On Wavelet Tree Construction

German Tischler

Lehrstuhl für Informatik 2, Universität Würzburg, Germany

CPM 2011

Table of contents

Table of contents

Ordered alphabet and string

- Finite ordered alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$, $\sigma \in \mathbb{N}^+$
- Bits per symbol $\delta = \lceil \log_2 \sigma \rceil$
- Symbol $a \in \Sigma$ stored as bit vector b_a of length δ where

$$a = \sum_{i=0}^{\delta-1} 2^{\delta-i-1} b_a[i]$$

  most significant to least significant bit
- String $s \in \Sigma^n$ of length $|s| = n$ stored in δn bits.
- Address single bits of $s[i]$ by $s[i][j], 0 \leq j < \delta$
- Example $s = 53163 = (101)(011)(001)(110)(011)$ over $\Sigma = \{0 = (000), 1 = (001), \ldots, 6 = (110)\}$ where $\sigma = 7$ and $\delta = 3$

String partitioning

- $\hat{f}_i(a) = a - i2^{\delta-1}$ if $a[0] = i$ and $\hat{f}_i(a) = \epsilon$ otherwise
- $f_i(ua) = f_i(u)\hat{f}_i(a)$ and $f_i(\epsilon) = \epsilon$
- $\hat{b}(a) = a[0]$
- $b(ua) = b(u)\hat{b}(a)$ and $b(\epsilon) = \epsilon$
- String partitioning p given by

$$p(s) = b(s)f_0(s)f_1(s)$$

- Example for $s = (101)(011)(001)(110)(011)$

$$p(s) = 10010(11)(01)(11)(01)(10)$$

Balanced wavelet tree

Wavelet tree for s is (Grossi, Gupta, Vitter 2003)

- leaf assigned $b(s)$ if $\delta = 1$
- root assigned $b(s)$ with
  - left child wavelet tree for $f_0(s)$ and
  - right child wavelet tree for $f_1(s)$

  if $\delta > 1$

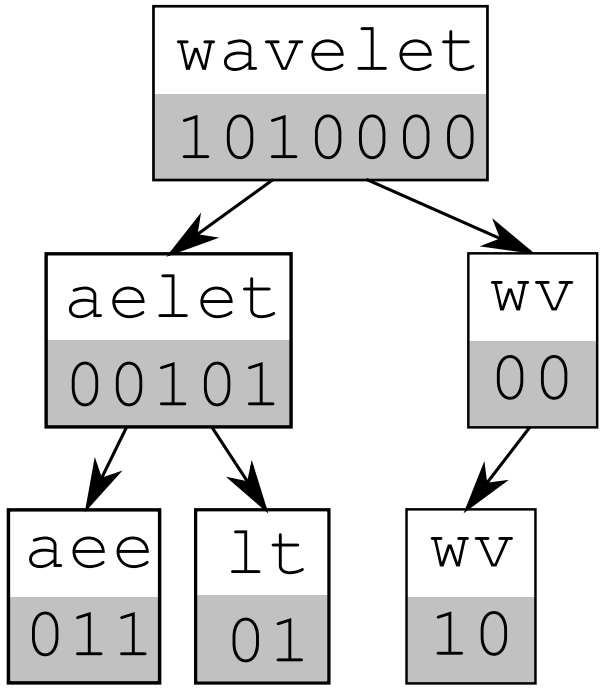
Tree can be represented by concatenation of bit vectors assigned to nodes in

- breadth first or
- depth first order.

Using rank dictionary tree can be navigated without pointers in both orders.

Example for string `wavelet`

represented by $a = 0 = 000$, $e = 1 = 001$, $l = 2 = 010$, $t = 3 = 011$, $v = 4 = 100$, $w = 5 = 101$



Navigating balanced wavelet tree

Assume we have a rank dictionary on bit vector giving us number of 0 or 1 bits up to a given position in constant time.

- Breadth first:
  - Left child is found a start of parent plus n bits. Size of left child is number of 0 bits in parent.
  - Right child is found at left child plus number of 0 bits in parent. Size of right child is number of 1 bits in parent.
- Depth first:
  - Left child is found at start of parent plus size of parent. Size of left child is number of 0 bits in parent.
  - Right child is found at start of left child plus number of 0 bits in parent times distance of left child from leaf level (number of bits per integer used to construct left child). Size of right child is number of 1 bits in parent.

In both cases top down traversal is possible in constant time per step using finite words of additional memory ($O(\log_2(n\delta))$ bits)

