Applications of Suffix Trees

Dr. Amar Mukherjee
CAP 5937 – ST: Bioinformatics
University of central Florida

7.1 Exact String Matching

- Three important variants:
  - Both P (|P|=n) and T (|T|=m) are known:
    - Suffix tree method achieves same worst-case bound O(n+m) as KMP or BM.
  - T is fixed and build suffix tree, then P is input:
    - k: number of occurrences of P
    - Using suffix tree: O(n+k)
    - In contrast (preprocess P): O(n+m) for any single P
  - P is fixed, then T is input
    - Select KMP or BM rather than suffix tree.

7.2 Exact Set Matching

- Both Aho-Corasick and Suffix methods find all occurrences of P in T in O(n+m+k). But have preference by case.
- Comparison:
  - AC: build keyword tree: size O(n), time O(n).
  - When set of patterns is larger than T, suffix tree approach uses less space, but more time to search.
    - E.g. molecular biology, where pattern library is large.

  - When total size of patterns is smaller than T, AC method use less space. But suffix tree uses less time.
  - Neither method is superior in time and space.
- One case where suffix tree is better, see Application 8.
  - Time/space trade-off remains, but suffix tree can be used for chosen time/space combinations, whereas no choice for keyword tree.

7.3 Substring Problem for a Database of Patterns

- The most interesting version:
  - A set of string, or a database, is known and fixed. A sequence of strings will be presented. For each presented string S, find all the strings in the database containing S as a substring.
- The total length of all the strings, m, in the database is assumed to be large.
- In the context of genomic DNA data, the problem of finding substring cannot be solved by exact set matching.

- Suffix tree solution:
  - A generalized suffix tree is built for database in O(m) time and O(m) space.
  - Any single string S of length n can be found or declared not to be there in O(n) time.
  - If S matches a path in the tree.
    - A full string is in S iff matching path reaches a leaf when last symbol of S is examined.
    - Find all occurrences containing S as substring in O(n+k) time by traversing subtree below where S is found.

7.4. Longest Common Substring for Two Strings

- Different from ***Longest Common Subsequence*** problem.
- E.g. S1: *superiorcalifornialivers* S2: *sealiver*
  - Longest common substring: *alive*
- Longest common substring of two strings can be found in linear time using a generalized suffix tree.
  - Find the node with the greatest depth that is marked both 1 and 2.
  - Linear construction time
  - Node marking and calculation of string depth can be done by standard linear tree traversal methods.
- 1970: Don Knuth conjectured that a linear time algorithm would be impossible.

7.5 Recognizing DNA Contamination

- Given a string S1(the newly isolated and sequenced string of DNA) and a known string S2 (the combined sources of possible contamination), find all substrings of S2 that occur in S1 and are longer than some given length *l*. These substrings are candidates of unwanted pieces of S2 that have contaminated the desired DNA string.

Finding common substrings

- Can be solved in linear time by extending the longest common substring of two strings.
- Build a generalized suffix tree for S1 and S2.
- Mark each internal node that has in its subtree a leaf representing a suffix of S1 and also a leaf representing a suffix of S2.
- Report all marked nodes that have string depth of l or greater.

7.6 Common Substrings of more than two sequences

Given a set of strings, find substrings that are common to a large number of those strings.

Formal statement:

Given: *K* strings whose lengths sum to *n*

- 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*.

Example:

Strings - {*sandollar*, *sandlot*, *handler*, *grand*, *pantry*}

l(2) – 4, *l*(3) – 3, *l*(4) – 3, *l*(5) – 2

7.6: Linear-time Solution

- Build generalized suffix tree *T* for all the input strings.
- For every internal node *v* of *T*, define *c*(*v*) to be the number of distinct string identifiers that appear at the leaves in the subtree of *v.*[It is easy to compute the number of leaf nodes under v; but computing c(v) is complicated by the fact that more than two leaves may have same identifier.]
- *l(k)* is the depth of the deepest node *v* such that *c*(*v*) $\geq k$

7.6: Complexity

- Counting the number of leaves under an internal node *v* does not give *c*(*v*).
- Therefore, each internal node *v* maintains a *K*-length bit vector. Bit *i* in the vector is set to 1 if there is at least one leaf under *v* belonging to string *i.*
- The bit vector for an internal node *v* can be obtained **OR**ing the bit-vectors of all the children of *v.*
- Since there are O(*n*) edges in the tree, the time needed will be O(*Kn*)
- There is a O(n) solution. See Chapter 9.

