Lecture 17:
Suffix Arrays

Not in Book
Homework #3
will be posted tonight
Searching Sequence Review

• Searching for a string of length $m$ in a text of length $n$
• Indexing strings with trees
  – Keyword tree, $O(m)$ search independent of number of keywords
  – Suffix tree $O(n)$ construction, $O(m)$ search
• Suffix Arrays: a practical alternative to Suffix tree search time: $O(\log n)$
• Burrows-Wheeler transform, back to $O(m)$
• A tree for representing a “dictionary” of terms
• Merges common prefixes into a single path
• Example:
  – miss
  – mississippi
  – mist
  – mister
  – sister
  – sippy
Queries supported:
Does keyword, k, appear in my text?
  – missstep
  – sip

Searching via “Threading”

Useful for spell checking, but hashing is preferred

Not good for how many words contain “sis”
Recall Suffix Trees

• A compressed keyword tree of suffixes from a given sequence

• Leaf nodes are labeled by the starting location of the suffix that terminates there

• Note that we now add an end-of-string character ‘$’
Suffix Tree Features

• How many leaves in a sequence of length $m$? $O(m)$
• How many nodes? (assume an alphabet of $k$ characters) $O(m)$
• Given a suffix tree for a sequence. How long to determine if a pattern of length $n$ occurs in the sequence? $O(n)$
SUFFIX TREE FEATURES

• How much storage?
  – Just for the edge strings \(O(n^2)\)
  – Trick: Rather than storing an actual string at each edge, we can instead store 2 integer offsets into the original text

• In practice the storage overhead of Suffix Trees is too high, \(O(n)\) vertices with data and \(O(n)\) edges with associated data

3/30/15
There exists a depth-first traversal that corresponds to lexicographical ordering (alphabetizing) all suffixes:

1. mississippi
2. ssissippi$  
3. sissippi$  
4. issipii$  
5. ssipii$  
6. sipii$  
7. ippi$  
8. ppi$  
9. pi$  
10. i$i  
11. $
Suffix Tree Construction

• One could exploit this property to construct a Suffix Tree
  – Make a list of all suffixes: $O(m)$
  – Sort them: $O(m^2 \log m)$
  – Traverse the list from beginning to end while threading each suffix into the tree created so far, when the suffix deviates from a known path in the tree, add a new node with a path to a leaf.

• 😞 Slower than the $O(m)$ Ukkonen algorithm given last time
• Sorting however did capture important aspects of the suffix trees structure
• A sorted list of tree-path traversals, our sorted list, can be considered a “compressed” version of a suffix tree.
• Save only the index to the beginning of each suffix
  11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2

• Key: Argsort(text): returns the indices of the sorted elements of a text
Argsort

- One of the smallest Python functions yet:

```python
def argsort(text):
    return sorted(range(len(text)), cmp=lambda i,j: -1 if text[i:] < text[j:] else 1)

print argsort("mississippi")
```

```
$ python suffixarray.py
[11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]
```

- What types of queries can be made from this “compressed” form of a suffix tree
- We call this a “Suffix Array”
Suffix Array Queries

- Has similar capabilities to a Suffix Tree
- Does ‘sip’ occur in “mississippi”?
- How many times does ‘is’ occur?
- How many ‘i’’s?
- What is the longest repeated subsequence?
- Given a suffix array for a sequence. How long to determine if a pattern of length $n$ occurs in the sequence? $O(n \log m)$
Searching Suffix Arrays

• Separate functions for finding the first and last occurrence of a pattern via binary search

def findFirst(pattern, text, sfa):
    """ Finds the index of the first occurrence of pattern in the suffix array ""
    hi = len(text)
    lo = 0
    while (lo < hi):
        mid = (lo+hi)//2
        if (pattern > text[sfa[mid]:]):
            lo = mid + 1
        else:
            hi = mid
    return lo

def findLast(pattern, text, sfa):
    """ Finds the index of the last occurrence of pattern in the suffix array ""
    hi = len(text)
    lo = 0
    m = len(pattern)
    while (lo < hi):
        mid = (lo+hi)//2
        i = sfa[mid]
        if (pattern >= text[i:i+m]):
            lo = mid + 1
        else:
            hi = mid
    return lo-1
Augmenting Suffix Arrays

- It is possible to augment a suffix array to facilitate converting it into a suffix tree.

- Longest Common Prefix, (lcp)
  - Note that branches, and, hence, interior nodes if needed are introduced immediately following a shared prefix of two adjacent suffix array entries.

