Suffix trees: How to do Google Search in Bioinformatics?

By Ananth Kalyanaraman, Washington Statee University

Rubric for Student Assessment

This module can be graded out of 100%, which allows for easy conversion to a different points-based grade or a percent-based grade depending on the instructor's preference. For instance, the instructor can weight the homework problems (exercises and review questions) at 50% and the the programming project at 50%.

Review Questions and Exercises (weighted at 50%)

Note: Instead of absolute points, each problem below is attached to one of three difficulty levels:

A-level:	Advanced (hardest)
I-level:	Intermediate level
B-level:	Beginner level (very easy)

L1) Lookup table for "mississippi" using k=2:
(B-level)

1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i

Mapping characters {i,m,p,s} to integer values {0,1,2,3} respectively, the lookup table will have 4^k (=16) entries as follows:

Lookup table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ii	im	ip	is	mi	mm	mp	ms	Pi	pm	pp	ps	Si	sm	sp	ss
			2	1								4			3
		8	5					10		9		7			6

L2) Pattern matching using lookup tables:
(I-level)

E.g., query q="issi" and text T= "ississippi" should return text indices 2 and 5 as its answer.

Pseudocode for the pattern matching algorithm:

```
0. Assume that the lookup table has already been built for the text T.
   Also assume that the string indexing starts from 1,
   and that the lookup table indexing starts from 0.
   Let n denote the length of the text T, and m denote the length of the query q.
   And k denotes the k-mer length.

1. LinkedList *PostitionList = NULL; // initialize an empty linked list of integers
2. int i=1;
3. QueryWindow = q[i..i+k-1];
4. idx = LookupTableIndexOf(QueryWindow);
   // returns the index of the substring denoted by QueryWindow
5. Add all indices in the list of LookupTable[idx] to PositionList.

6. while(i< (m-k+1) ) {
     a. If PositionList is empty, then return "Pattern not found" and exit.
     b. i=i+k;
     c. QueryWindow = q[i..i+k-1];
     d. idx = LookupTableIndexOf(QueryWindow);
     e. Let P_idx represent the set of indices in LookupTable[idx] such that:
        j ∈ P_idx if and only if, j=r+k, for some r ∈PositionList;
     f. If P_idx is empty, then return "Pattern not found" and exit.
     g. Retain only those indices r in PositionList for which r+k ∈P_idx.
        Remove all other indices.
     h. Replace all retained indices r in PositionList by r+k.

   } // end while

7. Manually check which of the positions left over in PositionList continue to
   spell out the remaining k-1 characters of the query.
8. Report the starting indices of those positions as answer.
```

Run-time not including the cost for lookup table construction:
O(m + sum of the sizes of PositionList over all iterations)

Memory = $O(|\Sigma|^k + N)$ for the lookup table + O(m) for the query
, where Σ is the alphabet
Note: the space required to store PositionList will be dominated by the space taken by the lookup table

L3) Algorithm to compute K-distance measure:

(B-level)

Input: strings s1 and s2;
Output: K-distance(s1,s2) which is equal to |K(s1)\K(s2)| + |K(s2)\K(s1)|

```
// To compute |K(s1)\K(s2)|:
    1. int dist=0;
    2. Build a lookup table for s2;
    3. Slide a window of length k over s1, and for each window check if that k-mer
       is present in the lookup table for s2 or not. If it is not present, then dist++;
    4. The value of dist is equal to |K(s1)\K(s2)|
```

To compute |K(s2)\K(s1)| follow a similar procedure, with just the roles of s1 and s2 reversed.

Finally, add both values to report the final K-distance.

Run-time: O(|s1|+|s2| + $|\Sigma|^k$)
Memory: O(|s1|+|s2| + $|\Sigma|^k$)

<u>L4) Lookup table construction algorithm:</u>
(I-level)

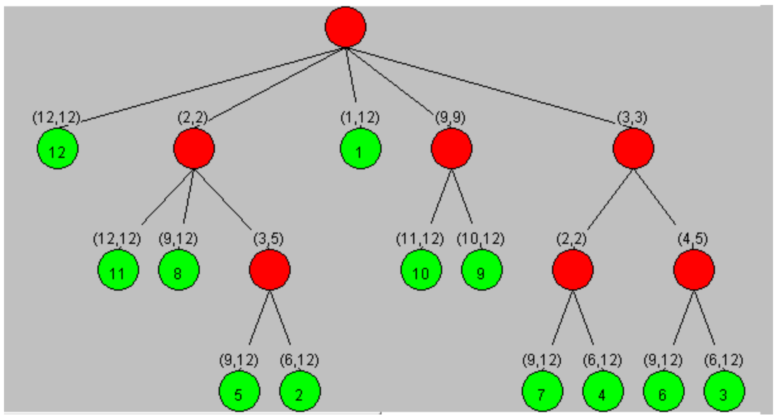
Assume Σ={a,c,g,t}; k is some value > 1
T is the text string of length N, indexed from 0 to N-1
Our algorithm will map characters {a,c,g,t} to {0,1,2,3} integer values
This implies that a k-mer will be interpreted as a k-bit base-4 number.
E.g., map (“caa”) = 16, and map (“cag”) = 18.

