1 APL6: Common substrings of more than two strings

One of the most important questions asked about a *set* of strings is what substrings are common to a large number of the *distinct* strings. This is in contrast to the important problem of finding substrings that occur repeatedly in a *single* string.

In biological strings (DNA, RNA or protein) the problem of finding substrings common to a large number of distinct strings arises in many different contexts. In fact, the task of finding (inexactly matching) substrings that appear frequently in a set of strings is almost an industry in some subareas of molecular biology. We will say much more about this when we discuss database searching in Chapter **??** and multiple string comparison in Chapter **??**. Most directly, the problem of finding common substrings arises because mutations that occur in DNA after two species diverge will more rapidly change those parts of the DNA or protein that are less functionally important. The parts of the DNA or protein that are critical for the correct functioning of the molecule will be more highly conserved, because mutations that occur in those regions will more likely be lethal. So finding DNA or protein substrings that occur commonly in a wide range of species helps point to regions or characters that may be critical for the function or structure of the biological string.

Less directly, the problem of finding (exactly matching) common substrings in a set of distinct strings arises as a subproblem of many heuristics developed in the biological literature to *align* a *set* of strings. That problem, called multiple alignment, will be discussed in some detail in Section **??**.

The biological applications motivate the following *exact* matching problem: Given a set of strings, find substrings "common" to a large number of those strings. The word "common" here means "occurring with equality". A more difficult problem is to find "similar" substrings in many given strings, where "similar" allows a small number of differences. Problems of this type will be discussed in Part III.

Formal problem statement and first method

Suppose we have K strings whose lengths sum to n.

Definition For each k between 2 and K, we define $l(k)$ to be the length of the *longest substring common to at least k of the strings.*

We want to compute a table of $K-1$ entries where entry k gives $l(k)$ and also points to one of common substrings of that length. For example, consider the set of strings {*sandollar, sandlot, handler, grand, pantry*}. Then the $l(k)$ values (without pointers to the strings) are:

```
k              l(k)    one substring
```

```
-------------------------------------
 2                4       sand
 3                3        and
 4                3        and
 5                2        an
```

Surprisingly, the problem can be solved in linear, $O(n)$, time [1]. It really is amazing that so much information about the contents and substructure of the strings can be extracted in time proportional to the time needed just to read in the strings. The linear time algorithm will be fully discussed in Chapter **??** after the constant time lowest common ancestor method has been discussed.

To prepare for the $O(n)$ result, we show here how to solve the problem in $O(Kn)$ time. That time bound is also non-trivial, but is achieved by a generalization of the longest common substring method for two strings. First, build a generalized suffix tree $\mathcal{T}$ for the K strings. Each leaf of the tree represents a suffix from one of the K strings, and is marked with one of K unique string identifiers, 1 to K, to indicate which string the suffix is from. Each of the K strings is given a distinct termination symbol, so that identical suffixes appearing in more than one string end at distinct leaves in the generalized suffix tree. Hence, each leaf in $\mathcal{T}$ has only one string identifier.

Definition For every internal node v of $\mathcal{T}$, define $C(v)$ to be the number of *distinct* string identifiers that appear at the leaves in the subtree of v.

Once the $C(v)$ numbers are known, and the string-depth of every node is known, the desired $l(k)$ values can be easily accumulated with a linear time traversal of the tree. That traversal builds a vector V where, for each value of k from 2 to K, $V(k)$ holds the string-depth (and location if desired) of the deepest (string-depth) node v encountered with $C(v) = k$. (When encountering a node v with $C(v) = k$ compare the string-depth of v to the current value of $V(k)$ and if v's depth is greater than $V(k)$, change $V(k)$ to the depth of v.) Essentially, $V(k)$ reports the length of the longest string that occurs *exactly* k times. Therefore $V(k) \leq l(k)$. To find $l(k)$ simply scan V from largest to smallest index writing into each position the maximum $V(k)$ value seen. That is, if $V(k)$ is empty or $V(k) < V(k+1)$ then set $V(k)$ to $V(k+1)$. The resulting vector holds the desired $l(k)$ values.

1.1 Computing the $C(v)$ numbers

In linear time, it is easy to compute for each internal node v the number of leaves in v's subtree. But that number may be larger than $C(v)$ since two leaves in the subtree may have the same identifier. That repetition of identifiers is what makes it hard to compute $C(v)$ in $O(n)$ time. So, instead of counting the number of leaves below v, the algorithm uses $O(Kn)$ time to explicitly compute which identifiers are found

below any node. For each internal node v, a K-length bit-vector is created which has a 1 in bit i if there is a leaf with identifier i in the subtree of v. Then $C(v)$ is just the number of 1-bits in that vector. The vector for v is obtained by OR'ing the vectors of the children of v. For l children, this takes lK time. Therefore over the entire tree, since there are $O(n)$ edges, the time needed to build the entire table is $O(Kn)$. We will return to this problem in Section 2 where an $O(n)$ time solution will be presented.

