Color Set Size Problem with Applications to String Matching *

Lucas Chi Kwong Hui

Computer Science
University of California, Davis
Davis, CA 95616
hui@cs.ucdavis.edu

Abstract. The *Color Set Size* problem is: Given a rooted tree of size n with l leaves colored from 1 to m, $m \leq l$, for each vertex u find the number of different leaf colors in the subtree rooted at u. This problem formulation, together with the *Generalized Suffix Tree* data structure has applications to string matching. This paper gives an optimal sequential solution of the color set size problem and string matching applications including a linear time algorithm for the problem of finding the longest substring common to at least k out of m input strings for all k between 1 and m. In addition, parallel solutions to the above problems are given. These solutions may shed light on problems in computational biology, such as the multiple string alignment problem.

1 Introduction

We consider the following problem: Given m strings, for every pattern P that occurs in any one of the m strings, find the number of strings that contain P. We give an $O(n)$ time (implicit) solution to the above problem (denoted as the *all-patterns* problem) where n is sum of the lengths of the m strings. By implicit, we mean that we do not count the time to output the solution. As n is the size of the input, $O(n)$ time is optimal.

We show that the implicit all-patterns problem can be viewed as a *color-set-size* problem on the Generalized Suffix Tree representing the m strings. The color-set-size (abbrev. CSS) problem on trees will be defined precisely in §2. We give a linear time solution to the CSS problem, which implies a linear time solution (linear in the sum of length of the m strings) to the all-patterns problem. The CSS solution has an underlying structure that is concise and clear, and the running time constant is small. This makes it very practical. We believe that the CSS problem will have other applications in string matching and other areas.

1.1 Applications

In §3 we give a list of string matching problems are variations of the all-patterns problem. All use the all-patterns solution. The problem that motivated us is: Given

* This work was partially supported by NSF Grant CCR 87-22848, and Department of Energy Grants DE-AC03-76SF00098 and DE-FG03-90ER60999.

m strings of total length n, find the longest substring that is common to at least k strings for every k between 1 and m. Our all-patterns solution implies an $O(n+q)$ solution to the above, where q is the total output size.

A special case of the above problem is to find the longest substring that is common to at least k strings for a *fixed* k between 1 and m. We refer this as the *k-out-of-m* problem. Historically, this problem was claimed to be solved by [17] but no details are given. Also the repeat finding algorithm of [14] (which sorts the suffixes in lexicographic ordering) can solve this problem in $O(n \log n)$ expected time. We present an optimal solution to the k-out-of-m problem, which is a special simplified case of the solution to the all-patterns problem. This solution is the first optimal one that appears in literature and is practical as the components (building a Generalized Suffix Tree and solving the CSS problem) are easy to implement.

Multiple string alignment

One possible motivation of the all-patterns problem and its variations is the multiple string (sequence) alignment problem, which is a difficult problem of great value in computational biology. The CSS solution does not solve the multiple string alignment directly, but it may shed light. A simplified description of the multiple string alignment problem follows. Details can be found in [4, 6, 8].

An alignment $\mathcal{A}$ of two strings X and Y is obtained by adding spaces into X and Y and then placing the two resulting strings (with length $= l$) one above the other. The value of $\mathcal{A}$, $V(\mathcal{A})$, is defined as $\sum_{i=1}^{l} s(X(i), Y(i))$, where $s(X(i), Y(i))$ is the value contributed by the two oppositing characters in position i. A simple but common scheme is $s(X(i), Y(i)) = 0$ if $X(i)$ and $Y(i)$ are the same character, and $s(X(i), Y(i)) = 1$ otherwise. An optimal alignment of X and Y is an alignment $\mathcal{A}$ which minimizes $V(\mathcal{A})$.

A multiple alignment of $m > 2$ strings $\mathcal{X} = \{X_1, X_2, ..., X_m\}$ is a natural generalization of the pairwise alignment defined above. Chosen spaces are inserted into each string so the resulting strings have the same length l, and then the strings are arrayed in m rows of l columns each so that each character and space of each string is in a unique column. However, the value of a multiple alignment is not so easily generalized. Several approaches are used, for example the value of a multiple string alignment can be defined as the sum of values of the induced pairwise alignments [4, 6, 8]. Recently, Gusfield [8] introduced two computational efficient multiple alignment methods (for two multiple alignment value functions) whose deviation from the optimal multiple alignment value is guaranteed to be less than a factor of two.