Huffman shaped wavelet tree

Balanced structure replaced by Huffman tree shape

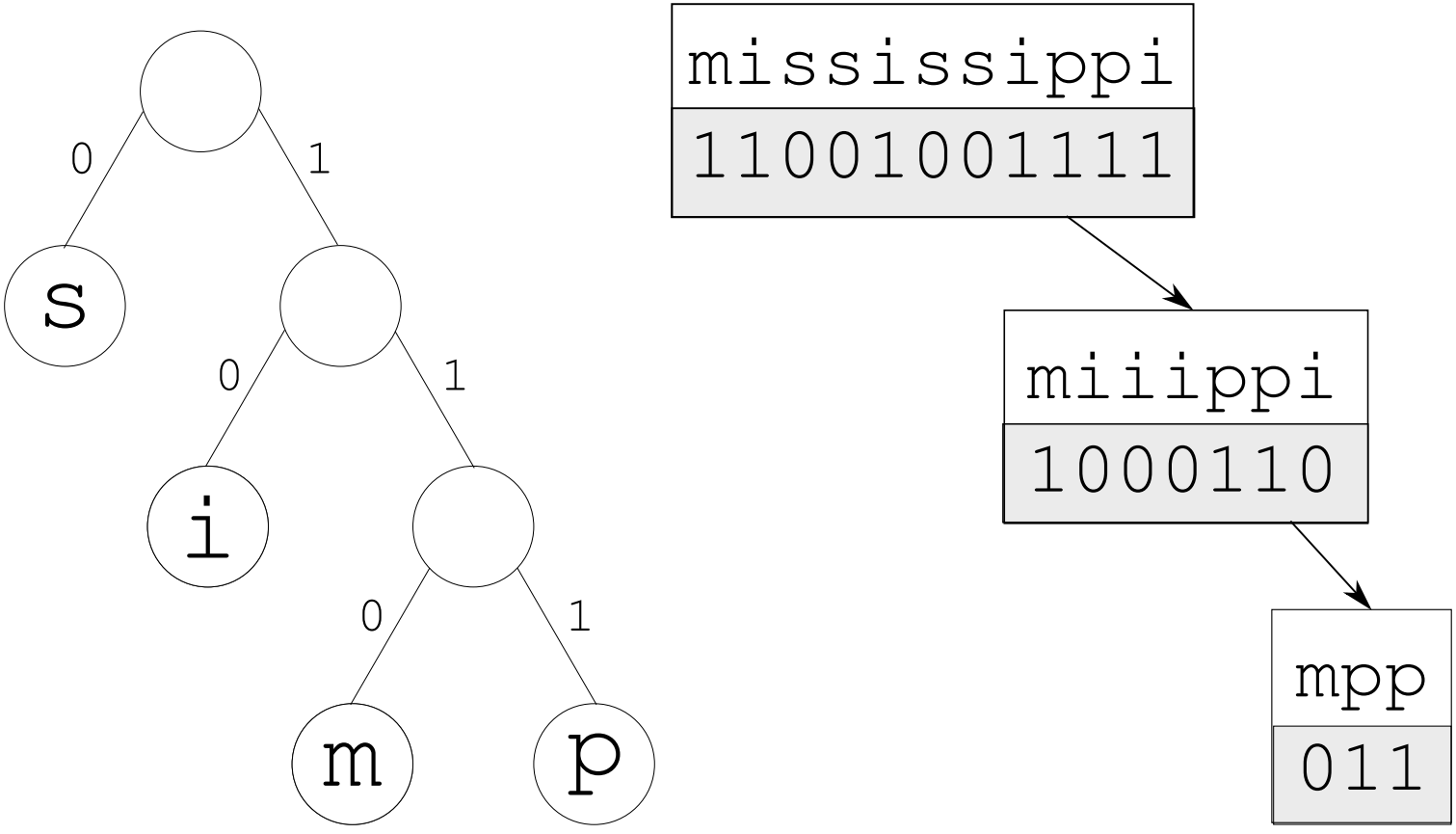


Table of contents

Stable in place adjacent block swapping

Swap two adjacent blocks $B = A[i\,..\,j-1]$ and $C = A[j\,..\,k-1]$ in an array A by computing CB as $(B^R C^R)^R$ where R is reversal

Stable merging of bit sequences

Let $B = 0^{z_b}1^{o_b}$ and $C = 0^{z_c}1^{o_c}$ concatenated in $A = BC$. B and C can be merged in linear time $O(|A|)$ by computing

$$0^{z_b}((1^{o_b})^R(0^{z_c})^R)^R 1^{o_c} = 0^{z_b}0^{z_c}1^{o_b}1^{o_c}$$

Can be extended to integers in which one certain bit (e.g. most significant bit) of each number is used as key.

Sorting with bit keys

Construct merge sort using stable bit key merging to obtain $O(n \log_2 n)$ time algorithm using constant additional space (in words, $O(\log_2 n)$ bits)

BITMERGESORT(A, m, b)

```
 1  l ← 1
 2  while l < m do
 3          for i ← 0 to ⌈m/2l⌉ − 1 do
 4                  (l_l, r_l, l_r, r_r) ← (2il, (2i + 1)l), (2i + 1)l, min(2(i + 1)l, m))
 5                  b_l ← |{j | l_l ≤ j < r_l and A[j]&b = b}|
 6                  o_r ← |{j | l_r ≤ j < r_r and A[j]&b ≠ b}|
 7                  REVERSE(A[r_l − b_l .. r_l − 1])
 8                  REVERSE(A[l_r .. l_r + o_r − 1])
 9                  REVERSE(A[r_l − b_l .. l_r + o_r − 1])
10          l ← 2l
```

Sorting with bit keys

Reversals depend on $l_l, r_l, l_l, r_r, b_l, o_r$, which remain unchanged if *keys remain in their original place*. Thus we can leave the keys in place while we sort the attached values.

BITMERGESORT(A, m, b)

```
 1  l ← 1
 2  while l < m do
 3        for i ← 0 to ⌈m/2l⌉ − 1 do
 4              (l_l, r_l, l_r, r_r) ← (2il, (2i+1)l), (2i+1)l, min(2(i+1)l, m))
 5              b_l ← |{j | l_l ≤ j < r_l and A[j]&b = b}|
 6              o_r ← |{j | l_r ≤ j < r_r and A[j]&b ≠ b}|
 7              REVERSE(A[r_l − b_l .. r_l − 1])
 8              REVERSE(A[l_r .. l_r + o_r − 1])
 9              REVERSE(A[r_l − b_l .. l_r + o_r − 1])
10        l ← 2l
```

Unsorting with bit keys

If keys remain in place we can compute the unsorted from the sorted array by

- Letting ℓ run from $2^{\lfloor \log_2 m \rfloor}$ down to 1
- Reversing the order of lines $7-9$

BITMERGESORT(A, m, b)

```
 1  l ← 1
 2  while l < m do
 3          for i ← 0 to ