Engineering Rank and Select Queries on Wavelet Trees

Roland Larsen Pedersen

Datalogi, Aarhus Universitet

Thesis defence

June 25, 2015

Overview

1. What is a Wavelet Tree?
   - Definitions
   - Constructing the Wavelet Tree
2. Queries
   - Rank
   - Select
3. Applications
   - Compression
     - Run-length encoding
     - Burrows-Wheeler Transform
     - Huffman Shaped Wavelet tree

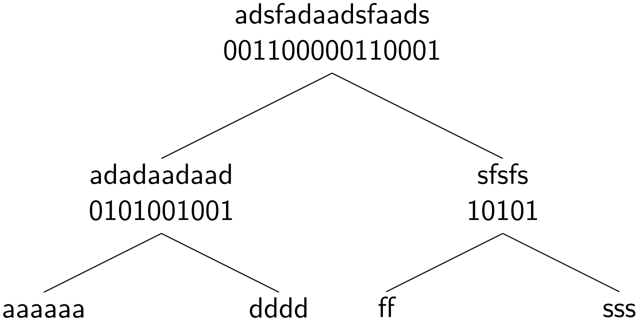
Wavelet Tree: Definitions

- In its basic form, the wavelet tree is a balanced binary tree.
- It stores a *sequence* $S[1, n] = c_1 c_2 c_3 \ldots c_n$ of *symbols* $c_i \in \Sigma$, where $\Sigma = [1 \ldots \sigma]$ is the *alphabet* of S.
- The tree has height $h = \lceil \log \sigma \rceil$, and $2\sigma - 1$ nodes, with σ of those as leaf nodes and $\sigma - 1$ as internal nodes.

Constructing the Wavelet Tree

- The wavelet tree is constructed recursively, starting at the root node and moving down the tree, with each node in the tree receiving a string constructed by its parent, except the root node that receives the full input string.
- Each node calculates the middle character of Σ and uses it to set the bits in the bitmap and split S in two substrings S_{left} and S_{right}.

Wavelet Tree Example



$S = \text{adsfadaadsfaads}, \Sigma = \text{adfs}$

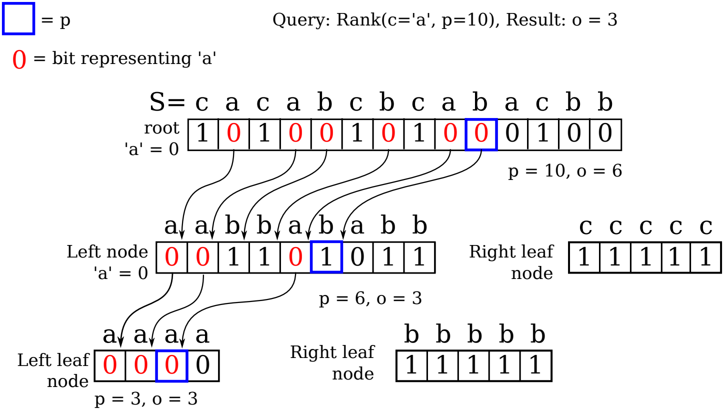
Construction time and memory usage

- Construction time: $O(n \cdot h) = O(n \log \sigma)$
  - The Wavelet Tree can theoretically be constructed in $O(n \cdot h) = O(n \log \sigma)$ time as the sum of the lengths of the strings being processed at any single layer of the tree is the length of the input string to the tree.
- Memory usage: $O(n \log \sigma + \sigma \cdot ws)$ bits
  - At each level in the tree at most n bits are stored in the bitmaps in total, making $n \cdot h = n \cdot \log \sigma$ an upper bound to the total number of bits that a wavelet tree stores in its bitmaps.
  - In addition to this, each node takes some constant amount of machine words of space, and there are $2\sigma - 1$ nodes in the tree. ws is the size of our machine words. This makes the total memory consumption $O(n \log \sigma + \sigma \cdot ws)$ bits.