By solving the all-patterns problem of $\mathcal{X}$ in linear time, we get the following information: for every sub-string occurred in $\mathcal{X}$, we know the number of input strings containing this sub-string. This information is certainly helpful to the construction of an optimal multiple alignment, but research effort is needed to figure out the details. One possible approach is to use a 'greedy' approach to construct the multiple alignment. We start working with the sub-strings which occur in most strings in $\mathcal{X}$. Those sub-strings are used as *anchor* points to align the input strings. This may give us an efficient algorithm which is optimal or nearly optimal.

Computer virus detection

Another area of possible application is in computer virus detection. We consider a computer program as a string of machine instructions. The alphabet is the set of

different machine instructions. If a computer system is infected by a computer virus, many computer programs will have the same piece of virus code (again a string of machine instructions) attached to it. Therefore a string of machine instructions which is common to a lot of existing programs is likely to be a virus. We can build a Generalized Suffix Tree for the computer program strings, assign a different color for each program string, and solve the CSS problem. The CSS solution will contain the following information: For every pattern (or piece of code) that occurs in any one of the computer program strings, we know the number of computer programs that contain it [11].

This is certainly useful for the virus detector, and this can be computed in linear time. However, one possible problem is that the alphabet size (size of set of instructions) may be too large for an efficient construction of the Generalized Suffix Tree. Research effort is needed to solve this problem.

This paper is organized as follows: §2 describes the CSS problem and its solution. §3 describes the *Generalized Suffix Tree*, the all-patterns problem, the k-out-of-m problem and other string matching applications. §4 and §5 give parallel algorithms for the CSS and the string matching problems.

2 The Color Set Size problem

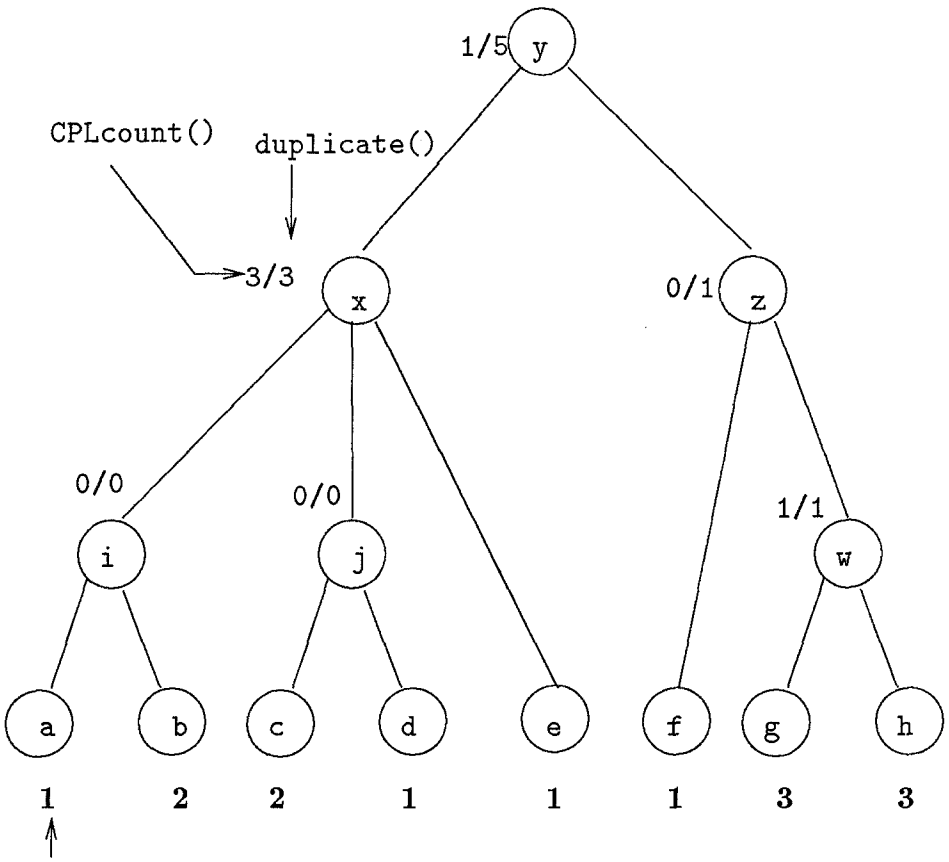
Let $C = \{1, \ldots, m\}$ be a set of colors and T be a general rooted tree with n leaves where each leaf is associated with a color c, $c \in C$. The *color set size* for a vertex v, denoted as $css(v)$, is the number of different leaf colors in the subtree rooted at v. The Color Set Size (CSS) problem is to find the color set size for every internal vertex v in T. For example in Figure 1 $css(x) = 2$ and $css(y) = 3$.

