Suffix Trees

Martin Farach-Colton
Rutgers University & Tokutek, Inc
What’s in this talk?

✦ What’s a suffix tree?
✦ What can you do with them?
✦ How do you build them?
✦ A challenge problem
What’s a suffix tree?

- Compacted trie of all suffixes of a string.
- Trie?

```
abba
aba
bbb
bba
```
What’s trie good for?

• Looking up strings.
  • Build: $O(n \log \sigma)$ time, $O(n)$ size
    • for strings of length $n$
    • alphabet of size $\sigma$
  • Query: $O(m \log \sigma + p)$
    • for pattern of length $m$
    • if we need to report $p$ occurrences

• We’re finding all string with query as a prefix.
What’s a suffix tree?

- Compacted trie of all suffixes of a string.
- Compacted Trie?
  
  2,3,1
  
  abba
  
  aba
  
  bbb
  
  bba
Compacted trie?

- Also for looking up strings.
  - Build: $O(n \log \sigma)$ time, $O(k)$ size
    - if there are $k$ strings
  - Query: $O(m \log \sigma + p)$
And a suffix tree?

+ It’s still the compacted trie of all suffix of a string.
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi
Brute-force Algorithm

- We insert one suffix at a time.
- Each suffix insertion takes $O(n)$ time.
- Total time is $O(n^2)$
- Size of suffix tree?
  - n-leaf compacted trie...
  - So it has size $O(n)$.
What’s in this talk?

- What’s a suffix tree?
- What can you do with them?
- How do you build them?
- A challenge problem
It’s still a trie...

- **Pattern Matching:**
  - Preprocess: $T[1,n]$
  - Query: Given pattern $P[1,m]$, find all occurrences of $P$ in $T$.

- **Naive is standard pattern matching.**
  - Time: Preprocessing --, Query $O(n+m)$
Pattern Matching

- Build suffix tree of $T$
- Trace from root with $P$. (Use ST as trie.)
  - If you don’t fall of out of tree, all leaves below wherever you end up are the occurrences.
- Time:
  - Preprocessing: $O(n)$ [or $O(n \log n)$]
  - Query: $O(m \log \sigma + p)$, for alphabet of size $\sigma$ and $p$ occurrences.
What's going on?

P occurs in T if it's the prefix of a suffix of T:

banana s

nanan
k-differences

* Given a text $T[1,n]$ and a pattern $P[1,m]$, error threshold $k$, find all occurrences of $P$ in $T$ with hamming distance at most $k$.

* $T=$bananas, $P=$nan, $k=1$
k-differences

- Brute force would be $O(nm)$.
- We’ll build a suffix tree of $T$ and preprocess it for constant-time LCA.
  - $O(n+m)$ for both.
- How can we use LCA in a suffix tree to give us what we need?
Suffix Tree LCA = LCP
LCP solves k-differences

✦ At each position of the text, we’ll spend O(k) time deciding if there’s a match.
  ✦ Queries will be O(kn), instead of O(nm).
✦ How?
  ✦ $x_1 = \text{LCP}(T[i,n],P[1,m])$
  ✦ $x_2 = \text{LCP}(T[i+x_1+1,n],P[x_1+1,m])$, etc.
  ✦ After k LCP, you’ve either used up P (match), or there’s some of P left (mismatch).
k-diff

Thursday, March 17, 2011
Other uses?

- What’s the longest repeated substring in a string?
- Try finding all palindromes in a string.
- Try doing a Lempel-Ziv compression quickly.
- Other uses later, but there are books written just on using suffix trees.
What’s in this talk?