7.10 All-pairs Suffix-Prefix Matching

Definition:

Given two strings S_i and S_j, any suffix of S_i that matches a prefix of S_j is called a suffix-prefix match of S_i, S_j.

Given a collection of strings $S = S_1, S_2, \ldots S_k$, the **all-pairs suffix-prefix problem** is the problem of finding, for each ordered pair S_i, S_j in S, the *longest* suffix-prefix match of S_i, S_j.

Motivation:

- Approximate methods for the *shortest superstring* problem.

7.10: linear time solution

- We call an edge *terminal edge* if it is labeled with only a string termination symbol.
- Solution:
  - Build a generalized suffix tree $T(S)$ for the k strings in S.
  - Build list L(v) for each internal node v
    - $L(v)$ contains index i if a terminal edge labeled i is incident on v
  - The deepest node v on the path to leaf j such that $i \in L(v)$ identifies that longest match between a suffix of S_i and a prefix of S_i.

7.10 (continued…)

- Traverse T(S) in a depth-first manner
- Maintain k stacks, one for each string
- When a node v is reached in forward direction, push v on to the ith stack, for each $i \in$L(v).
- When a leaf j corresponding to the entire string S_j is reached, scan the k stacks and record the current top of each stack.
- When the depth-first traversal backs up past a node v, pop the top of any stack whose index is in L(v)
- Complexity:O(m+k^2)

Importance of Repetitive Structures in molecular strings

- Over 50% of human genome consists of repeats.
- Complimentary palindromes regulate transcription (by forming hair-pin loops)
- Clustered genes that code for similar proteins
- Pseudogenes
- Restriction enzyme cutting sites
- Tandem repeats and tandem arrays

Uses of repetitive structures

- Genetic mapping
  - Requires the identification of markers that are highly variable between individuals
  - Tandem arrays can be used as such markers
    - The number of repeats in a tandem array varies from individual to individual
  - Micro satellite markers – tandem repeats of very short strings

Finding all maximal repetitive structures

- Defining repeats is crucial
  - A string consisting of n copies of the same character will have $O(n^4)$ pairs of repeats
- *Maximal repeated pair* in a given string S:

  "A pair of identical substrings α and β in S such that extending α and β in either direction would destroy the equality of the two strings"
  - i.e, occurrences $x\alpha y$ and $v\beta w$, $x \neq v$ and $y \neq w$, where x,y,v and w are characters will give a maximal repeated pair α and β .
  - Represented by a triple (p_1, p_2, n'), where p_1 and p_2 give the staring positions and n' gives the length.
  - $R(S)$ – the set of all triples describing maximal pairs in S

Maximal repeated pairs

Example:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	x	a	b	c	y	i	i	i	z	a	b	c	q	a	b	c	y	r	x	a	r

Maximal pairs:

(2,10,3) – xabcyiiizabcqabcyrxar

(2,14,4) – xabcyiiizabcqabcyrxar

(10,14,3) – xabcyiiizabcqabcyrxar

(6,7,2) – xabcyiiizabcqabcyrxar

- Allows overlaps!

More definitions

- *Maximal repeat* :

  "A substring of *S* that occurs in a maximal pair in *S*".

