Suffix Trees

Problem

- Given a set of strings $S$, preprocess it into a data structure so that a query string $s$ can be found in $S$ quickly.
- Algorithm 1. Input $S$, output $D$ (a data structure).
- Algorithm 2. Input $s$, output: pointers to all occurrences of the string $s$ in $S$ or “not found”.

Standard Trie

- The standard trie for a set of strings $S$ is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of $S$
- Example: standard trie for the set of strings $S = \{\text{bear, bell, bid, bull, buy, sell, stock, stop}\}$

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - $n$ is the total size of the strings in $S$,
  - $m$ is the size of the string parameter of the operation,
  - $d$ is the size of the alphabet.

Word Matching with a Trie

- We insert the words of the text into a trie
• Each leaf stores the occurrences of the associated word in the text

Variations
• Label edges instead of nodes
• Sentinel $\$
  – useful to avoid nested words and
  – makes one-to-one correspondence between words and leaves

Compressed Trie
• A compressed trie has internal nodes of degree at least two
• It is obtained from standard trie by compressing chains of “redundant” nodes
Compact representation of a compressed trie for an array of strings:

- Stores at the nodes ranges of indices instead of substrings
- Uses $O(s)$ space, where $s$ is the number of strings in the array
- Serves as an auxiliary index structure

The suffix tree of a string $X$ is the compressed trie of all the suffixes of $X$. 

3
Compact representation of the suffix trie for a string $X$ of size $n$ from an alphabet of size $d$.

- Uses $O(n)$ space
- Supports arbitrary pattern matching queries in $X$ in $O(dm)$ time, where $m$ is the size of the pattern

**Store**

- Edge label $(i, j)$ for every node
- Pointers to *child* and *sibling*

**PATRICIA (PAT TREE)**
Practical Algorithm To Retrieve Information Coded In Alphabets - introduced by D.R. Morrison in October 1968
Collapse all unary nodes

*Construction of suffix trees*
• Theorem [Weiner, 1973]: $T(S')$ can be constructed in $O(n)$ time.

• There are two practical algorithms that construct the suffix tree in linear time: McCreight (1976) and Ukkonen (1993).

• A simpler algorithm is the WOTD (write-only, top-down) algorithm:

1. Let $X$ be the set of all suffixes of $S$.
2. Sort the suffixes in $X$ according to their first character.
3. For each group $X(c)$ ($c$ is a character):
   (i) if $X(c)$ is a singleton, create a leaf;
   (ii) otherwise, find the longest common prefix of the suffixes in $X(c)$, create an internal node, and recursively continue with Step 2, $X$ being the set of remaining suffixes from $X(c)$ after splitting off the longest common prefix.

• Analysis: $O(n^2)$ worst-case time, $O(n \log n)$ expected time, $O(n)$ space.

Example

Off-line and on-line string matching

• Off-line: KMP algorithm and Z-algorithm

• $O(n + m)$ search time. No preprocessing

BR On-line algorithms using suffix trees:
Weiner, McCreight, and Ukkonen.
Preprocessing in $O(n)$ time, searching in $O(m)$ time.

• Donald Knuth regarded Weiners paper as the most important discovery in algorithms in 1973 (In 1970 Knuth conjectured that a linear-time algorithm would be impossible).

Ukkonen algorithm
Constructs an implicit suffix tree $T_i$ for each prefix $S[1..i]$ of $S$.

Starts from $T_1$ and increments $i$ by 1 until $T_m$ is built. Extension.

Naïve approach takes cubic time.

**Wiener algorithm**

Starts with the string $\$.

Adds one suffix at a time in increasing order of length.

Naïve approach takes $O(m)$ time for the insertion. Look at example [Wiener algorithm]

**Applications of suffix trees**

- On-line exact string matching
- Substring search in a database
- Longest common substring of two strings
- Recognizing DNA contamination
- Common substring of more than 2 strings
- Matching statistics
- Repeats

*Search in database* Preprocess a database of strings with total length $m$ so that all occurrences of a query string can be found quickly.

- Generalized suffix tree.

- The space is $O(m)$. The query time is $O(n)$ where $n$ is the length of query string.

