Bottlenecks in search for Maximal Unique Matches in Suffix Tree

Julio César García Vizcaíno
Computer Architecture & Operating Systems Department
Universitat Autònoma de Barcelona
Bellaterra (Barcelona), Spain
Email: jcgarcia@caos.uab.cat

***Abstract*—A suffix tree is a data structure very known to solve many classic string matching problems. One problem that can be solved with a suffix tree is the MUM-problem. Assume we are given an alphabet Σ^*, two sequences R, Q $\in \Sigma^*$, and a number L > 0. The maximal unique matches problem (MUM-problem) is to find all sequences u $\in \Sigma^*$ with: $|u| \geq L$, u occurs exactly once in R and once in Q, and for any character a $\in \Sigma^*$ neither ua nor au occurs both in R and Q. However, the search for MUMs in a suffix tree presents challenges due to irregular and unpredictable data accesses in the suffix tree traversal. This paper analyzes these bottlenecks in the MUM-problem.**

I. Introduction

Let's assume the Reference sequence $R[0, \ldots, n-1]$ of size $|R| = n$ over an alphabet $\Sigma = \$, A, C, G, T$ which has a sentinel character $R[n-1] = \$$ that occurs nowhere else in the Reference genome and is lexicographically less than all the characters that occur in the alphabet. The suffixes of the Reference genome are zero indexed by their position in the original Reference by a full-text index data structure like a suffix tree.

Definition 1. *A suffix tree, ST, for an n-character string R is a rooted directed tree with exactly n leaves numbered 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of R. No two edges out of a node can have edge-labels beginning with the same character. For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of R that starts at position i. That is, it spells out $R[i \ldots n]$. [1].*

A search for a MUM between a Reference, $|R| = n$, and a Query, $|Q| = m$, can be done in a suffix tree in $O(n+m)$ steps. However it is possible to improve the search for MUMs in a suffix tree by using suffix links [2]. This improvement reduce the search for MUMs to $O(m)$ steps.

We also define a suffix link, which is an important trick to achieve linear complexity to search for MUMs in both suffix tree.

Definition 2. *A suffix link is a pointer from string $\overline{aw}$ to substring $\overline{w}$.*

II. Organization of suffix tree in memory

We use a suffix tree to search for MUMs. It follows an explanation of how suffix tree is stored in memory and how we traverse the suffix tree.

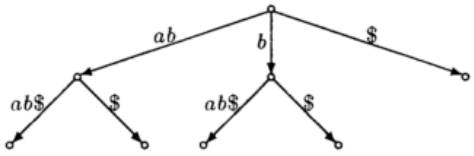

Fig. 1. Suffix tree for string x=abab, [3].

T_{leaf}					
leaf	$\overline{abab\$}$	$\overline{bab\$}$	$\overline{ab\$}$	$\overline{b\$}$	$\overline{\$}$
leaf number j	1	2	3	4	5
$T_{leaf}[j]$	**3**	**4**	nil	nil	nil

T_{branch}			
branching node	*root*	$\overline{ab}$	$\overline{b}$
node number	1	2	3
firstchild	2	**1**	**2**
branchbrother	nil	3	**5**
depth	0	2	1
headposition	1	3	4
suffixlink		3	1

Fig. 2. Tables to store suffix tree for string x=abab, [3].

A suffix tree for string x=abab is shown in Figure 1.

We use the implementation of a suffix tree suggested in [3] which has two structures: a reference structure which consists of an address pointing a leaf, or to a branching node. The boolean `toleaf` is `True` iff `address` points to a leaf. And a branchinfo structure:

- headposition: the head position of the branching node
- depth: the depth of the branching node
- suffixlink: the suffix link is always to a branching node
- firstchild: the reference to the first child
- branchbrother: the reference to the right brother, if this doesn't exist then it's `NULL`

For the suffix tree in Figure 1 the tables are shown in Figure 2.

The successors of a branching node are therefore found in a list whose elements are linked via the firstchild, branchbrother, and T_{leaf} references. To speed up the access to the successors, each such list is ordered according to the first character of the edge labels. A record component has the following notation: for any component c and any branching node $\overline{w}$, $\overline{w}.c$ denotes the component stored in the branch record $T_{branch}[\overline{w}]$. The head position j of some branching node $\overline{wu}$ tells us that the leaf $\overline{S}_j$ occurs in the subtree below node $\overline{wu}$. Hence wu is the prefix of S_j of length $\overline{wu}.depth$. As a consequence, the label

of the incoming edge to node $\overline{wu}$ can be obtained by dropping the first $\overline{w}.depth$ characters of wu, where $\overline{w}$ is the predecessor of $\overline{wu}$.

Observation 1. *If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu}$ is a branching node, then we have $u = x_i \dots x_{i+l-1}$ where $i = \overline{wu}.headposition + \overline{w}.depth$ and $l = \overline{wu}.depth - \overline{w}.depth$.*

Similarly, the label of the incoming edge to a leaf is determined from the leaf number and the depth of the predecessor.