- What’s a suffix tree?
- What can you do with them?
- How do you build them?
- A challenge problem
History of suffix trees

- Weiner ’73 defined ST and gave $O(n)$ for binary alphabets.
  - Directly gives $O(n \log \sigma)$ for large alphabets.
- Many variants over the years with same time, but simplified or on-line or...
- Farach ’97 gave $O(n)$ for integer alphabets.
  - This is optimal for any alphabet.
Outline of algorithm

- Sort the suffixes
- Convert to Suffix Array
- Convert to Suffix Tree
Suffix Sorting

* Given a string, produce the sorted order of its suffixes.
* This step is bulk of the presentation.
* The suffix array and final suffix tree steps are easier.
Example: Mississippi

{ippi$ ississippi$ sissippi$ isissippi$ sissippi$ sippi$ ippi$ ppi$ pi$ i$ $

{Mississippi$ ississippi$ i$ Mississippi$ pi$ ppi$ sippi$ ssissippi$ ssissippi$ ssissippi$ $

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

Thursday, March 17, 2011
How fast can we sort?

- Sorting suffixes of a string $\Sigma^n$ can be no faster than sorting the characters $\Sigma$ of a string. $\Omega$(char sorting)

- Can we match the sorting lower bound?
Recall that Radix Sort proceeds in steps:

- Lexicographically sort the last \( i \) characters of each string.
- Stably sort by preceding character. Now strings are lexicographically sorted by last \( i+1 \) characters.
Suffix Sorting

✦ How would Radix Sort do?

✦ Time $O(n^2)$ to $O(n^2 \log n)$
  ✦ Depending on assumptions on sortability of characters.
  ✦ $O(n^2 \log n)$ in comparison model.
  ✦ $O(n^2)$ for small integers.
  ✦ In between in general word model.

✦ We can do better by combining Merge Sort with Radix Sort.
Building Blocks

- Range Reduction
- Radix Step
- Chunking
Range Reduction

Observation: If we apply a monotone function to the characters, the sorted order doesn’t change.
Example: Mississippi$
Range Reduction

- We use one range reduction:
  - Replace every character by rank in sorted order of characters.
  - After RR, length $n$ string will be in $[n]^n$
RR helps running time

- Radix Sort on $\Sigma^n$ input takes $O(n^2 \log n)$.
- RR takes at most $O(n \log n)$.
- Radix Sort on $[n]^n$ inputs takes $O(n^2)$.
- Total time for radix sort is $O(n^2)$. 
Radix Step

- We already reviewed Radix Sort
  - It’s a bunch of radix steps
- What happens if we apply radix step to a subset of suffixes?
Example: 214414413315

Odd Suffixes
Example: 214414413315

Even Suffixes
Radix Step

* Normal Radix Step:
  * Every string gets a little more sorted.

* Our use of Radix Step:
  * The order of one set of suffixes is determined from the order of another set of suffixes.
Step 1: Recursively sort odd suffixes.

How? And how is it recursive? A recursive step must sort every suffix! We’ll get to that.

Step 2: Sort even suffixes in linear time.

By Radix Step.

Step 3: Merge!
Merging is tricky...

- F ‘97 gave first linear time solution.
- This yields an optimal suffix sorting routine.
- It’s a fun algorithm...
  - but highly unintuitive.
Merging is tricky...

- F ‘97 gave first linear time solution.
- This yields an optimal suffix sorting routine.
- It’s a fun algorithm...
  - but highly unintuitive.
  - Or so people keep telling me!
Flow of algorithm

Odd Suffixes: 5 1 1 1 9 7 3

Even Suffixes: 8 2 1 0 6 4 1 2

8 5 2 1 1 1 9 1 0 6 3 7 4 1 2
Let’s solve the recursion problem first.

Given two integers \( i \) and \( j \), let \( \langle i, j \rangle \) be their bit concatenation (with 0 padding).

- If \( i, j \in [n] \), then \( \langle i, j \rangle \in [n^2] \).

Given a string \( S = (s_1, s_2, \ldots, s_n) \)

- Let \( S' = (\langle s_1, s_2 \rangle, \langle s_3, s_4 \rangle, \ldots, \langle s_{n-1}, s_n \rangle) \)
Observation: The order of the odd suffixes of \( S = (s_1, s_2, \ldots, s_n) \) is computable from the order of all suffixes of \( S' = (\langle s_1, s_2 \rangle, \langle s_3, s_4 \rangle, \ldots, \langle s_{n-1}, s_n \rangle) \).