We start by defining the following. Let $LeafList$ be the list of leaves ordered according to a post-order traversal of the input tree T. For v a leaf of T with color c, let $lastleaf(x)$ be the last leaf which precedes x in $LeafList$ and has color c. If no leaf with color c precedes x in $LeafList$, then $lastleaf(x) = Nil$. See Figure 1 for an example. For vertices x and y of T, let $lca(x, y)$ be the least common ancestor of x and y. For an internal vertex u of T let $subtree(u)$ be the subtree of T rooted at u, and $leafcount(u)$ be the number of leaves in $subtree(u)$.

An internal vertex u is a *color-pair-lca* (of color c) if there exists a leaf x of color c, such that $u = lca(lastleaf(x), x)$. For example in Figure 1 vertex w is a color-pair-lca of color 3. Note that a single vertex can be a color-pair-lca for several different colors. For an internal vertex u of T, define $CPLcount(u) = k$ if among all colors, there are k occurrences of leaf pairs for which u is their color-pair-lca ($CPLcount()$ stands for the color-pair-lca counter). Also define $duplicate(u) = \sum_{v \in subtree(u)} CPLcount(v)$ $= CPLcount(u) + \sum_{w \in W} duplicate(w)$ where W is set of children of u.

2.1 Core idea for a linear CSS solution

We want an O(n) solution to the CSS problem. The direct approach is to compute whether a leaf colored c exists in $subtree(u)$ or not, for every internal vertex u and color c. However even if we could answer the above query in constant time it




```
lastleaf(a) = Nil      lastleaf(b) = Nil
lastleaf(c) = b        lastleaf(d) = a
lastleaf(e) = d        lastleaf(f) = e
lastleaf(g) = Nil      lastleaf(h) = g
```

The two numbers associated with each internal vertex are CPLcount() and duplicate() in the format of : CPLcount()/duplicate().

Fig. 1. CSS Problem Example

still takes $O(mn)$ time to compute all of them. We achieve the $O(n)$ run time by avoiding explicit computation of the above queries. Instead we compute $css()$ values by counting leaves and color-pair-lca's.

Lemma 1 (CSS Counting Lemma). $css(u) = leafcount(u) - duplicate(u)$.

Proof. The color set size of an internal vertex u is determined by the colors of leaves in $subtree(u)$. If each color appears in either zero or one leaf in $subtree(u)$, then $css(u) = leafcount(u)$. However $leafcount(u)$ is higher than $css(u)$ when there are $k > 1$ leaves with the same color in $subtree(u)$. Among those k leaves, we should only count the leaf that appears first in $LeafList$ and treat the other $k - 1$ leaves as duplicate leaves. For example in Figure 1, we should only count leaves a, b, and g to compute $css(y)$.

Our method of counting the leaves that 'appear first in $LeafList$' is first to count all leaves and then subtract from it the number of 'duplicate leaves'. If x is a 'duplicate' leaf in $subtree(u)$, then both x and $lastleaf(x)$ must be in $subtree(u)$. Otherwise x is not a 'duplicate' leaf. Therefore $lca(lastleaf(x), x)$ must be in $subtree(u)$. Conversely if $lca(lastleaf(x), x)$ is in $subtree(u)$ then both $lastleaf(x)$ and x must be in $subtree(u)$. Thus x is a 'duplicate' leaf in $subtree(u)$. In summary, we have,

Fact 1 x is a 'duplicate' leaf in $subtree(u)$ *iff* $lca(lastleaf(x), x)$ is in $subtree(u)$.

Suppose that in $subtree(u)$ for an internal vertex u, there are $i \geq 1$ 'duplicate' leaves among all colors. Each 'duplicate' leaf x is associated with a color-pair-lca, $lca(lastleaf(x), x)$. Observe that an internal vertex v in $subtree(u)$ can be associated with more than one 'duplicate' leaf. (For example in Figure 1, x is $lca(a, d)$, $lca(d, e)$, and $lca(b, c)$.) It is obvious that if $CPLcount(v) = j \geq 1$, then there are j 'duplicate' leaves in $subtree(u)$ associated with v. This implies that the sum of all $CPLcount()$ values in $subtree(u) = i =$ number of 'duplicate' leaves in $subtree(u)$. Hence,

$$\begin{aligned} leafcount(u) - css(u) &= \text{number of 'duplicate' leaves in } subtree(u) \\ &= \sum_{v \in subtree(u)} CPLcount(v) \\ &= duplicate(u). \end{aligned}$$

□

As a result we can find $css(u)$ by subtracting number of color-pair-lca's in $subtree(u)$ from $leafcount(u)$. For example in Figure 1, there are 5 leaves in $subtree(x)$ and 3 color-pair-lca's (x being $lca(a, d)$, $lca(d, e)$, and $lca(b, c)$), so $css(x) = 5 - 3 = 2$.