  Example: abc in *S* = xabcyiiizabcqabcyrxar

*Note: There can be numerous **maximal repeated pair**s, but there can be only a limited number of **maximal repeat**s.*

- *Supermaximal repeat*

  "A maximal repeat that never occurs as a substring of any other maximal repeat"

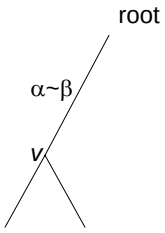
  Example: *abcy* in *S* = *xabcyiiizabcqabcyrxar*

Using suffix trees to find maximal repeats

Lemma 7.12.1:

"If a string α is a maximal repeat in *S*, then α will be the path-label of an internal node *v* in *T*(*S*)"

Proof: Gusfield, page 144



Theorem 7.12.1:

"There can be at most *n* maximal repeats in any string of length *n*"

- Why?

Finding maximal repeats: Definitions

left character

- For each position *i* in *S*, *S*(*i*-1) is called the left character *i*.
- *Left character* of a leaf in *T*(*S*) is the left character of the suffix position represented by that leaf.

left diverse

- An internal node *v* in *T*(*S*) is called *left-diverse* if at least two leaves in *v*'s subtree have different left characters.

Theorem:

"The string α labeling the path to a node *v* of *T(S)* is a *maximal repeat* **if and only if** *v* is *left diverse*"

Finding left diverse nodes in linear time

- For each internal node *v,* the algorithm either:
  - Records that *v* is left diverse, or:
  - Records the character *left*(*v*) that is the left character of every leaf in *v*'s subtree.
- Starts by recording the left character of each leaf in *T*(*S*)
- Processes the internal nodes in *T*(*S*) bottom-up
  - If any child of *v* is left diverse, then v is left diverse
  - If none of the children are left diverse, then it examines the recorded characters of all the children:
    - If all of the characters are *x*, then the left character of *v* is *x*
    - If all of them are not *x*, then *v* is left-diverse

Finding all maximal repeats in linear time

- Path labels to all internal nodes in *T*(*S*) that are left diverse

  - Simply delete all internal nodes that are not left diverse!

Finding Supermaximal repeats in linear time

Near-supermaximal repeat

A substring α is near-supermaximal repeat if α is a maximal repeat that occurs at least once in a location where it is not contained in another maximal repeat

Example:

- in aαbxαyaαbxαb, α is neither supermaximal nor near-supermaximal
- *abc* in *xabcyiiizabcqabcyrxar* is near-supermaximal

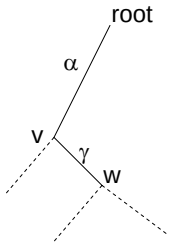
Note:

The set of near-supermaximal repeats is not the same as the set of maximal repeats that are not super-maximal

Finding super-maximal repeats

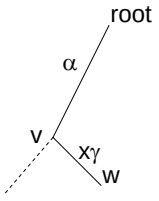
Lemma 7.12.2:

If *v* and *w* are internal nodes in *T*(*S*) such that *w* is a child of *v*, and if α is the path-label of *v*, then none of the occurrences of α specified the leaves in the subtree under *w* witness the near-supermaximality of α.



Lemma 7.12.3:

Let *w* be a leaf representing a suffix starting at position *i*, and let *w* be a child of *v*. Then, the occurrence of α at position *i* witnesses the near-supermaximality of α if and only if x is the left character of no other leaf below *v*.



Finding supermaximal repeats in linear time

Theorem 7.12.4

> "A left diverse internal node *v* represents a near-supermaximal repeat α if and only if one of *v*'s children is a leaf, and its left-character is the left character of no other leaf below *v.*"

> "A left diverse internal node *v* represents a supermaximal repeat α if and only if all of v's children are leaves, and each has a distinct left character"

Degree of near-supermaximality: The fraction of occurrence of α that witness its near super-maximality

7.13: Circular string linearization

- Problem
  - Cut a circular string S so that the resulting linear string is lexically smallest of all the *n* possible linear strings created by cutting *S*.
- Solution
  - Cut S at an arbitrary position to give a linear string *L*.
  - Build the suffix tree *T* for the string LL$, where $ is lexically greater than any character in L.
  - Traverse tree *T*
    - At every node, take the lexically smallest edge
    - Traverse until the traversed has string-depth of *n.*
    - Any leaf *l* at that point can be used to cut the string