Since bit concatenation preserves lexicographic ordering.
Example: 214414413315

17 36 12 33 27 13
36 12 33 27 13
12 33 27 13
33 27 13
27 13
13

12 33 27 13
13
17 36 12 33 27 13
27 13
33 27 13
36 12 33 27 13

2x-1

3 6 1 5 4 2

14413315
15
214414413315
3315
413315
4414413315

3 bits per character
3 bits per character
treat as base 8

Thursday, March 17, 2011
Chunking + Recursion: II

- Chunking + Range Reduction = Recursion
  - Input is in \([n]^n\).
  - Chunked Input is in \([n^2]^{n/2}\).
  - Range Reduced Chunking is in \([n/2]^{n/2}\).
  - So now problem instance is half the size and we can recurse.
Example: 2144144133315

\(\langle 21\rangle\langle 44\rangle\langle 14\rangle\langle 41\rangle\langle 33\rangle\langle 15\rangle\)

\[
\begin{array}{c|c}
<14> & 1 \\
<15> & 2 \\
<21> & 3 \\
<33> & 4 \\
<41> & 5 \\
<44> & 6 \\
\end{array}
\]

3 6 1 5 4 2

361542
61542
1542
542
42
2
1542
2
361542
42
542
61542

Thursday, March 17, 2011
Recall Basic Operations

- Range Reduction
- Radix Step
- Chunking
- How we are ready for the whole algorithm.
Suffix Sorting

✦ Step 1: Chunk + Range Reduction.
  ✦ Recurse on new string.
  ✦ Get sorted order of odd suffixes.

✦ Step 2: Radix Step. (Not 2nd Recursion!)
  ✦ Get sorted order of even suffixes.

✦ Step 3: Merge!
  ✦ We still don’t know how to do this.
Why no 2\textsuperscript{nd} Recursion?

- Odd suffixes: $O(n) + T(n/2)$
- Even suffixes by Recursion: $O(n) + T(n/2)$
- $T(n) = 2T(n/2) + O(n) + O(\text{Merging})$
  - In this case, $T(n)$ is $\Omega(n \log n)$
- Even suffixes by Radix Step: $O(n)$
- $T(n) = T(n/2) + O(n) + O(\text{Merging})$
  - In this case $T(n)$ is $O(\text{Merging})$
The Trouble with Merging

- Know how the odd suffixes compare.
- Know how the even suffixes compare.
- No idea how odd & even compare!
The difference between 3 and 2

- It’s possible to merge the lists.
  - By F ’97 “unintuitive” algorithm.

- But Kärkkäinen & Sanders showed the elegant way to merge.

- They modified the recursion to make the merge easy.

- I modified their algorithm to make it even easier.
Mod 3 Recursion

* Given a string $S = (s_1, s_2, \ldots, s_n)$

* Let $S_1 = (\langle s_1, s_2, s_3 \rangle, \langle s_4, s_5, s_6 \rangle, \ldots, \langle s_{n-2}, s_{n-1}, s_n \rangle)$

* Let $S_2 = (\langle s_2, s_3, s_4 \rangle, \langle s_5, s_6, s_7 \rangle, \ldots, \langle s_{n-1}, s_n, \$ \rangle)$

* Let $O_{12}$ be order of suffixes congruent to 1 or 2 mod 3.

  * You get this recursively from sorting the suffixes of $S_1 S_2$
Example: 21441441331315

\[ \langle 214 \rangle \langle 414 \rangle \langle 413 \rangle \langle 315 \rangle \langle 144 \rangle \langle 144 \rangle \langle 133 \rangle \langle 155 \rangle \]

1 2 3 4 5 6 7 8

\[ \langle 214 \rangle \langle 414 \rangle \langle 413 \rangle \langle 315 \rangle \langle 144 \rangle \langle 144 \rangle \langle 133 \rangle \langle 155 \rangle \]

1 4 7 10 2 5 8 11

47652213

\[ 7 \ 6 \ 5 \ 8 \ 1 \ 4 \ 3 \ 2 \]

13 213 2213 3 47652213 52213 652213 7652213

3(x-4)-1 if x>4
3x-2 otherwise

O_{12}

\[ 8 \ 5 \ 2 \ 1 \ 1 \ 1 \ 10 \ 7 \ 4 \]

