $O(\log m)$ and work $O(m\sigma)$ but since we still need to keep order between weights of symbols we have to sort all m columns, by parallel merge-sort [5], which gives total work $O(m\sigma \log \sigma)$. When the table is ready, we compute the pattern $\mathcal{P}_{MAX}$, flip tables $\mathcal{F}_1$, $\mathcal{F}_2$ and the table $\mathcal{S}$ in time $O(\log m)$ and linear work. Then the table $\mathcal{F}_2$ is preprocessed for minimum range queries in logarithmic time and work $O(m \log m)$ (computing minimum in every block of size 2^i, for $i = 1, .., m$). When all data structures are ready we start the searching algorithm. We assume that at every node v of the trie $T_{\mathcal{T}}$ there is a linear number of processors according to the degree of v. We compute in constant time the weights of all *feasible edges*, i.e. the edges of length 1 placed under nodes of degree ≥ 2 (with help of the table of weights) and the edges representing chains which are subwords of $\mathcal{P}_{MAX}$ (using LCA queries and table S). If the path from root of the trie to the node v is composed only of feasible edges, then the node v is called *feasible node*. We use any *Euler tour technique*, see e.g. [9], to compute all feasible nodes and the weights of a strings which are represented by paths from the root. Computation of the feasible nodes is done in time $O(\log n)$ and linear work. Now if a feasible node v (placed on depth k and under a string s) is the first node of a chain, the weight of a candidate is composed from the weight of the string s, table $\mathcal{S}$ and flip tables. If the node v has degree $l \geq 2$, then $O(l)$ processors associated with the node find in logarithmic time the heaviest symbol y in column $k+1$ after deletion of l symbols placed in edges coming out of v. This can be done by testing only $l+1$ heaviest symbols in column $k+1$. In this case the weight of the candidate is composed from the weight of the string s, symbol y and table $\mathcal{S}[1..m]$. When the weights of candidates are ready, we apply any tree contraction algorithm, see e.g. [9], looking for the maximum in the tree representing optimal pattern $\tilde{\mathcal{P}}_{MAX}$. $\square$

4 Conclusion

We have presented a new and optimal $O(n \log \sigma)$-time algorithm for the (sequential) external inverse pattern matching, showing that the internal case is the hardest part of inverse pattern matching. It is an interesting question if there exists a faster algorithm solving internal inverse pattern matching but it looks this question has no simple answer. Another interesting task for further research is to improve the bounds of the external inverse pattern matching in the parallel issue. Notice that if the product of m and σ is small, i.e. $m\sigma = O(n)$, our parallel implementation is fast and efficient. But if we want to keep linear complexity for all feasible values of n, m and σ, one has to pass the bottleneck hidden in the computation of the weights of symbols in every window Win_i. Another interesting question appears when we ask for the complexity of sequential and parallel inverse pattern matching in case of other measures of distances, e.g. the edit distance.

References

[1] K. Abrahamson, Generalized String Matching, *SIAM Journal on Computing*, 16(6):1039-1051, 1987.

[2] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts and J.D. Watson, Molecular Biology of the Cell, Garland Publishing, N.Y., 1989.

[3] Amihood Amir, Alberto Apostolico and Moshe Lewenstein, Inverse Pattern Matching, Manuscript, to appear in *Journal of Algorithms*.

[4] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica*, 3:347–365, 1988.

[5] R. Cole, Parallel merge sort, *SIAM, J. Computing*, 4(1988), pp 770–785.

[6] M.J. Fischer and M.S. Paterson, String matching and other products, *Complexity of Computation*, R.M. Karp (editor), SIAM-AMS Proceedings, 7:113–125, 1974.

[7] E. Fredkin, Trie Memory, *Communications of the ACM*, 3:490–499, 1962.

[8] H.N. Gabow, J.L. Bentley and R.E. Tarjan, Scaling and related techniques for geometry problems, In Proceedings of *16th ACM Symposium on Theory of Computing (STOC)*, pp. 135–143, 1984.

[9] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, *Cambridge University Press*, 1988.

[10] R.W. Hamming, Error detecting and error correcting codes, *Bell. Sys. Tech. Journal*, 26(2):147–160, 1950.

[11] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM Journal on Computing*, 13:338–355, 1984.

[12] H. Karloff, Fast algorithms for approximately counting mismatches, *Information Processing Letters*, 48(2):53–60, 1993.

[13] D.E. Knuth, J.H. Morris, and V.B. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* 6 (1977), 323–350.

[14] D. Russel and G.T. Gangemi, Sr., Computer Security Basics, O'Reilly and Associates, Inc., Sebastopol, California, 1991.

[15] B. Schieber and U. Vishkin, On finding lower common ancestors: simplification and parallelization, In Proceedings of *3rd Aegean Workshop on VLSI Algorithms and Architecture*, LNCS 319:111–123, 1988.

[16] P. Weiner, Linear pattern matching algorithms, In Proceedings of *14th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 1–11, 1973.