Observation 2. *If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu} = \overline{S}_j$ for some $j \in [1, n+1]$, then $u = x_i \dots x_n\$$ where $i = j + \overline{w}.depth$.*

Every string occurring in a suffix tree ST can uniquely be described in terms of the nodes and edges of ST. This is formalized in the notion of *locations*:

Definition 3. *Let ST be a suffix tree and $s \in words(ST)$. The location of s in ST, denoted by $loc_{ST}(s)$ is defined as follows:*

- *If $\overline{s}$ is an internal node, then $loc_{ST}(s) = \overline{s}$.*
- *If $\overline{s}$ is a leaf in ST, then there is a leaf edge $\overline{u} \xrightarrow{v} \overline{s}$ in ST and $loc_{ST}(s) = (\overline{u}, v, \epsilon, \overline{s})$.*
- *If there is no node $\overline{s}$ in ST, then there is an edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ in ST such that $s = uv, v \neq \epsilon, w \neq \epsilon$ and $loc_{ST}(s) = (\overline{u}, v, w, \overline{uvw})$.*

$locations(ST) = \{loc_{ST}(s) | s \in words(ST)\}$ is the *set of locations* in ST. If a location is a node, we call it *node location*, otherwise *edge location*. Sometimes we identify a node location with the corresponding node.

By convention the *root* is the location of ϵ in the empty ST. The location of a string corresponding to a leaf is not the leaf itself. Instead, it is defined in terms of the edge leading to the leaf. Note that an edge-location $(\overline{u}, v, w, \overline{uvw})$ corresponds to a leaf iff $w = \epsilon$. In an edge-location $(\overline{u}, v, w, \overline{uvw})$ the string w and the node $\overline{uvw}$ are redundant. Both can uniquely be determined from $\overline{u}$ and v provided the ST is given.

For convenience some additional notions related to locations are introduced.

Definition 4. *Given a suffix tree ST, suppose $s \in words(ST)$.*

1) *$|loc_{ST}(s)| = |s|$ is the depth of the location $loc_{ST}(s)$.*
2) *Let v be the shortest string s.t. $\overline{sv} \in nodes(ST)$. Then $\overline{sv}$ is denoted by $ceiling(loc_{ST}(s))$.*
3) *For all $a \in \Sigma$ we define: $occurs(loc_{ST}(s), a) \longleftrightarrow sa$ occurs in ST.*
4) *$rescan(loc_{ST}(s), w)$ denotes $loc_{ST}(sw)$ for all $sw \in words(ST)$.*

Following a path is the most important operation to search for MUMs. Function $TraverseSuffixTree$ describes this operation.

Definition 5. *Let ST be a suffix tree. For each $s \in words(ST)$ and each string w the function $scanprefix : locations(ST) \times \Sigma^* \to locations(ST) \times \Sigma^*$ is specified as follows:*

$$TraverseSuffixTree(loc_{ST}(s), w, ST) = (loc_{su}, v),$$

where $uv = w$ and u is the longest prefix of w such that $su \in words(ST)$.

Moreover, it is very important to have an efficient access from $loc_{ST}(cy)$ to $loc_{ST}(y)$. This access is provided by a function suffixlink, that uses the suffix links of the inner nodes as a "shortcut".

Definition 6. *Let ST be a suffix tree. Function suffixlink, $suffixlink : locations(ST) \setminus \{root\} \to locations(ST)$ is defined as:*

$$suffixlink(\overline{s}) = \overline{z} \text{ where } \overline{s} \to \overline{z} \text{ is the suffix link for } \overline{s}$$

$$suffixlink(\overline{u}, av, w, \overline{uavw}) = \begin{cases} loc_{ST}(v) & \text{if } \overline{u} = root \\ rescan(\overline{z}, av) & \text{otherwise} \end{cases}$$

$$\text{where } \overline{u} \to \overline{z} \text{ is the suffix link for } \overline{u}$$

III. Search for MUMs in suffix tree

Once we have defined the preliminaries in the MUM-problem, we need to answer the question: Where are the MUMs of R and Q of some minimum length L? We show the algorithm used to answer this question using a suffix tree, see Algorithm 1.

Algorithm 1: Search for MUMs in a suffix tree.

```
input : R, Q, L
output: List of MUMs of length ≥ L, with start
        position in R and Q and length
begin
    ST ← buildST(R);
    loc ← TraverseSuffixTree(Q[0],ROOT,ST);
    for i=1 to |Q| do
        if isLeafNode(loc) and loc.length ≥ L then
            /* Leaf saves the position of
               a suffix in ST.                */
            ;
            if R[leaf − 1] ≠ Q[i − 1] then
                MUMcands ←
                saveMUMcand(R_leaf,i,length);
        if isRootNode(loc) then
            loc ← TraverseSuffixTree(Q[i +
            1],ROOT,ST);
        else
            suffixlink(ST,loc);
            loc ← TraverseSuffixTree(Q[i +
            1],loc.node,ST);
    /* Get unique MUMs from list of
       MUM-candidates.                        */
    ;
    MUMs ← cleanMUMcand(MUMcands)
end
```

In Algorithm 1 there are two functions related to suffix tree. TraverseSuffixTree performs the traversal of the suffix tree with the current string and it starts in some given node. suffixlink is a function which gets the suffix link from the

current node pointed out by loc.node. If b is an internal node v of suffix tree, then the algorithm can follow its suffix link to a node $s(v)$. If b is not an internal node, then the algorithm can back up to the node v just above b. If v is the root, then the search starts in root. But if v is not the root, then the algorithm follows the suffix link from v to $s(v)$.