2.2 Algorithm

Now we use the CSS Counting Lemma to solve the CSS problem. All we need is to compute the $leafcount()$ and $duplicate()$ values. To compute $duplicate()$ values, we need to compute $CPLcount()$ values. To compute $CPLcount()$ values, we need to compute $lastleaf()$ values. So the algorithm is as follows:

Algorithm CSS1;
    Create the $LeafList$
    Compute the $leafcount()$ values
    Compute the $lastleaf()$ values
    Compute the $CPLcount()$ values
    Compute the $duplicate()$ values
    Compute the $css()$ values
end CSS1

Creating $LeafList$ and computing $leafcount()$ values are straight-forward, using a post-order traversal of T. The $lastleaf()$ values can be computed by a traversal of the $LeafList$ as follows: For every color c we keep $last[c]$, the index of the last leaf colored c that was seen so far. Initially this index is Nil. Every time a new leaf x colored c is encountered, we set $lastleaf(x)$ to $last[c]$, and we update $last[c]$.

Now, we compute $CPLcount()$ values as follows: Initialize all $CPLcount()$ values to zero. For every leaf x, we compute $u = lca(x, lastleaf(x))$ and increment $CPLcount(u)$ by 1.

Finally, we use a post-order traversal to compute all $duplicate()$ values by the recursive relationship $duplicate(u) = CPLcount(u) + \sum_{w \in W} duplicate(w)$ where W is set of children of u. Having computed $leafcount()$ and $duplicate()$ values, computing $css()$ values is trivial.

2.3 Time and space analysis

Computing $lastleaf()$ involves a single scan of $LeafList$, hence can be done in $O(n)$ time and space and $O(m)$ extra space. [19] gives an algorithm that can compute the lca of an arbitrary pair of leaves in $O(1)$ time, with $O(n)$ time preprocessing and $O(n)$ space. So the $CPLcount()$ values can be computed in $O(1)$ time per leaf, and $O(n)$ total time including preprocessing.

Creating $LeafList$, computing $leafcount()$, $duplicate()$, and $css()$ all takes $O(n)$ time and space. Hence we have the first main result:

Theorem 2 (CSS Linear Theorem). *Algorithm CSS1 solves the CSS problem in $O(n)$ time, $O(n)$ space, and $O(m)$ extra space.*

Remark. If $m \gg n$ we can use hashing to get rid of the $O(m)$ extra space. In this case we have $O(n)$ expected time and $O(n)$ space.

3 String matching problems

The all-patterns problem described in §1 can be solved by the CSS solution. The following is a non-complete list of some pattern matching problems that use the all-patterns solution and their time bounds:

1. Given m input strings, for all k between 1 and m, find the longest pattern which appears in at least k input strings.
2. Given m input strings and a fixed integer k between 1 to m, find the longest pattern which appears in at least k input strings. This is a special case of problem 1.
3. Given m input strings and integers k and l, find a pattern with length l which appears in exactly k input strings.
4. Given m input strings and integers $l_1 < l_2$, find the pattern with length between l_1 and l_2 which appears in as many input strings as possible.
   All the above problems can be solved in $O(n)$ time + time to output the answer.
5. Given m input strings and $O(n)$ preprocessing time, answer the query "How many input strings contain a given pattern of length p". This is solved in $O(p)$ time on-line.

There are other variations that fall in the above list, including many problems involving a pattern selection criteria about the length of patterns and the number of input strings containing the pattern. As the solutions to the above problems are similar, we only present solution of problem 2, the k-out-of-m problem. This problem may find applications in DNA matching and computer virus detection. Other problem solutions can be found by modifying the k-out-of-m solution.

3.1 Generalized Suffix Tree

A Suffix Tree is a trie-like data structure that compactly represents a string and is used extensively in string matching. Details of suffix tree can be found in [1, 2, 15, 22]. We only give a brief review.

A suffix tree of a string $W = w_1, \ldots, w_n$ is a tree with $O(n)$ edges and n leaves. Each leaf in the suffix tree is associated with an index i ($1 \leq i \leq n$). The edges are labeled with characters such that the concatenation of the labeled edges that are on the path from the root to leaf with index i is the ith suffix of W. See Figure 2. A suffix tree for a string of length n can be built in $O(n)$ time and space [15]. We say that "vertex v represents the string V" instead of "the path from root to vertex v represents the string V" for simplicity. Properties of suffix tree include:

1. Every suffix of the string W is represented by a leaf in the suffix tree.
2. Length of an edge label of the suffix tree can be found in $O(1)$ time.
3. If a leaf of the suffix tree represents a string V, then every suffix of V is also represented by another leaf in the suffix tree.
4. If a vertex u is an ancestor of another vertex v then the string that u represents is prefix of the string that v represents.
5. If vertices $v_1, v_2, \ldots, v_k$ represent the strings $V_1, V_2, \ldots, V_k$ respectively, then the least common ancestor of $v_1, \ldots, v_k$ represents the longest common prefix of $V_1, \ldots, V_k$.