47652213 7652213 652213 52213 2213 213 13 3

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

- We have $O_{12}$ from the recursion.
- One Radix Step gives us $O_{01}$
  - Radix stepping a 1 suffix gives a 0 suffix.
  - Radix stepping a 2 suffix gives a 1 suffix.
- Another Radix Step gives us $O_{02}$
  - Each suffix pair is now comparable.
  - Each suffix appears in two lists.
Example: 214414413315

13315
14413315
14414413315
15
214414413315
315
413315
414413315

413315
414413315
214414413315
315
3315
413315
4413315
4414413315
5

214414413315
315
3315
413315
414413315
4413315
4414413315
5

\( \bigoplus_{12} \begin{array}{cccccc}
8 & 5 & 2 & 1 & 1 & 10 7 4
\end{array} \)

\( \bigoplus_{01} \begin{array}{cccccc}
1 & 9 & 10 & 6 & 3 7 4 12
\end{array} \)

\( \bigoplus_{02} \begin{array}{cccccc}
8 & 5 & 2 & 1 & 19 6 3 12
\end{array} \)
Merging... at last!

+ An example is worth a thousand words...
Example: 214414413315
Example: $214414413315$
Example: 214414413315

\[O_{12}: 5 \ 2 \ 11 \ 1 \ 10 \ 7 \ 4\]

\[O_{01}: 1 \ 9 \ 10 \ 6 \ 3 \ 7 \ 4 \ 12\]

\[O_{02}: 5 \ 2 \ 11 \ 9 \ 6 \ 3 \ 12\]
Example: 2144144133315
Example: 214414413315

$O_{12}^{1074}$

$O_{01}^{1910637412}$

$O_{02}^{96312}$
Example: 214414413315

$O_{01}$: 1 10 7 4

$O_{01}$: 1 9 10 6 3 7 4 12

$O_{02}$: 9 6 3 12

8 5 2 1 1 1
Example: 2144144133315

O_{12} 10 | 7 | 4

O_{01} 9 | 10 | 6 | 3 | 7 | 4 | 12

O_{02} 9 | 6 | 3 | 12

8 | 5 | 2 | 1 | 1 | 1

etc.
Total time

- $T(n)$ to sort suffix of strings in $[n]^n$
- $T(n) = \text{recursion} + 2\times\text{radix} + \text{merging}$
- $T(n) = O(n) + T(2n/3) + O(n) + O(n)$
- $T(n) = O(n)$
Total time

- The initial Range Reduction step to get a general string in $\Sigma^n$ into the integer alphabet -- $[n]^n$ -- is the bottleneck.

- So this algorithm is optimal for any alphabet.
Why did we want to sort suffixes anyway?

- Please read up on Burrows-Wheeler, especially the uses in compression.
- Recall: Suffix Sorting to Suffix Arrays to Suffix Trees...
- We’re getting there!
Suffix Arrays

✦ A suffix array is:

✦ The sorted order of suffixes

✦ Their pairwise adjacent longest common prefixes (lcp).

✦ They are handy as space-efficient indexes.

✦ How do we compute the lcp’s from the suffix order?
Example: *Mississippi*$

*Mississippi*$
Example: **Mississippi**$\}$

```plaintext
Mississippi$\}$
```

Thursday, March 17, 2011
Brute force?

- Compute the lcp one-by-one from left to right, in brute force character-by-character fashion
  - $O(n)$ to compute each lcp.
  - $O(n^2)$ to compute them all.
  - After all that work to get a linear suffix sorting???
Better idea

- Compute the lcp one-by-one, in character-by-character fashion
  - Pick a better order to compute the lcp’s.
  - Reuse the character-by-character comparisons as much as possible.
  - Suggestion: compute suffix 1’s lcp with predecessor, then suffix 2, etc.
Example: **Mississippi**

