Suffix Tree and Array
Things To Study

• So far we learned how to find “approximate” matches – the alignments. And they are difficult.
• Finding exact matches are much easier. Suffix tree and array are two data structures for this purpose.
• Suffix Tree
  • Data structure
  • A few examples of using suffix tree to solve practical problems.
• Suffix Array
  • Data structure
  • The skew algorithm for constructing suffix array.
A Little History

• 1973, Weiner introduced the concept of suffix tree (position tree), which Donald Knuth subsequently characterized as "Algorithm of the Year 1973".

• 1990, Gene Myers and Udi Manber proposed suffix array.
  • Gene Myers: former VP Informatics Research at Celera Genomics
  • Udi Manber: VP engineering, Google.

• 1992, Gonnet, Baeza-Yates & Snider independently discovered suffix array (called PAT array).
  • Gaston Gonnet: cofounders Maplesoft and OpenText.
  • Baeza-Yates: VP for Yahoo! Europe and Latin America.
What Can Suffix Tree Do?

• Build an index structure of the Internet (the text).
  • Given a length-k string (the query), in O(k) time, find the first occurrence of the string in the text.
  • Cannot afford more than linear space.

• Find the longest shared string between human and mouse genomes
  • Each has $3 \times 10^9$ base pairs.
  • Cannot afford $3 \times 10^9 \times 3 \times 10^9$ comparisons.
  • This can be done in linear time!
• Suffix trees can do many string operations you might think are very hard, in linear time.

• It surprised everybody when it was first invented by Weiner in 1973. Knuth is claimed to have called the result “the algorithm of 1973”.

• Easier suffix tree construction algorithms have been developed since then.
- Here is the suffix tree for GAAGAT$.

- An edge is labelled with a substring of the original string.
- A **node**’s label is the concatenation of all edge labels for the path leading to that node.
- The path from the root, $r$, to any leaf $x$ is a suffix of the string $S$.
- Suppose there is a special “end-of-string” character, each suffix will end at the leaf.
- Each internal node has at least 2 children.
Some basic facts

1) If we’re really going to include all of the suffixes, then the last letter must be unique (as in the example). Else, suffix may end at an internal node!
   • We solve this problem by adding a unique letter at end.

2) We’re going to assume throughout that the alphabet size is constant. Thus each internal node has constant number of child nodes.

3) The suffix tree is unique except for ordering of child nodes.
Application I. Search for a substring.

• Any substring of S is a **prefix** of a **suffix**.

• Example of using this: Is the string \( x \) a substring of S?
  • Start at the root, and follow paths labelled by the characters of \( x \). If you can get to the end of \( x \), then yes, it is.
Each edge doesn’t need to be labelled with a string, but just with starting and ending in the sequence.

This is the same suffix tree as before, but in linear space.
How to construct a suffix tree?

• We’ll start with a quadratic-time algorithm and talk about the uses of suffix trees.

• Then we’ll finish with some applications.

• The linear-time construction isn’t easy on the first time. Chapters 5-8 of the following book are excellent. It’s strongly recommended that you look at them.

  • Dan Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology
A simple algorithm

- Given: A string $S$ of length $m$ over a finite alphabet. The last character of $S$ is a unique $\$ \text{ character.}$
- We’ll build the suffix tree from right to left.
  - $S[m..m], S[m-1..m], S[m-2..m], \ldots$
- Begin with this tree:

```
  $\$
```

- Then, for $i = m \text{ downto } 1$:
- Follow the letters of $S[i\ldots m]$ along the edges of the tree $T$.
- When we reach a point where no path exists, break the current edge and add a new edge for what is left.
One round

• Suppose the following is the suffix tree for GAAGA$, add another suffix AGAAGA$.

• First, follow the edges for A and for GA from the root.
• Then split after the A since the only path in the tree is for $, and we have an A, instead.
• Add a new edge for AGA$.
New tree

- This yields this new tree for AGAAGA$.

- Note: There are 3 places where “AGA$” is sitting off by itself. Maybe this repetition might be something to think about.
Obvious runtime

- Time complexity: $O(m^2)$, since it’s doing $O(m)$ work in each phase.
- There are other non-trivial construction algorithms with time complexity $O(m)$.
- We will not study these linear time construction algorithms. Instead, we focus on the applications.
Application II: Longest Common Substring

• What’s the longest substring common to both $S_1$ and $S_2$?
• Build a suffix tree for $S=S_1#S_2\$, where $#$ and $\$ are unique characters.
• All suffixes of $S_1$ ends with an edge including $#S_2$.
• Color all nodes $v$ of the tree:
  • red if $v$’s label is a substring of $S_1$
  • blue if it’s a substring of $S_2$
  • purple if it’s a substring of both
• We want the lowest purple node.
• Linear time, linear space.
Application III: Maximal repeats