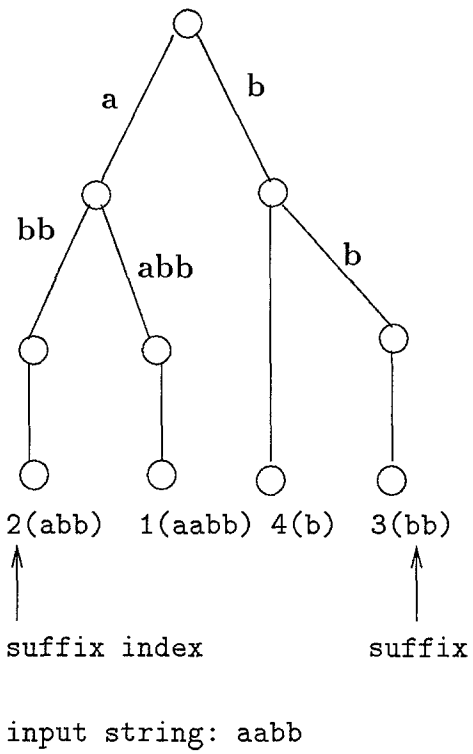
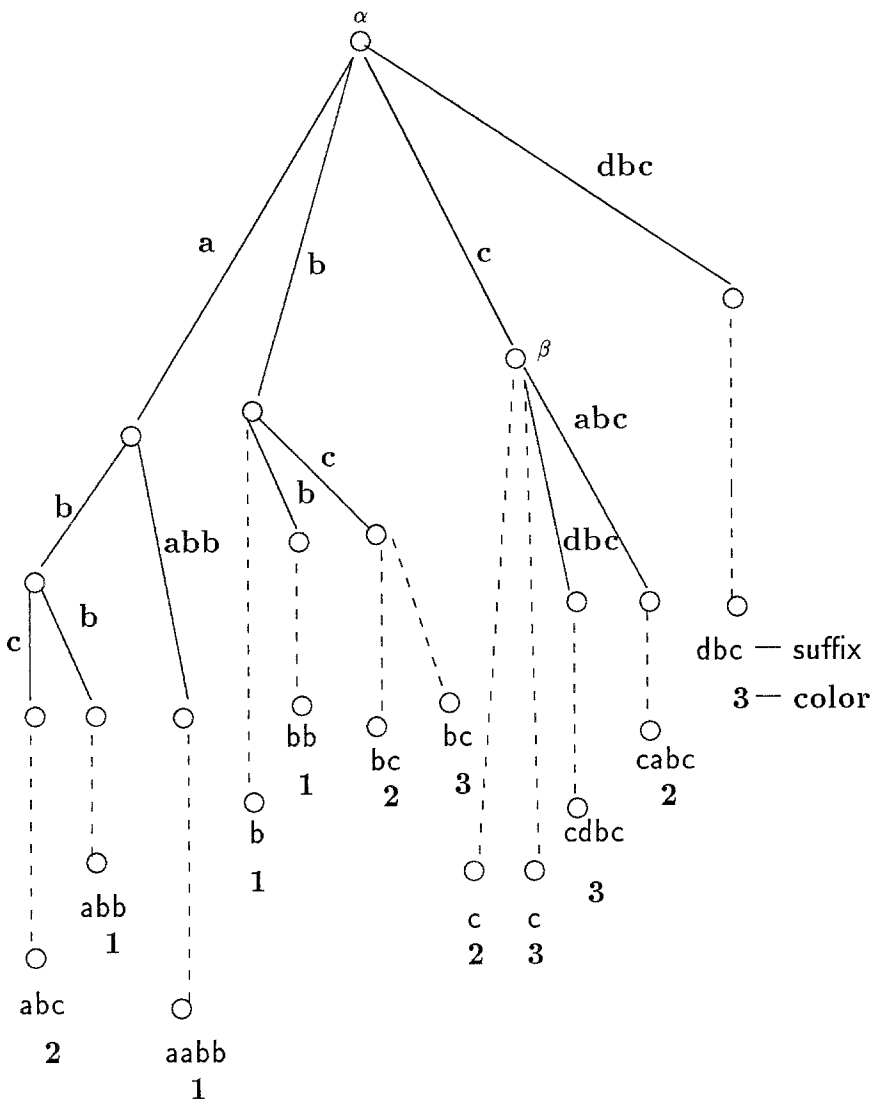


Fig. 2. Suffix Tree Example

The concept of suffix tree can be extended to store more than one input strings. This extension is called the *Generalized Suffix Tree* (GST). Note that sometimes two or more input strings may have the same suffix, and in this case the GST has two leaves corresponding to the same suffix, each corresponds to a different input string. See Figure 3 for an example. All properties of a suffix tree also exist in a GST, except that (**1.**) should be replaced by:

1'. Every suffix of every input string is represented by a leaf in the GST.

