Lecture 17: Suffix Arrays and Burrows Wheeler Transforms

Not in Book
Homeworks #4 & #5 will be merged
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? $O(n)$

$O(m)$
• 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
There exists a depth-first traversal that corresponds to lexicographical ordering (alphabetizing) all suffixes.

11. $  
10. i$  
7. ippi$  
4. issippi$  
1. ississippi$  
0. mississippi$  
9. pi$  
8. ppi$  
6. sippi$  
3. sissippi$  
5. ssippi$  
2. ssissippi$
One could exploit this property to construct a Suffix Tree

- Make a list of all suffixes: $O(m)$
- Sort them: $O(m \log m)$
- 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

```python
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 than branches, and, hence, interior nodes if needed are introduced immediately following a shared prefix of two adjacent suffix array entries.

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

```plaintext
<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>lcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ippi</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>issipi</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>ississippi</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>mississippi</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>p</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>ippi</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>sissippi</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>issippi</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

11/5/13 Comp 555 Fall 2013
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.
String Rotation

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.

\[
\begin{align*}
\text{Rotate(“tarheel$”, 3) } & \rightarrow \text{ “heel$tar”} \\
\text{Rotate(“tarheel$”, 7) } & \rightarrow \text{ “$tarheel”} \\
\text{Rotate(“tarheel$”, 1) } & \rightarrow \text{ “arheel$t”}
\end{align*}
\]
BWT Algorithm

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

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

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 $m$
Inverse of BWT

• 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

    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('$')]
How to use a BWT?

- A BWT is a "last-first" mapping meaning the \(i\)th occurrence of a character in the first column corresponds to the \(i\)th occurrence in the last.

- Also, recall the first column is sorted

- BWT("mississippi$") \(\rightarrow\) "ipssm$pissii"

- Compute from BWT(s) a sorted dictionary of the number of occurrences of each letter

  \[ C[*][i+1] = \{ '\$':1, 'i':4, 'm':1, 'p':2, 's':4 \} \]

- Using the last entry it is a simple matter to find indices of the first occurrence of a character on the "left" sorted side

  \[ O = \{ '\$':0, 'i':1, 'm':5, 'p':6, 's':8 \} \]
Searching for a Pattern

- Find “iss” in “mississippi”
- Search for patterns take place in reverse order (last character to first)
- Use the O index to find the range of entries starting with the last character

```python
l = {'$': 0, 'i': 1, 'm': 5, 'p': 6, 's': 8}

C[letter][i] = $imps
0 $mississippi 00000
1 i$mississippi 01000
2ippi$mississ 01010
3 issippi$miss 01011
4 isissippi$m 01012
5 mississippi$ 01112
6 pi$mississippi 11112
7 ppi$mississi 11122
8 sippi$missis 12122
9 sissippi$mis 12123
10 ssippi$missi 12124
11 ssissippi$mi 13124
   14124
```

```python
O[letter] = 01568
```
Searching for a Pattern

- Find “sis” in “mississippi”
- Of these, how many BTW entries match the second-to-last character? If none string does not appear
- Use the C-index to find all offsets of occurrences of these second to last characters, which will be contiguous

\[
C[\text{letter}][i] = \text{imps}
\]

\[
0 \begin{array}{|l|} \hline \\
$\text{mississippi}$ & 00000 \\
\hline \\
1 & i$\text{mississippi}$ 01000 \\
\hline \\
2 & ippi$\text{mississ}$ 01010 \\
\hline \\
3 & issippi$\text{miss}$ 01011 \\
\hline \\
4 & ississippi$\text{m}$ 01012 \\
\hline \\
5 & mississippi$\text{i}$ 01112 \\
\hline \\
6 & pi$\text{mississip}$ 11112 \\
\hline \\
7 &ippi$\text{mississ}$i 11122 \\
\hline \\
8 & sippi$\text{mississ}$ 12122 \\
\hline \\
9 & ssippi$\text{missi}$ 12123 \\
\hline \\
10 & ssippi$\text{missi}$ 12124 \\
\hline \\
11 & ssississippi$\text{m}$i 13124 \\
\hline \\
14 & 0[\text{letter}] = 01568 \\
\hline \\
\end{array}
\]
Searching for a Pattern