• Given: String S.
• A maximal repeat is a substring \( t \) that occurs \( k > 1 \) times in S, and if \( t \) is extended to left or right, it will occur less than \( k \) times.
• Find all maximal repeats.
• Example:
• AAGATATGATAGGAT$
• Maximal repeats include:
• GATA (as in positions 3 and 8):
• AAGATATGATAGTGAT$
• GAT (as in position 3, 8 and 13):
• AAGATATGATAGTGAT$
• And so on. (Try to find another one)
Suffix trees are magic…

• These can be identified in $O(m)$ time.
• We’ll prove this slowly.

**Lemma 1:** If $\alpha$ is a maximal repeat, then there is an internal node with that label.

• **Proof:** Since $\alpha$ is a maximal repeat, then twice when it occurs, it’s followed by different letters. Hence, it’s a bifurcation point in the tree.

**Lemma 2:** There are at most $m$ maximal repeats.

• **Proof:** The tree has $m+1$ leaves, and hence at most $m$ internal nodes.

• OK, so which internal nodes correspond to maximal repeats?
Finding maximal repeats

- Consider a node \( x \) with path label \( \alpha \).
- Now, consider all suffixes of \( S \) that begin with \( \alpha \). (all paths from the root that go through \( x \))
- Let the “left character” of a suffix be the character before that suffix in \( S \).
- (For the string “banana$”,
  - \( \text{left(ana$) = n} \)
  - \( \text{left(anana$) = b} \)
  - \( \text{left(banana$) = NULL} \)
Theorem

**Theorem**: $\alpha$ is a maximal repeat iff $\alpha$ is an internal node in the tree and there are at least two suffixes that begin with $\alpha$ and have different left characters.

**Proof**:

Can’t extend to left $\iff$ There are two suffixes begin with $d\alpha$ and $e\alpha$ for two different letters $d$ and $e$.

Can’t extend to right $\iff$ It is an internal node.
Finding maximal repeats

• An internal node’s leaf descendants are suffixes, each of which has a left character.
• If the left characters of all leaf descendants are not all identical, it’s called a “left-diverse” node.
• Maximal repeats $\iff$ left-diverse internal nodes.

**Algorithm:**
• Do a DFS on the tree
• For each leaf, label it with its left character
• For each internal node:
  – If at least one child is labelled with *, then label it with *
  – Else if its children’s labels are diverse, label with *.
  – Else then all children have same label, copy it to current node.
• Easily performed in $O(m)$ time.
Application IV: Large-scale global alignment

- Large-scale global alignment

- Idea:
  - Pick some “anchors” through which the true alignment is very likely to fall.
  - Align the regions between the anchors either recursively or just using classical global alignment tools.

- Can we use suffix trees to choose good global anchors?
- First program that does so: MUMMER by Delcher et al.
Seeds come from suffix trees

• Where Mummer differs from other programs is in the anchor it uses.

• An anchor is a maximal unique match, or MUM.

• That is, it’s a string that occurs exactly one time in each genome, and which can’t be extended either way and still be a match.

• What is that in a suffix tree?
How to find mums?

• We want to find all mums instead of just the longest single one.
• Just as for maximal repeats, build a suffix tree for $S\#T\$.
• A mum is a maximal match between the two sequences…
• Can’t extend on the right:
  \[\Rightarrow\] an internal node in the tree
• It occurs once in $S$ and once in $T$:
  \[\Rightarrow\] 2 descendants, one from $S$ and one from $T$
• Can’t extend on the left:
  \[\Rightarrow\] Different left characters in both descendants

• All mums between $S$ and $T$ can be found in … linear time.
Applications of suffix trees

• That’s all we’re going to do with suffix trees.

• Most commonly used in 2 areas of bioinformatics
  1) Repeat finding
  2) Very-large-scale sequence alignment
Quick Note on Suffix Array

• Suffix tree is not a compact data structure.
  • A lot of pointers
• Gene Myers and Udi Manber (VP engineering, Google) proposed suffix array.
• A suffix array stores the positions in a string. Each position is an integer so this is a length n integer array.
• Each position corresponds to a suffix starting at this position.
• The suffix array is sorted according to the string order of the corresponding suffixes.
Suffix Array

- AGAAGAT

1 = AGAAGAT
2 = GAAGAT
3 = AAGAT
4 = AGAT
5 = GAT
6 = AT
7 = T

3 = AAGAT
1 = AGAAGAT
4 = AGAT
6 = AT
2 = GAAGAT
5 = GAT
7 = T

3, 1, 4, 6, 2, 5, 7
String Matching

• Binary search to find substring of length m.
  • $O(m \log n)$ if implemented straightforwardly
  • $O(m + \log n)$ if with an auxiliary data structure called longest common prefix (LCP) array.
**Suffix Array Construction**

- The construction of suffix array is also referred to as suffix sorting, which can be done in linear time.
  - LCP array also takes linear time to construct
- We only learn one of the linear time suffix sorting algorithms.
Preparation: Radix Sort

• **Radix Sort**: Multiple passes. Each pass stable sorts according to one digit. From the least to the most significant digit.