**Mississippi**

```
1 4 1 0 0 1 0 3 1 2 0
8 5 2 1 1 1 9 1 0 6 3 7 4 1 2
```
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi$
Example: Mississippi

upper hand jumps around

lower hand marches right


8 5 2 1 1 1 9 1 0 6 3 7 4 1 2
Time

- Some lcp’s might take $O(n)$.
- Each lcp comp. ends in a mismatch.
  - So all mismatch work takes time $O(n)$.
- Each lcp comp. may find matching chars
  - Each match pushes the lower frown face to the right. $O(n)$
- $O(n)$ total: Kasai, Lee, Arimura, Arikawa & Park CPM01
Turning Suffix Arrays into Suffix Trees

- There is a linear-time algorithm for turning a suffix array into a suffix tree.
- The short version of the algorithm is:
  - Cartesian Trees
  - Construction time is linear
What’s a Cartesian Tree?

- Given a numerical array
  - Root is min number
  - This splits the array in parts
  - The cartesian trees of those parts are the children of the root
- Can you come up with a linear algorithm for this?
Example: Mississippi$
Example: Mississippi$ etc.
Suffix Tree Construction

- Suffix Sorting + LCP + Cartesian Tree
- \(O(\text{Alphabet Sorting}) + O(n) + O(n) + O(n)\)
  - \(= O(\text{Alphabet Sorting})\)
What’s in this talk?

- What’s a suffix tree?
- What can you do with them?
- How do you build them?
- A challenge problem
Challenge Problem

- Document listing
- Document frequency
What if we’re given a collection of documents $D_1, D_2, ..., D_n$.

When given a pattern $P$, you want to know which documents $P$ occurs in.

Naive thing is to build suffix tree of $D_1 \$_1 D_2 \$_2 ... D_n \$_n$

But then you get a (long) list of all occurrences, rather than a (short) list of all documents.
Document Frequency

- Same as Document Listing, but instead of reporting all documents where the pattern occurs, you report the number of documents where a pattern occurs.
Naïve for both problems is

- prep: $O(n)$, query: $O(m \log \sigma + p)$
- $n =$ length of documents, $m =$ length of pattern, $\sigma =$ size of alphabet, $p =$ number of occurrences
Target Times

✶ Document Listing:
  ✶ prep: $O(n)$, query: $O(m\log\sigma + d)$
  ✶ $d = \text{number of documents where pattern occurs}$

✶ Document Frequency
  ✶ prep: $O(n)$, query: $O(m\log\sigma)$
Hint

Consider, for each leaf, the position of the next leaf to the left in the suffix tree that comes from the same document...
The End
Building suffix trees from suffix arrays

- Order of suffix array gives order of leaves.
- Process suffix array from left to right, adding each new suffix as rightmost leaf.
- You can build the suffix tree left to right by keeping a stack of the right-most path.
- Total construction is linear for any alphabet.
Example: Mississippi$
Example: Mississippi$
Example: Mississippi
Example: Mississippi$
Example: Mississippi$
Example: Mississippi

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

push
Example: Mississippi$
Example: Mississippi
Example: Mississippi$
Example: Mississippi$
Cartesian Tree

- Each step of algo yields a push or a pop.
- If top of stack \( \leq \text{lcp} \), push at most 2 nodes onto stack.
  
  - \( O(n) \)

- If top of stack > \( \text{lcp} \), pop node.
  
  - Each node can only be popped once: \( O(n) \)

- Total: \( O(n) \)
Suffix Tree Optimality

- If you are using Suffix Tree as a trie, then each node must be sorted, and the construction is optimal.

- If you are using Suffix Tree + LCAs (=LCP), then the order of children is irrelevant.

  - The children of each node can be in any order, and it need not even be consistent between nodes.
Suffix Tree Optimality

- For small integers, construction is already $O(n)$, so this is optimal, even for Scrambled Suffix Trees.

- In comparison model, suffix trees have a lower bound from element uniqueness (depends on degree of root) so we have optimal algorithm.

- For large integers (word model of computation), lower bound is linear, upper bound is super-linear.
Open Problem

- Close the gap in the time for building a large-alphabet suffix tree, when child order is irrelevant.

- Related to Deterministic Hashing Open Problem:
  - Given $n$ large integers, can you map them to small integers (poly $n$) in linear time in the word model?