A GST can be built in $O(n)$ time and space where n is the total length of all input strings.



input strings: 1 aabb 2 cabc 3 cdbc
bc and c are common suffixes of 2 and 3
$css(\alpha) = 3 \qquad css(\beta) = 2$

Fig. 3. Generalized Suffix Tree Example

3.2 Solving the all-patterns problem by CSS

After we have built the GST for the input strings $W_1, \ldots, W_m$, we color the leaves of a GST in the following sense: we assign color i to a leaf if it represents a suffix of W_i. Now we want to analyze the meaning of a vertex u in GST having color set size b. This means there are leaves representing suffixes from b different input strings in *subtree*(u), which also means that U, the string u represents, is the prefix of suffixes from exactly b different strings. On other words, U is a substring of exactly b different input strings. Hence,

Lemma 3. *Let u be an internal vertex of GST. If $css(u) = b$ then the path from the root to u is a substring of b input strings.*

Therefore, solving the CSS problem instance for the GST is equivalent to finding the number of input strings containing the pattern, for every pattern that appears in the input strings, which is also solving the all-patterns problem.

Building GST requires O(n) time and space. Since m must be $\leq n$, by Theorem 2, solving the CSS problem requires O(n) time and space. Hence,

Theorem 4. *By building the GST and solving the CSS problem instance for it, the all-patterns problem can be solved in $O(n)$ time and space.*

3.3 Solving the k-out-of-m problem

Finally, after solving the all-patterns problem, solving the k-out-of-m problem is the same as finding an internal vertex u in GST such that $css(u) \geq k$ and the path length from root to u is maximum. So the algorithm to solve the k-out-of-m problem is as follows:

```
Algorithm KM;
    Build GST for the input strings
    Solve the CSS problem instance as stated above
    Compute path length for all vertices using pre-order traversal
    Among all vertices, select one with css() value ≥ k and have
        maximum path length
    Output the path from root to the vertex found above as answer
end KM
```

The first two steps together is the solution for the all-patterns problem.

To compute path length for every vertex in GST, we use two observations: (i) path length of root is zero; and (ii) if u is son of v in the GST, then path length of u = path length of v + edge label length of (u,v). With the above two facts, a pre-order traversal is sufficient to compute path length of all vertices. It takes O(n) time and space. Selecting the vertex as answer takes O(n) time and space. Finally, outputting the answer takes O(q) time and space where q is the output size. Hence,

Theorem 5. *Algorithm KM solves the k-out-of-m problem in $O(n + q)$ time and space, where n is the input size and q is the output size.*

Remark. Other string matching problems listed in this paper can be solved by modifying the k-out-of-m solution. The first three steps: building the GST, solving the CSS problem instance, and computing path lengths are common to all problem solutions. Only the last selection step is different. As designing the selection step is straight-forward, we skip the details.

4 Parallel implementation of CSS1

The CSS1 algorithm is easily parallelizable. There is a PRAM solution of the CSS problem (called the PCSS algorithm) which uses the exact framework of the sequential CSS1. We just parallelize all the steps.

4.1 The Euler Tour technique

The PCSS algorithm uses the *Euler Tour* technique [20, 21] which is extensively used in parallelizing tree algorithms such as computing ear decomposition [16] and the accelerated centroid decomposition technique [7]. A tree T is treated as an undirected graph and a directed graph T' corresponding to T is created in the following sense: (i) The vertices of T and T' are the same. (ii) For each edge u — v in T there are two edges, $u \rightarrow v$ and $v \rightarrow u$, in T'. Since the in-degree and out-degree of each vertex in T' are the same, T' has an Euler path that starts and ends at the root of T'. This is an Euler Tour of T. The Euler Tour for Figure 1 is : y →x →i →a →i →b →i →x →j →c →j →d →j →x →e →x →y →z →f →z →w →g →w →h →w →z →y. For a vertex u, if u has k children then there are $k+1$ appearances of u in the Euler Tour. We call each appearance an *Euler record* of u. The *Euler segment* of vertex u is the segment of the Euler Tour which starts at the first Euler record of u and ends at the Euler record of u. All Euler records of any vertex in $subtree(u)$ must fall in the Euler segment of u.