IV. Suffix tree bottleneck

Suffix tree is the archetypical index structure used in bioinformatics. For large-scale bioinformatics applications, however, memory consumption really becomes a bottleneck. Moreover, the algorithm to search for MUMs involves comparing the search of a character to the character stored at a specific node at every level of the tree, and traversing a child node based on the comparison results. Only one node at each level is actually accessed, resulting in ineffective cache line utilization, to the linear storage of the tree. Since the result of the comparison is required before loading the appropiate child node, cache line prefetches cannot be issued beforehand. Moreover, meanwhile the Algorithm 1 shows a $O(m)$ complexity by using suffix links, this feature has the disadvantage of a quite random access to the next node used to traverse the suffix tree. As a consequence a search for MUM typically involves a long-latency TLB/cache miss followed by small number of arithmetic operations at each level of the tree, leading to ineffective utilization of the processor resources. The best scenario (short-latency and no TLB/cache miss) in a search for MUM is a layout of a whole branch in a cache line size. The worst scenario (long-latency and TLB/cahce miss) is a layout of only one branch read in a cache line size. In Figure 1 the suffix tree is implemented with two tables as in Figure 2. The layout of branch table has a poor temporal and spatial locality. If string "ba" is searched in the suffix tree by using Algorithm 1, then branch node number 1 in Figure 2 is accessed, it follows an access to branch node number 3 and finally leaf number 3. Use of tables leaf and branch in Figure 2 helps to traverse a suffix tree, but since each branch node is implemented as a linked list this is a serious performance issue. The linked list have items in disjoint areas of memory. To traverse the list the cache lines cannot be utilized effectively. One could say that the linked list is cache line hostile, or that the linked list maximizes cache line misses. The disjoint memory will make traversing of the linked list slow because RAM fetching will be used extensively.

A. Memory reference locality

This layout in Figure 2 of a suffix tree and the random search of strings do not help to a better use of processor resources. The use of suffix links reduces the time complexity of a search for a MUM in a suffix tree but it is not very cache-friendly.

As a proof a poor memory reference locality of this data structure, two experiments were carried out. The features of the computer used are: 2 Processor Intel(R) Xeon(R) E5645 @ 2.4GHz of 6 cores each one, 32KB L1 cache, 256KB L2 and 12MB L3 shared cache per socket. RAM: 96 GB. GCC 4.7.0 with OpenMP support + Linux.

The genomes used were a reference genome of 312Kbp[1] and 169Mbp, a query genome of 2.91Gbp and a minimum length of MUM of 20bp.

By using the memory heriarchy, we may overlap the latency of a random traversal in a suffix tree. The first experiment is with a small suffix tree (reference of 312Kbp), which is kept in the L3 cache (6MB of memory consumption for suffix tree), the query genome of 2.91Gbp is kept in main memory. The total time to search for MUMs was 149.58 seconds. The second experiment is with a bigger sufffix tree (reference of 169Mbp), which is kept in main memory (3380GB of memory consumption for suffix tree), the query genome of 2.91Gbp is kept in main memory. The total time to search for MUMs was 268.58 seconds. Furthermore, the functions in Algorithm 1 were profiled to get how many times are accessed in order to have an idea of the bottlenecks while traversing a suffix tree with suffix links. In experiment 1, 6.18% of times the traversal started in the root of suffix tree. 60.38% of times suffix links were used. By using suffix links 59.99% of times the function rescan was used. These measures show how heavy and intensive is the used of suffix links and how it may degrade the performance. In experiment 2, 4.12% of times the traversal started in the root of suffix tree. 66.47% of times suffix links were used. In suffix links 69.23% of times the function rescan was used.

[1]Measure unit in DNA genomes.

V. Conclusion

Traverse of suffix tree and use of suffix links are bottlenecks for a better processor performance. In suffix trees larger than the Last Level of Cache, the performance is dictated by the number of cache lines loaded from the memory, and the hardware features available to hide latency due to potential TLB and Last Level of Cache misses.

Acknowledgment

This work was supported by grant from "Ejecución eficiente de aplicaciones multidisciplinares: nuevos desafíos en la era multi/many core", with reference TIN2011-28689-C02-01.

References

[1] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0521585198

[2] W. Chang and E. Lawler, "Sublinear expected time approximate string matching and biological applications," 1991. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1991/CSD-91-654.pdf

[3] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," vol. 29, no. February, pp. 1149–1171, 1999.