*Generalized Suffix Tree*

Suffix tree for the strings ABAB and BABA:
**Longest common substring of 2 strings**

Problem: Given two strings $S_1$ and $S_2$, find the longest string $s$ that is a substring of $S_1$ and $S_2$.

Example: ATAGTTAGGTAAATG and TAGTATTAGGTATT

- Generalized suffix tree
- Node labels:
  1. if all leaves below have suffixes of $S_1$ only
  2. if all leaves below have suffixes of $S_2$ only
  3. if there is a leaf below with suffix of $S_1$ and there is a leaf with suffix of $S_2$

- Labeling in $O(|S_1| + |S_2|)$ time

**Recognizing DNA contamination**

- Important Problem: Given a string $S_1$ (newly isolated and sequenced string of DNA) and a known string $S_2$ (the combined sources of possible contamination), find all substrings of $S_2$ that occur in $S_1$ and that are longer than some given length $l$.

- These substrings are candidates for unwanted pieces of $S_2$ that have contaminated the desired DNA string.
- Generalized suffix tree with labeling.
- Traverse tree and report nodes labeled 3 and whose strings have length at least $l$.

**Common substring of > 2 strings**

- Suppose we have $K$ 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: one,done,none,moon,nonempty

- $k$ $l(k)$ one substring
  - 2 4 none
  - 3 3 one
  - 4 3 one
  - 5 2 on

- Surprisingly, it can be found in linear time.
• $O(Kn)$ solution: Generalized suffix tree. Each string is given a distinct termination symbol. Let $C(v)$ be the number of distinct termination symbols below $v$. Compute $V(k)$, longest substring of exactly $k$ strings, by traversing tree. Then find $l(k)$ for all $k$.

**Matching statistics**

• $T$ is a text of size $m$, $P$ is a pattern of size $n$.
  Reverse problem: construct suffix tree for $P$ to search substrings of $T$ in $P$.
• $ms(i)$ matching statistics, the length of the longest prefix of $T[i..]$ that matches a substring in $P$.
• Example: $T$=mississippi, $P$=tissue, $ms(0) = 0, ms(1) = 3$
• $O(mn)$ time: suffix tree for $P$ and check $T[i..i+n]$ for each $i$.
• Better solution using suffix links. If a node $v$ has path-label $xs$ where $x$ is a character, then suffix link $s(v)$ points to a node with path-label $s$. Suffix links can be found in linear time.

**Suffix links**

| ACACACAC | $\$ |
| AC | $\$ |
| AC | $\$ |
| AC | $\$ |
| ACS | $\$ |

**More applications of suffix trees**

• DNA of eukaryotes is highly repetitive (Cell or organism with membrane-bound nucleus. Eukaryotes include all organisms except viruses, bacteria, and blue-green algae)
• Repeat regions are rapidly changing hot spots in evolution.
• Vast literature on repetitive structures and their hypothesized functional and evolutionary roles: ALUs, SINEs, LINEs, microsatellites, minisatellites, ...
• Repeats are involved in several biological mechanisms, including genetically inherited diseases. e.g. Huntington’s disease
• Repeats tend to confuse sequence analysis programs and hence should be masked in a preprocessing step.

Repeats are very important when studying genomic DNA.

**Repeats**

Definitions:
• A pair of substrings $R = (S[i_1, j_1], S[i_2, j_2])$ is called a *repeat*.
• *exact repeat* if $S[i_1, j_1] = S[i_2, j_2]$
  Example TATGGTAATGGGT (may overlap)
- **k-mismatch repeat** if there are \(k\) mismatches between \(S[i_1, j_1]\) and \(S[i_2, j_2]\)
  
  \[
  \text{TATTGGTAATAGGGT}
  \]

- **k-differences repeat** if there are \(k\) differences (mismatches, insertions, deletions) between \(S[i_1, j_1]\) and \(S[i_2, j_2]\)
  
  \[
  \text{TATTAGGGTAATATTGGT}
  \]

Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in \(S\) in linear time.

Idea

- consider string \(S\) and its suffix tree \(T(S)\).
- repeated substrings of \(S\) correspond to internal locations in \(T(S)\).
- leaf numbers tell us positions where substrings occur.