2 A linear time solution to the multiple common substring problem

Above, a generalized suffix tree $\mathcal{T}$ was constructed for the K strings of total length n, and the table of all the $l(k)$ values was obtained by operations on $\mathcal{T}$. That method had a running time of $O(Kn)$. In this section we reduce the time to $O(n)$. The solution was obtained by Lucas Hui [1][1].

Recall that for any node v in $\mathcal{T}$, $C(v)$ is the number of distinct leaf string identifiers in the subtree of v, and that a table of all the $l(k)$ values can be computed in $O(n)$ time once all the $C(v)$ values are known. Recall also that $S(v)$ is the total number of leaves in the subtree of v, and that $S(v)$ can easily be computed in $O(n)$ time for all nodes.

Certainly $S(v) \geq C(v)$ for any node v, and it will be strictly greater when there are two or more leaves of the same string identifier in v's subtree. Our approach to finding $C(v)$ is to compute both $S(v)$ and a correction factor $U(v)$ which counts how many "duplicate" suffixes from the same string occur in v's subtree. Then $C(v)$ is simply $S(v) - U(v)$.

Definition: $n_i(v)$ is the number of leaves with identifier i in the subtree rooted at node v. Let n_i be the total number of leaves with identifier i.

With that definition, we immediately have the following

Lemma 2.1 *$U(v) = \sum_{i:n_i(v)>0}(n_i(v) - 1)$, and $C(v) = S(v) - \sum_{i:n_i(v)>0}(n_i(v) - 1)$.*

We show below that all the correction factors for all internal nodes can be computed in $O(n)$ total time. That then gives an $O(n)$ time solution to the k-common substring problem.

[1]In the introduction of an earlier unpublished manuscript [3], Pratt claims a linear time solution to the problem but the claim doesn't specify whether the problem is for a *fixed* k or for *all* values of k. The section where the details were to be presented is not available and was apparently never finished [2].

2.1 The method

The algorithm first does a depth-first traversal of $\mathcal{T}$, numbering the leaves in the order that they are encountered. That numbering has the familiar property that for any internal node v, the numbers given to the leaves in the subtree rooted at v are consecutive, i.e., they form a consecutive interval.

For purposes of the exposition, let us focus on the single identifier i and show how to compute $n_i(v) - 1$ for each internal node v. Let L_i be the list of leaves with identifier i, in increasing order of their *dfs* numbers. For example, in Figure 1, the leaves with identifier i are shown boxed and the corresponding L_i is $1, 3, 6, 8, 10$. By the properties of depth-first numbering, for the subtree rooted at any internal node v, all the $n_i(v)$ leaves with identifier i occur in a consecutive interval of list L_i. Call that interval $L_i(v)$. If x and y are any two leaves in $L_i(v)$, then the *lca* of x and y is a node in the subtree of v. So if we compute the *lca* for each *consecutive* pair of leaves in $L_i(v)$, then all of the $n_i(v) - 1$ computed *lca*'s will be found in the subtree of v. Further, if x and y are not both in the subtree of v, then the *lca* of x and y will *not* be a node in v's subtree. This leads to the following lemma and method.

Lemma 2.2 *If we compute the lca for each consecutive pair of leaves in L_i, then for any node v, exactly $n_i(v) - 1$ of the computed lca's will lie in the subtree of v.*

Lemma 2.2 is illustrated in Figure 1.