• This is done using the FMIndex as follows:

```python
def find(pattern, FMindex):
    lo = 0
    hi = len(FMindex)
    for l in reversed(pattern):
        lo = O[l] + C[lo][l]
        hi = O[l] + C[hi][l]
    return lo, hi
```

```plaintext
find("iss", Fmindex)
lo0, hi0 = 0, 12
lo1 = 0['s'] + C[0]['s'] = 8 + 0 = 8
hi1 = 0['s'] + C[12]['s'] = 8 + 4 = 12
lo2 = 0['s'] + C[8]['s'] = 8 + 2 = 10
hi2 = 0['s'] + C[12]['s'] = 8 + 4 = 12
lo3 = 0['i'] + C[10]['i'] = 1 + 2 = 3
hi3 = 0['i'] + C[12]['i'] = 1 + 4 = 5
```

```
C[letter][i] = $imps
0 $mississippi 00000
1 i$mississippi 01000
2 ippi$mississ 01010
3 issippi$miss 01011
4 ississippi$m 01012
5 mississippi$ 01112
6 pi$mississi 11112
7 ppi$mississi 11122
8 sippi$missis 12122
9 sissippi$mis 12123
10 ssippi$missi 12124
11 ssissippi$mi 13124
```

```
0[letter] = 01568
```
Recovering the $i$th Suffix

The Search algorithm returns the indices of matches within a suffix array that is implicitly represented by the BWT.

We can recover any suffix array entry by also using the FM-index.

Recall at this point we only have access to the BWT (shown in black) and the FMindex (Shown in red and green).

\[
\begin{array}{c|c}
\text{C[letter][i]} & \text{imps} \\
0 & $\text{mississippi} 00000 \\
1 & i$\text{mississi} 01000 \\
2 & ippi$\text{mississ} 01010 \\
3 & issippi$\text{miss} 01011 \\
4 & ississippi$\text{m} 01012 \\
5 & mississippi$ 01112 \\
6 & pi$\text{mississi} 11112 \\
7 & ppi$\text{mississi} 11122 \\
8 & sippi$\text{missi} 12122 \\
9 & sissippi$\text{m} 12123 \\
10 & ssippi$\text{missi} 12124 \\
11 & ssissippi$\text{mi} 13124 \\
14 & 0[\text{letter}] = 01568
\end{array}
\]
Recovering the i\textsuperscript{th} Suffix

- The i\textsuperscript{th} entry of the “hidden” Suffix Array can be found as follows:

```python
def suffix(i, Fmindex, bwt):
    result = ‘’
    j = i
    while True:
        j = O[bwt[j]] + C[j][bwt[j]]
        result = bwt[j] + result
        if (i == j):
            break
    return result

suffix(3, Fmindex, bwt)
```

```python
c[l][i] = imps
0 $mississippi 00000
1 i$mississippi 01000
2 ippi$mississ 01010
3 issippi$miss 01011
4 ississippi$m 01012
5 mississippi$ 01112
6 pi$mississippi 11112
7 ppi$mississi 11122
8 sippi$mississ 12122
9 sissippi$mis 12123
10 ssippi$missi 12124
11 ssississippi$mi 13124
14124
```

$O[letter] = 01568$
Recovering the $i^{th}$ Suffix

- The $i^{th}$ entry of the “hidden” Suffix Array can be found as follows:

```python
def suffix(i, Fmindex, bwt):
    result = ‘’
    j = i
    while True:
        j = O[bwt[j]] + C[j][bwt[j]]
        result = bwt[j] + result
        if (i == j):
            break
    return result
```

(suffix(3, Fmindex, bwt)
(continued)

\[
\begin{align*}
\text{C[letter][i]} & = \text{imps} \\
0 & \text{mississippi 00000} \\
1 & \text{i$mississippi 01000} \\
2 & \text{ippi$mississ 01010} \\
3 & \text{issippi$miss 01011} \\
4 & \text{ississippi$m 01012} \\
5 & \text{mississippi$ 01112} \\
6 & \text{pi$mississip 11112} \\
7 & \text{ppi$mississ 11122} \\
8 & \text{ippi$mississ 12122} \\
9 & \text{issippi$miss 12123} \\
10 & \text{ssippi$missi 12124} \\
11 & \text{ssissippi$mis 13124} \\
14 & \text{issippi$miss 14124} \\
\end{align*}
\]

\[
\begin{align*}
\text{O[letter]} & = 01568 \\
\end{align*}
\]
BWT Search Details

• Searching for a pattern, \( p \), in a BWT requires \( O(|p|) \) steps (same as Suffix Tree!)
• Recovering any entry from the implicit suffix tree requires \( O(|n|) \) steps, where \( n \) is the length of the BWT encoded string
• There is actually yet another index that allows one to find prefixes, \( r \), of suffixes in \( O(|r|) \)
• The largest cost associated with the BWT is constructing and storing the FMindex. It can be built in \( O(|n|) \) steps, and stored in \( O(|\Sigma| |n|) \) memory, where \( \Sigma \) is the alphabet size
Summary

• Query Power (Big is good)
  – BWTs support the fewest query types of these data structs
  – Suffix Trees perform a variety of queries in $O(m)$
Summary

- Memory Footprint (Small is good)
  - BWTs compress very well on real data
  - Difficult to store the full suffix tree for an entire genome