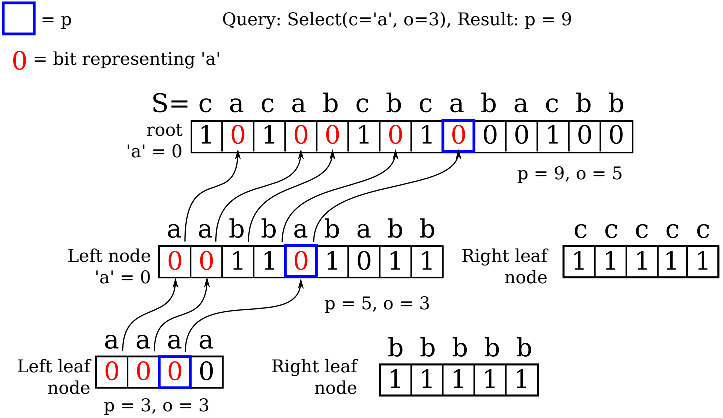
Wavelet Tree: Queries

- The wavelet tree supports three queries:
  - **Access(p)**: Return the character c at position p in sequence S.
    - Running time: $O(n \log \sigma)$.
    - We have not implemented Access because it resembles Rank.
  - **Rank(c, p)**: Return the number of occurrences of character c in S up to position p.
    - Running time: $O(n \log \sigma)$.
  - **Select(c, o)**: Return the position of the oth occurrence of character c in S.
    - Running time: $O(n \log \sigma)$

Rank on a Wavelet Tree



Select on a Wavelet Tree



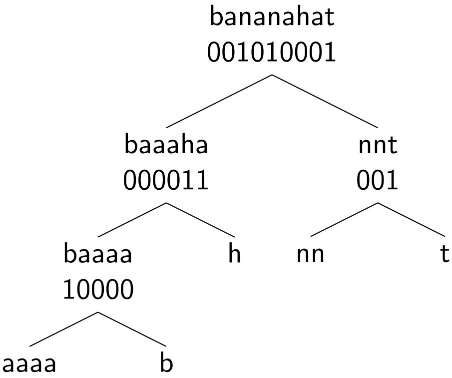
Wavelet Tree: Applications

- Compression
  - Zero-order entropy compression (H_0) using a RLE Wavelet Tree or a Huffman Shaped Wavelet Tree.
  - Higher-order entropy compression (H_k) using Burrows-Wheeler transformation and a RLE wavelet tree.
  - $H_k <= H_0 <= \log \sigma$.
- Information Retrieval
  - Positional inverted index
  - Document retrieval
  - Range Quantile Query: Return the kth smallest number within a subsequence of a given sequence of elements.
  - FM-count: Return number of occurrences of a pattern p in S.

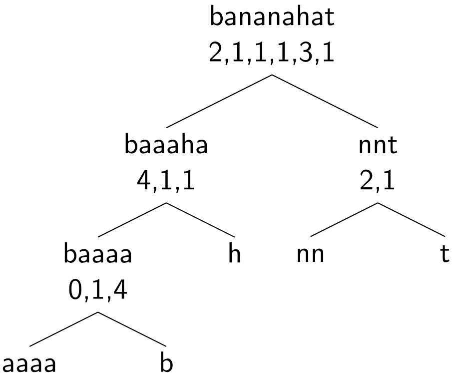
Compression: Run-length encoding

- Run-length encoding counts the number of consecutive occurrences of a symbol and substitutes the consecutive occurrences with the symbol followed by its number of occurrences.
- Example: $RLE(\text{aaaaabbbaaccccc aaaaa}) = \text{a5,b3,a2,c5,a5}$.
- Binary example: $RLE(000000000011111 00000) = 10, 5, 5$
  - We can avoid specifying the symbol by assuming that 0 is always the first symbol.
  - If the binary number begins with a 1 we just add a 0 to the beginning of the result.
- Query by reversing RLE. It takes linear time $O(n)$ to reverse. Rank and select query time becomes $O(2n \log \sigma) = O(n \log \sigma)$
- Achieves space complexity within H_0

RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



(a) Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



(b) RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$

Compression: Burrows-Wheeler transform

- BWT permutes the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row.
- As a result it groups symbols more which improves the effect of Run-length encoding
- BWT is reversible
- Combined with RLE Wavelet Tree it achieves H_k compression.