• original: its, iss, ipp, abc, att
• pass1: abc, ipp, its, iss, att
• pass2: abc, ipp, iss, its, att
• pass3: abc, att, ipp, iss, its

• Radix sorting requires $O(k)$ space, where $k$ is the size of the alphabet.
• Each pass takes linear time.
Skew Algorithm For Suffix Sorting

- Let $S_0, S_1, S_2, \ldots, S_{n-1}$ be all the $n$ suffixes. $S_i$ starts at $i$-th position.
- Skew algorithm uses divide and conquer.
- Two sets $SA^{0} = \{S_i : i = 0 \mod 3\}$ and $SA^{12} = \{S_i : i=1 \text{ or } 2 \mod 3\}$.
  - 1. Sort $SA^{12}$ recursively.
  - 2. Sort $SA^{0}$ in linear time.
  - 3. Merge sort $SA^{0}$ and $SA^{12}$ in linear time.
- The time complexity $T(n) = O(n) + T(2n/3)$. So it is linear.
  - Compare with merge sort.
Skew Algorithm For Suffix Sorting

1. Sort $SA^{12}$ recursively.
2. Sort $SA^0$ in linear time.
3. Merge sort $SA^0$ and $SA^{12}$ in linear time.
Sort $S^{12}$ recursively

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 \\
\hline
\text{mississippi} \\
\end{array}
\]

• We need to know the order of these suffixes.
Sort $S^{12}$ recursively

0 1 2 3 4 5 6 7 8 9 0
\textbf{mississippi}

suffixes mod 1
\textit{ississippi}

suffixes mod 2
\textit{ississippi00} \textit{ssissippi}

- Sorting these suffixes is equivalent to sorting the suffixes of the sequences of triples. Length is shorter if we regard each triple as a basic unit.
- However, comparing two basic units takes more than one comparison.
Sort $S^{12}$ recursively

- Solution: rename each triple by an integer. A smaller triple assigned with a smaller integer.
Renaming in Linear Time

• Renaming is to map each triple to an integer. Smaller triple has smaller integer.

• Thus, we can just radix sort the triples, and map each triple to its index in the sorted list.

• Example:
  • original: its, iss, ipp, abc, att
  • sorted: abc, att, ipp, iss, its
  • rename: abc➔1, att➔2, ipp➔3, iss➔4, its➔5

• Note: We can’t do quicksort or merge sort because $O(n \log n)$ time. Radix sort takes linear time.
Skew Algorithm For Suffix Sorting

• We assume $SA^{12}$ is sorted already, and learn the other two steps first.
• 1. Sort $SA^{12}$ recursively.
• 2. Sort $SA^0$ in linear time.
• 3. Merge sort $SA^0$ and $SA^{12}$ in linear time.
Sort $S^0$ in linear time

- $S_i = s[i] S_{i+1}$.
- For all $S_i$ in $SA^0$, $S_{i+1}$ has been sorted already. Use $s[i]$ to do another pass of radix sorting will give us the right order of $SA^0$. This takes linear time.
Skew Algorithm For Suffix Sorting

1. Sort $SA^{12}$ recursively.
2. Sort $SA^0$ in linear time.
3. Merge sort $SA^0$ and $SA^{12}$ in linear time.
Merge $S^0$ and $S^{12}$

- Merging only requires to compare a suffix $S_j$ with $j \mod 3 = 0$ with a suffix $S_i$ with $i \mod 3 \neq 0$.

- Case 1: If $i \mod 3 = 1$, we write $S_i$ as $(s[i], S_{i+1})$ and $S_j$ as $(s[j], S_{j+1})$.
  - Since $(i + 1) \mod 3 = 2$ and $(j + 1) \mod 3 = 1$, the relative order of $S_{j+1}$ and $S_{i+1}$ can be determined from their position in $SA^{12}$.

- Case 2: If $i \mod 3 = 2$, we compare the triples $(s[i], s[i + 1], S_{i+2})$ and $(s[j], s[j + 1], S_{j+2})$. 
C codes

• 50 lines of C++ codes were given in J.C.M. Baeten et al. (Eds.): ICALP 2003, LNCS 2719, pp. 943–955, 2003.