Analysis: \(O(n + z)\) time and \(O(n)\) space where \(z = \text{|output|}\)

Finding exact repeats

- How to list the leaves below a node in time proportional to their number?
- Depth-first search DFS (BFS is OK).
- Time is proportional to the number of traversed edges. Every edge is traversed at most two times. The number of edges is at most 2 times the number of leaves.
Idea

- **Right-maximality**
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)

- **Left-maximality**
  - keep lists for the different left-characters
  - report only pairs from different lists

Analysis: $O(n + z)$ time and $O(n)$ space where $z = \vert \text{output} \vert$

**Maximal repeats**

- What are maximal repeats? What is the output?
- **Output1**: $(i, j, k)$ for substrings $S[i..i + k - 1]$ and $S[j..j + k - 1]$
- **Output2**: string $s$ (like ATAG..A) if $s$ appeared (at least) two times as substring of $S$ (maximal repeat!)
- Book uses output2 (and defines max repeat accordingly).
- Linear time for version 1 means $O(1)$ amortized time to report a triple. The size is $O(n^3)$. Is there a better bound?
- Version 2. Explicit output has $\Theta(n^2)$ size in worst case.
- Linear time for version 2 means compact representation of maximal repeats. Idea: a node $v$ of suffix tree is *left diverse* if at least two leaves below have different characters left to their suffixes.
- Green letter is left character of suffix. Red number is start index of suffix.
- There are 2 internal nodes (in circles), node A and TA.
- Node A is left diverse since it has T and D below.
- Node A creates 2 max repeats: A (4,6) (2,6).
- Node TA is left diverse and creates max repeat TA (3,1).
- Note that A (4,2) is not max repeat since 4 and 2 have the same left character T.
**k-mismatch problem**

- Given a pattern $P$ of length $n$, a text $T$ of length $m$, and a parameter $k$, a *k-mismatch* of $P$ is a $n$-length substring of $T$ that matches at least $n - k$ characters of $P$.

- Examples:
  - $T=\text{whatever}$, $P=\text{whenever}$, $k=2$, k-mismatch is *whatever*.
  - $T=\text{whowhowho}$, $P=\text{what}$, 2-mismatch is *whow*.

- Algorithm: counting mismatches in suffix tree.

**Longest Common Extension (LCE)**

- Preprocess strings $S_1$ and $S_2$ such that the following queries can be computed in $O(1)$ time each:
  - Given index pair $(i, j)$, find the length of the longest substring of $S_1$ starting at position $i$ that matches a substring of $S_2$ starting at position $j$.

- Example
  - $S_1: \ldots \text{abcdzzz} \ldots$
  - $S_2: \ldots \text{abcdefg} \ldots$

- Runtime $O(km)$.

Suppose that LCE query takes $O(1)$ time.

Solution of k-mismatch problem with runtime $O(km)$:

- For each $i = 1$ to $m - n + 1$ test if $P$ aligned at $T[i..m]$ is a $k$-mismatch as follows.
  1. $\text{count} = 0$; $i = i$; $j = 1$
  2. $l = \text{LCE}(I, j)$ // longest common extension of $T[i..m]$ and $P[j..m]$
  3. If $j + l = n + 1$ then "$k$-mismatch at position $i$" (actually only $\text{count}$ mismatches occur)
  4. If $\text{count} \leq k$ then
     $\text{count} = \text{count} + 1$; $j = j + l + 1$; $I = I + l + 1$
     go to step 2

**Lowest Common Ancestor (LCA)**

- Preprocess a tree $T$ such that the following queries can be computed fast:
  - Given vertices $u$ and $v$, find their lowest common ancestor LCA.

- Harel and Tarjan 1984. Preprocess in $O(n)$ time and answer query in $O(1)$ time.


**Longest Common Extension**

Preprocess in $O(|S_1| + |S_2|)$ time
• Build generalized suffix tree $T$ for $S_1$ and $S_2$.
• Preprocess $T$ for constant-time LCA queries.
• Compute string-depth of every node.

Answer query $(i, j)$ in $O(1)$ time

• Find LCA node $v$ of leaves corresponding to suffix $i$ of $S_1$ and suffix $j$ of $S_2$.
• Return string-depth($v$).