BWT example

$S =$ bananahat.

$$\begin{bmatrix} bananahat\# \\ ananahat\#b \\ nanahat\#ba \\ anahat\#ban \\ nahat\#bana \\ ahat\#banan \\ hat\#banana \\ at\#bananah \\ t\#bananaha \\ \#bananahat \end{bmatrix} \Rightarrow \begin{bmatrix} \#bananaha\mathbf{t} \\ ahat\#bana\mathbf{n} \\ anahat\#ba\mathbf{n} \\ ananahat\#\mathbf{b} \\ at\#banana\mathbf{h} \\ bananahat\# \\ hat\#banan\mathbf{a} \\ nahat\#ban\mathbf{a} \\ nanahat\#b\mathbf{a} \\ t\#bananah\mathbf{a} \end{bmatrix}$$

$BWT(S) =$ tnnbhaaaa.

Burrows-Wheeler reverse transform example

$S = \text{dca}$

$$M = \begin{bmatrix} dca\# \\ ca\#d \\ a\#dc \\ \#dca \end{bmatrix} \Rightarrow M' = \begin{bmatrix} \#dc\mathbf{a} \\ a\#d\mathbf{c} \\ ca\#\mathbf{d} \\ dca\boldsymbol{\#} \end{bmatrix}$$

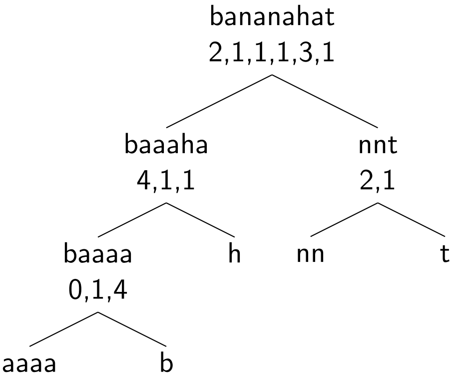
$BWT(S) = acd$

Reverse BWT:

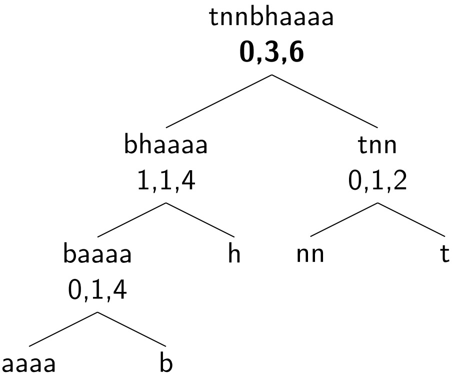
Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4
a	$\#$	$a\#$	$\#d$	$a\#d$	$\#dc$	$a\#dc$	$\#dca$
c	a	ca	$a\#$	$ca\#$	$a\#d$	$ca\#d$	$a\#dc$
d	c	dc	ca	dca	$ca\#$	$dca\#$	$ca\#d$
$\#$	d	$\#d$	dc	$\#dc$	dca	$\#dca$	**dca#**

*$\#$ = end of line character

RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



(a) RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



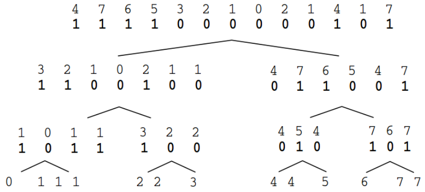
(b) BWT RLE Wavelet Tree on string *tnnbhaaaa* with alphabet $\Sigma = abhnt$

Huffman shaped wavelet tree

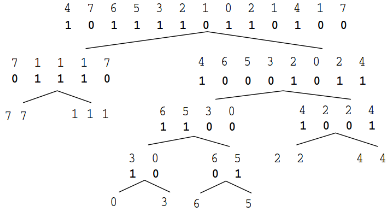
- Use Huffman codes of symbols to shape the tree
- A Huffman code is a binary value assigned to each symbol. The symbol with the highest frequency gets the lowest value.
- Shaping the tree based on Huffman codes places the most frequent symbols at the top of the tree and least frequent symbols at the bottom of the tree.
- Huffman shaping only makes sense on non-uniformly distributed data like a natural language text.

Huffman Shaped Wavelet Tree: Example

Wavelet Tree:



Huffman Shaped WT:



Huffman Shaped WT: Space complexity

- Balanced version: $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits
- Huffman-shaped: $n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(\sigma \log n)$ bits. [Efficient Compressed Wavelet Trees over Large Alphabets by Navarro et al.]
- Huffman-shaped + Compressed Bitmap (RLE): $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \log n)$ bits.

Information Retrieval

- k

The End