<table>
<thead>
<tr>
<th>Suffix Array Entry</th>
<th>lcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>0</td>
</tr>
<tr>
<td>i$</td>
<td>1</td>
</tr>
<tr>
<td>ippi$</td>
<td>1</td>
</tr>
<tr>
<td>issipi$</td>
<td>4</td>
</tr>
<tr>
<td>ississippi$</td>
<td>0</td>
</tr>
<tr>
<td>mississippi$</td>
<td>0</td>
</tr>
<tr>
<td>$pi$</td>
<td>1</td>
</tr>
<tr>
<td>s$</td>
<td>1</td>
</tr>
<tr>
<td>i$</td>
<td>1</td>
</tr>
<tr>
<td>ssipi$</td>
<td>0</td>
</tr>
<tr>
<td>ssissippi$</td>
<td>0</td>
</tr>
<tr>
<td>issippi$</td>
<td>1</td>
</tr>
<tr>
<td>sissippi$</td>
<td>0</td>
</tr>
<tr>
<td>issippi$</td>
<td>0</td>
</tr>
<tr>
<td>issippi$</td>
<td>0</td>
</tr>
<tr>
<td>issippi$</td>
<td>0</td>
</tr>
</tbody>
</table>

- If we store the lcp along with the suffix array it becomes a trivial matter to reconstruct and traverse the corresponding Suffix Array.
Other Data Structures

- There is another trick for finding patterns in a text string, it comes from a rather odd remapping of the original text called a “Burrows-Wheeler Transform” or BWT.
- BWTs have a long history. They were invented back in the 1980s as a technique for improving lossless compression. BWTs have recently been rediscovered and used for DNA sequence alignments. Most notably by the Bowtie and BWA programs for sequence alignments.
Before describing the BWT, we need to define the notion of Rotating a string. The idea is simple, a rotation of $i$ moves the prefix $i$, to the string’s end making it a suffix.

- $\text{Rotate}("\text{tarheel}$", 3) $\rightarrow$ "heel$\text{tar}"
- $\text{Rotate}("\text{tarheel}$", 7) $\rightarrow$ "$\text{tarheel}"
- $\text{Rotate}("\text{tarheel}$", 1) $\rightarrow$ "arheel$\text{t}"
BWT Algorithm

BWT (string text)
\[ \text{table}_i = \text{Rotate}(\text{text}, i) \text{ for } i = 0..\text{len(text)}-1 \]
sort table alphabetically
return (last column of the table)

\[
\begin{array}{ll}
tarheel$ & $starheel \\
arheel$t & arheel$t \\
rheel$ta & eel$tarh \\
heel$tar & el$tarhe \\
eel$tarh & heel$tar \\
e$l$tarhe & l$tarhee \\
l$tarhee & rheel$ta \\
\$tarheel & tarheel$
\end{array}
\]

BTW(“tarheels$”) = “ltherea$”
BWT in Python

• Once again, this is one of the simpler algorithms that we’ve seen

```python
def BWT(s):
    # create a table, with rows of all possible rotations of s
    rotation = [s[i:] + s[:i] for i in xrange(len(s))]
    # sort rows alphabetically
    rotation.sort()
    # return (last column of the table)
    return ''.join([r[-1] for r in rotation])
```

• Input string of length $m$, output a messed up string of length $m$
A property of a transform is that there is no information loss and they are invertible.

inverseBWT(string s)

add s as the first column of a table strings
repeat length(s)-1 times:
  sort rows of the table alphabetically
  add s as the first column of the table
return (row that ends with the 'EOF' character)
Inverse BTW in Python

- A slightly more complicated routine

```python
def inverseBWT(s):
    # initialize table from s
    table = [c for c in s]
    # repeat length(s) - 1 times
    for j in xrange(len(s)-1):
        # sort rows of the table alphabetically
        table.sort()
        # insert s as the first column
        table = [s[i]+table[i] for i in xrange(len(s))]
    # return (row that ends with the 'EOS' character)
    return table[[r[-1] for r in table].index('$')]
```
BWT advantages

• A BWT is smaller than a suffix array
  – A BWT requires $2m$ bits. As many bits as needed to represent a character in the alphabet, $\log_2(4) = 2$, times the length of the string $m$. With compression you can do even better.
  – A suffix array requires $m \log_2(m)$ bits, as many bits as needed to represent an index into the string, $\log_2(m)$, times the number of suffixes. Does not compress well

• A BWT is faster than a suffix array
  – BWT $O(n)$ search
  – Suffix array $O(n \log n)$
Next Time

- The details of search using BWT’s (get used to going backwards)
- FM indices
- Sampled FM indices