Suppose we have a numeric field $f(u)$ in every vertex u of T, and we want to compute $\sum_{v \in subtree(u)} f(v)$, for every internal vertex u. Here the Euler Tour technique is applicable. For each vertex we associate its $f()$ value with one of its Euler records in the Euler Tour. After that, for any vertex u, $\sum_{v \in subtree(u)} f(v)$ is equal to the sum of $f()$ values for all Euler records in u's Euler segment. Summation over all sublists (the Euler segments) in a list (the Euler Tour) can be easily reduced to a prefix sum problem, which requires $O(\log n)$ time using $\frac{n}{\log n}$ processors on an EREW PRAM [7]. We apply this technique to compute $leafcount()$ and $duplicate()$. We use parallel sorting [3, 5, 9, 10] and parallel prefix sum to implement the other steps of PCSS.

4.2 Creating *LeafList* and computing the *leafcount*() values

Note that the Euler Tour can be prepared using $\frac{n}{\log n}$ processors, in $O(\log n)$ time, on an EREW PRAM [7].

In the Euler Tour, the order of the leaves is exactly the same as the order of the leaves in *LeafList*. We create the *LeafList* from the Euler Tour by the following method. For the ith node in the Euler Tour, we assign a list L_i to it. If the ith node

in the Euler Tour is a leaf x then we set L_i to contain x only; otherwise we set L_i to an empty list. After that, we compute $LeafList = L1||L2||L3||...||L_{\hat{n}}$, where $||$ is the concatenate operator, and $\hat{n}$ is the length of the Euler Tour ($\hat{n}$ is $O(n)$). Since $||$ is associative, we can use parallel prefix sum [12]. This takes $\frac{n}{\log n}$ processors $O(\log n)$ time on an EREW PRAM.

By associating a '1' to every leaf and '0' to every internal vertex, computing $leafcount(u)$ means to sum the numbers in $subtree(u)$. Using the Euler Tour technique described in §4.1, we can reduce this to a prefix sum problem on the Euler Tour, which can be solved by $\frac{n}{\log n}$ processors, in $O(\log n)$ time on an EREW PRAM.

4.3 Computing the *lastleaf*() values

One obvious approach is to group the leaves according to their colors. This can be achieved by sorting. [3, 5] give optimal sorting algorithms which using n processors, takes $O(\log n)$ time on an EREW PRAM.

In many cases we can assume the number of colors, m, is $O(n)$, so we can use parallel bucket sort to overcome the $\Omega(n \log n)$ work barrier of comparison-based sorting algorithms. A parallel bucket sort can be computed: (i) using $\frac{n}{\log n}$ processors, $EO(\log n)$ parallel expected time on a priority CRCW PRAM [18] (we will discuss how this is achieved in the next paragraph); (ii) using $\frac{n}{\log n} \log\log n$ processors, $O(\log n)$ time on a priority CRCW by the algorithm of [9]; (iii) or using $n^{1-\epsilon}$ processors, $O(n^{\epsilon})$ time on an EREW PRAM for any $\epsilon > 0$ [10];

In the computation of $lastleaf()$ values we need stable sorting. The randomized parallel integer sorting algorithm in [18] is unstable. We use the following two steps to achieve the same effect as a stable sort on the colors: (Step 1) We sort the leaves according to their colors. After this step leaves of a particular color are grouped together. (Step 2) For every color c, we sort (in parallel) all leaves of color c according to their position in the $LeafList$. The above two steps together have the same effect as a stable sorting on the colors. The asymptotic running time is not increased.

4.4 Computing the *CPLcount*() values

One difficulty in computing the $CPLcount()$ values in parallel is that more than one leaves (of the same or different colors) may want to increment the $CPLcount()$ value of the same internal vertex at the same time. For example in Figure 1 the leaves c, d, and e want to increment $CPLcount(x)$ simultaneously. Following shows how to solve this problem using integer sorting.

[19] shows that with $\frac{n}{\log n}$ processors we can preprocess a tree in $O(\log n)$ time such that k lca queries can be answered in constant time using k processors on a CREW PRAM. We use this to parallelize computation of $CPLcount()$ values. Each leaf x is assigned a processor. This processor creates a record which contains the vertex id $u = lca(lastleaf(x), x)$. If $lastleaf(x)$ does not exist then the processor creates an empty record. After that all processors form an array, RA, of all the created records. This can be done in constant parallel time using n processors. For an internal vertex u, the number of occurrences of its id in RA is equal to $CPLcount(u)$.

We can use a parallel bucket sort on RA to group all occurrences of the same internal vertex together. Finally a prefix sum can be used to count the number of

occurrences of each internal vertex in RA, which is the $CPLcount()$ values of that internal vertex.

