Lecture 17: Suffix Arrays and Burrows Wheeler Transforms

Not in Book
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
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
Saving space

• 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

\[
\begin{align*}
\text{Symbol} & \quad lcp \\
\$ & \quad 0 \\
i & \quad 1 \\
ippi & \quad 1 \\
issipi & \quad 4 \\
issississippi & \quad 0 \\
mississippi & \quad 0 \\
\end{align*}
\]

• 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, 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)

\[
\text{table}_i = \text{Rotate}(\text{text}, i) \text{ for } i = 0..\text{len(text)}-1
\]

sort table alphabetically

return (last column of the table)

\[
\begin{align*}
\text{tarheel}$ & \quad \text{tarheel}
\hline
\text{arheel}$t & \quad \text{arheel}$t
\text{rheel}$ta & \quad \text{eel}$tarh
\text{heel}$tar & \quad \text{el}$tarhe
\text{eel}$tarh & \quad \text{heel}$tarh
\text{el}$tarhe & \quad \text{l}$tarhee
\text{l}$tarhee & \quad \text{rheel}$ta
\text{$starheel} & \quad \text{tarheel}$
\end{align*}
\]

BTW(“tarheels$”) = “ltherea$"
Once again, this is one of the simpler algorithms that we’ve seen

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.

```python
def 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)
```

```
l $  l$t  l$ta  l$tar  l$tarh  l$tarhe  l$tarhee
  t  ta  tar  tarh  tarhe  tarhee  tarheel  **tarheel$**
  h  he  hee  heel  heel$  heel$t  heel$ta  heel$tar
  e  ee  eel  eel$  eel$t  eel$ta  eel$tar  eel$tarh
  r  rh  rhe  rhee  heel  rheel  rheel$  rheel$t  rheel$ta
  e  el  el$  el$t  el$ta  el$tar  el$tarh  el$tarhe
  a  ar  arh  arhe  arhee  arheel  arheel$  arheel$t
  $  $t  $ta  $tar  $tarh  $tarhe  $tarhee  $tarheel
```
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('$')]
```
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

• \(\text{BWT(“mississippi$”)} \rightarrow “ipssm$pissii”\)

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

\[
N = \{ \text{"$":1, ‘i’:4, ‘m’:1, ‘p’:2, ‘s’:4 } \}
\]

• Using \(N\) it is a simple matter to find indices of the first occurrence of a character on the “left” sorted side

\[
I = \{ \text{"$":0, ‘i’:1, ‘m’:5, ‘p’:6, ‘s’:8 } \}
\]

• We also use \(N\) to compute the “right-hand” offsets or C-index

\[
\begin{align*}
0 & \text{mississippi} \quad 0 \\
0 & \text{i$mississippi} \quad 0 \\
1 & \text{ippi$mississi} \quad 0 \\
2 & \text{issippi$missi} \quad 1 \\
3 & \text{ississippi$mi} \quad 0 \\
0 & \text{pi$mississippi} \quad 0 \\
0 & \text{ppi$mississi} \quad 1 \\
1 & \text{ppi$mississi} \quad 1 \\
0 & \text{sippi$mississi} \quad 2 \\
1 & \text{sissippi$missi} \quad 3 \\
2 & \text{ssippi$missi} \quad 2 \\
3 & \text{ssissippi$mi} \quad 3 
\end{align*}
\]
Searching for a Pattern

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

\[ I = \{ \text{'$':0, 'i':1, 'm':5, 'p':6, 's':8} \} \]
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

$mississippi$

0

i$mississipp$

i$ppi$mississ$

issippi$miss$

ississippi$m$

mississippi$

p$mississip$

pi$mississi$

ppi$mississ$

sippi$missi$

sippi$miss$

ssippi$missi$

ssissippi$mi$

0

1

2

3
Searching for a Pattern

- Find “sis” in “mississippi”
- Combine offsets with I index entry to narrow search range
- Add the C-index offsets to the I-index of the second-to-last character to find new search range

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

• Find “sis” in “mississippi”
• Find BTW entries that match the previous next-to-next-to-last character, ‘s’
• Use the C index to find the offsets of these second to last characters
• Now we know that the string appears in the text, but not where
Searching for a Pattern

- Find “sis” in “mississippi”
- We can find the pattern’s offset on the left side by combining the C index with the I index value for the first character
- Now, if we had a Suffix array we could use it to find the offset into the original text

\[ I = \{ 'i':1, 'm':5, 'p':6, 's':8 \} \]
\[ sfa = [11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2] \]
Searching for a Pattern

• Find “sis” in “mississippi”
• Actually, there is an implicit suffix array in our BWT
• We can use the last first-last property and the C index to thread back through the array to find the start position
Searching for a Pattern

• Find “sis” in “mississippi”

• Actually, there is an implicit suffix array in our BWT

• We can use the last first-last property and the C index to thread back through the array to find the start position
Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, there is an implicit suffix array in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position
Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, there is an implicit suffix array in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position
- We’re done. The text offset is 3.
BWT Search Details

• The C-index can be easily compressed
  – Indices tend to appear in runs (a string of 0s, followed by a string of 1s, etc.)
  – Rather than store each index individually, store a 2-tuple, (index, # of times it is repeated)

• Speeding up the backtracking
  – Store a separate seeded array of BWT string positions of known text-string offsets
  – Obvious choices: C-index run boundaries and a few extra select positions
    • Starts of chromosomes
    • Uniformly every m/k positions
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