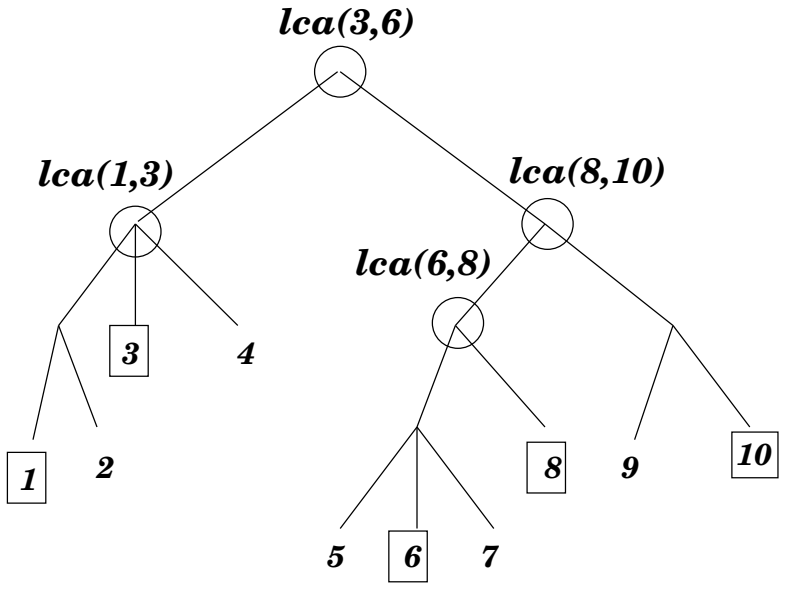

Figure 1: The boxed leaves have identifier i. The circled internal nodes are the lowest common ancestors of the four adjacent pairs of leaves from list L_i.

Given the lemma, we can compute $n_i(v) - 1$ for each node v by the following method. Compute the *lca* of each consecutive pair of leaves in L_i, and accumulate for

each node w a count of the number of times that w is the computed *lca*. Let $h(w)$ denote that count for node w. Then for any node v, $n_i(v) - 1$ is exactly $\sum[h(w) : w$ is in the subtree of $v]$. A standard $O(n)$-time bottom-up traversal of $\mathcal{T}$ can therefore be used to find $n_i(v) - 1$ for each node v.

To find $U(v)$, we don't want $n_i(v) - 1$ but rather $\sum_i [n_i(v) - 1]$. But the algorithm must not do a separate bottom-up traversal for each identifier, since then the time bound would then be $O(Kn)$. Instead, the algorithm should defer the bottom-up traversal until each list L_i has been processed, and it should let $h(w)$ count the total number of times that w is the computed *lca* over all of the lists. Only then is a single bottom-up traversal of $\mathcal{T}$ done. At that point, $U(v) = \sum_{i:n_i>0}[n_i(v)-1] = \sum[h(w) : w$ is in the subtree of $v]$.

We can now summarize the entire $O(n)$ method for solving the k-common substring problem.

Begin

1. Build a generalized suffix tree $\mathcal{T}$ for the K strings.
2. Number the leaves of $\mathcal{T}$ as they are encountered in a depth-first traversal of $\mathcal{T}$.
3. For each string identifier i, extract the ordered list L_i of leaves with identifier i. (The minor implementation detail needed to do this in $O(n)$ total time is left to the reader.)
4. For each node w in $\mathcal{T}$ set $h(w)$ to zero.
5. For each identifier i, compute the *lca* of each consecutive pair of leaves in L_i, and increment $h(w)$ by one each time that w is the computed *lca*.
6. With a bottom-up traversal of $\mathcal{T}$, compute, for each node v, $S(v)$ and $U(v) = \sum_{i:n_i>0}[n_i(v) - 1] = \sum[h(w) : w$ is in the subtree of $v]$.
7. Set $C(v) = S(v) - U(v)$ for each node v.
8. Accumulate the table of $l(k)$ values as detailed in Section 1.

End.

2.2 Time analysis

The size of the suffix tree is $O(n)$ and preprocessing of the tree for *lca* computations is done in $O(n)$ time. There are then $\sum_{i=1}^{K} |n(i) - 1| < n$ *lca* computations done, each of which takes constant time, so all the *lca* computations take $O(n)$ time in total. Hence only $O(n)$ time is needed to compute all $C(v)$ values. Once these are known, only $O(n)$ additional time is needed to build the output table. That part of the algorithm is the same as in the previously discussed $O(Kn)$ time algorithm of Section 1. Therefore

Theorem 2.1 *Let $\mathcal{S}$ be a set of K strings of total length n, and let $l(k)$ denote the length of the longest substring that appears in at least k distinct strings of $\mathcal{S}$. A table of all $l(k)$ values, for k from 2 to K, can be built in $O(n)$ time.*

That so much information about the substrings of $\mathcal{S}$ can be obtained in time proportional to the time needed just to *read* the strings is very impressive. It would be a good challenge to try to obtain this result without the use of suffix trees (or a similar data structure).

References

[1] L. Hui. Color set size problem with applications to string matching. *Proc. 3rd Symp. on Combinatorial Pattern Matching. Springer LNCS 644*, pages 227–240, 1992.

[2] V. Pratt. Personal Communication.

[3] V. Pratt. Applications of the weiner repetition finder. Unpublished manuscript, 1973.