As a prefix sum takes $O(\log n)$ time with $\frac{n}{\log n}$ processors, the bottleneck of this step is in the sorting part, which has the same run time as computing $lastleaf()$.

4.5 Computing the *duplicate*() values

By using the relation $duplicate(u) = \sum_{w \in subtree(u)} CPLcount(w)$, we can use the same Euler Tour technique as described before (by associating $CPLcount(v)$ to an Euler record of v in the Euler Tour, for every vertex v) to transform this to a prefix sum problem. The running time is the same as computing $leafcount()$.

4.6 Computing the *css*() values and overall run time

Finally computing the $css()$ values can be easily done by n processors, in $O(1)$ time on an EREW PRAM.

Therefore overall run time of PCSS is (assume m is $O(n)$):

- Using $\frac{n}{\log n}$ processors, $EO(\log n)$ parallel expected time on a priority CRCW PRAM.
- Or using $\frac{n}{\log n} \log\log n$ processors, $O(\log n)$ time on a priority CRCW PRAM.
- Or using $n^{1-\epsilon}$ processors, $O(n^{\epsilon})$ time for any $\epsilon > 0$ on a CREW PRAM.

Remark. If m is unbounded, we have to use the parallel sorting of [5], which uses n processors, $O(\log n)$ time on a CREW PRAM. The overall run time of PCSS is using n processors, $O(\log n)$ time on a CREW PRAM.

5 Parallel all-patterns solution

Computing the path length for all vertices can be reduced to a prefix sum problem on the Euler Tour. This is achieved by assigning length of the Euler Tour edges in the following way. Suppose u_1 is parent of u_2 in the GST, and the edge label length of (u_1, u_2) is d, then we assign $+d$ as label length of the edge $u_1 \rightarrow u_2$ in the Euler Tour, and assign $-d$ as label length of the edge $u_2 \rightarrow u_1$ in the Euler Tour. After this assignment, the path length from root to a vertex v in the GST is the sum of all edge label lengths from the start of the Euler Tour to the first Euler record of v. As stated before, this takes $\frac{n}{\log n}$ processors $O(\log n)$ time on an EREW PRAM.

The critical step here is to construct the generalized suffix tree in parallel. The first parallel algorithm in constructing suffix tree is given in [13]. It runs in $O(\log n)$ time and uses $n^2/\log n$ processors. The best result is [2]. An arbitrary CRCW PRAM algorithm was given in [2], which runs in $O(\log n)$ time and uses n processors. Its approach can be generalized to construct generalized suffix tree so this is also the current best parallel time bound for solving the all-patterns problems (and other listed strings matching applications).

Acknowledgement. The author wants to thank Chip Martel, Dan Gusfield, and Dalit Naor for their helpful comments about the presentation of this paper, and also Dan for introducing the k-out-of-m problem in his string matching class.

References

1. A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, NATO ASI Series, Series F: Computer and System Sciences, Vol. 12*, pages 85–96, Springer-Verlag, Berlin, 1985.
2. 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.
3. M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. of the 15th ACM Symposium on Theory of Computing*, pages 1–9, 1983.
4. S. Altschul and D. Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Math*, 49:197–209, 1989.
5. R. Cole. Parallel merge sort. In *Proc. 27nd Annual Symposium on the Foundation of Computer Science*, pages 511–516, 1986.
6. H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Math*, 48:1073–1082, 1988.
7. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
8. D. Gusfield. *Efficient methods for multiple sequence alignment with guaranteed error bounds.* Technical Report CSE-91-4, Computer Science, U. C. Davis, 1991.
9. T. Hagerup. Towards optimal parallel bucket sorting. *Information and Computation*, 75:39–51, 1987.
10. C. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Trans. Comput.* C-34:965-968, 1985.
11. R. Lo. personal communications. 1991.
12. L. Ladner and M. Fischer. Parallel prefix computation. *J.A.C.M.*, 27:831–838, 1980.
13. G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proc. of the 18th ACM Symposium on Theory of Computing*, pages 220–230, 1986.
14. H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11(13):4629–4634, 1983.
15. E. M. McCreight. A space-economical suffix tree construction algorithm. *J.A.C.M.*, 23(2):262–272, 1976.
16. Y. Maon, B. Schieber, and U. Vishkin. Open ear decomposition and s-t numbering in graphs. *Theoretical Computer Science*, 1987.
17. V. R. Pratt. Improvements and applications for the weiner repetition finder. 1975. unpublished manuscript.
18. S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18:594–607, 1989.
19. B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.
20. R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.
21. U. Vishkin. On efficient parallel strong orientation. *I.P.L.*, 20:235–240, 1985.
22. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.