(this pseudocode is written closely to align with C syntax)

```
    0. int SIZE=|Σ|^k
    1. int ** LT= (int **) malloc(sizeof(int *)*SIZE;          // init lookup table

    2. for (int i=0;i<SIZE;i++) LT[i]=NULL;

    3. First insert T[0..k-1] into LT in a brute-force manner
       i.e., append index 0 to the list pointed to by LT[map(T[0..k-1])].
       Note: This step will take O(k) time.

    4. int PrevMapValue = map(T[0..k-1]);

    5. i=1;

//     To insert all remaining k-mers follow these steps:
```

```
6. while(i<(N-k+1)) {
      a. int NewMapValue = (PrevMapValue-map(T[i-1]))*4 + map(T[i+k-1]);
         // because, T[i+k-1] is the new character that got added to window,
         and  T[i-1] is the character that got eliminated from the window.
         Note that this means it will only now take constant time to insert each
         k-mer from here on.
      a. Append index i to the list pointed  to by LT[NewMapValue];
      b. i++;
   }
```

L5) Suffix tree for string "mississippi$"
(B-level)



The above picture was auto-generated using the animation in
http://pauillac.inria.fr/~quercia/documents-info/Luminy-98/albert/JAVA+html/SuffixTreeGrow.html
The figure follows the convention discussed in the slides, which is to represent each edge as a pair of integers (starting index, ending index) of a substring of the input string that corresponds to the edge label.
Green nodes represent leaves.
Red nodes represent internal nodes.

L6)
(B-level)

If the input string is a concatenation of the same character n times, then the corresponding suffix tree will have n+1 leaves and n-1 internal nodes.
e.g., s = "aaaaaaaaa"

L7)
(B-level)

Same as above. E.g., s=” aaaaaaaaa”.

L8)
(I/A-level)

GST construction in linear time:

Since we know that the suffix tree for a single string s of length n can be constructed in O(n) time using McCreight or Weiner or Ukkonen algorithms, we can treat the problem of constructing the GST for a set of k strings {s1, s2, … sk} to be that of incrementally building the GST starting with the suffix tree of s1, and then superimpose the ST for s2 on top of the tree already built for s1, and ST for s3 on top of the tree already built for {s1, s2} and so on.

After k iterations we will have the GST for {s1, s2, … sk}.

This procedure can reuse the nodes already put in place by previous iterations, and so the overall runtime of the method cannot exceed O(N), if N is the sum of the lengths of all the k input strings.

L9)
(I-level)

LCS algorithm for more than 2 input strings:

0. First build the GST for all the k input strings, where k>2.
1. Traverse the tree in post-order (i.e., all children before parent), and in the process keep track of the number of *distinct* strings whose suffixes are represented under each internal node along the way.
2. Let u denote the deepest such internal node (i.e., the one with the largest string depth) with all k distinct strings represented under its subtree. Note: it is guaranteed that such a node exists (in the worst-case it is the root).
3. The LCS of all the input strings is then the path-label (u).

L10)
(B-level)

Suffix array and LCP array for “mississippi”:

Assume that $ to be the lexicographically smallest.
Also, assume the convention that LCP[i] = longest common prefix length between suffixes SA[i] and SA[i+1].

	1	2	3	4	5	6	7	8	9	10	11	12

String:	m	i	s	s	i	s	S	i	p	p	i	$
Suffix Array:	12	11	8	5	2	1	10	9	7	4	6	3
LCP Array:	0	1	1	4	0	0	1	0	2	1	3	Not defined

L11)
(A-level)

Ideas for building suffix trees on distributed memory parallel computers:

There are multiple possible answers to this open ended question. Here is one idea that could work. This idea uses a lookup table-based approach to first partition the suffix tree and then build it in parallel. In practice this algorithm is better suited to build the GST of numerous small-length strings.

1. All processors read O(N/p) of the input string.
2. Each processor scans its local input segment and populates a local lookup table using fixed size k-mers. Call each bin of this lookup table a “bucket”.
3. Using all-to-all communications, the buckets are redistributed such that every processor receives a sum total of O(N/p) suffixes, and no bucket is split across processor boundaries. Obviously this step assumes that every bucket in your lookup table is bounded by O(N/p) suffixes, which is likely to hold true for most practical inputs (at least in bioinformatics).
4. Each processor now has a set of local buckets. Each bucket contains suffixes of the input that should go into the same subtree of the suffix tree, where the subtree is rooted by an internal node of string-depth k.
5. Therefore, each processor can now iterate over its buckets, and construct the subtree for each of its buckets using brute-force comparison of the characters of the corresponding suffixes. To do this, it is required that the processor has in its local memory all suffixes of that bucket. This can be either done through disk I/O (which could highly inefficient due to the randomness of the suffix indices) or through communication (in which case each processor will need to intersperse the local tree computation with communication calls to get the next batch of strings).

If applied to the building of a GST of k strings, each of which is roughly l in length, it can be shown that the above algorithm’s computational cost can be bounded by $O(k*l^2/p)$, and the communication cost expected to be O(k*l/p) in bytes transfer although over multiple iterations (therefore implying a large latency which could potentially be masked by doing pre-fetches).

Programming Project (weighted at 50%)

It is encouraged that each student completes the programming project in teams of size 2 or 3 each. The project can be graded for 100 points using the following breakdown:

a) Algorithm design – 30 pts
b) Naïve algorithm implementation correctness – 20 pts
c) Suffix tree search implementation correctness – 20 pts
d) Performance testing and justification – 25 pts
e) Design simplicity and coding documentation – 5 pts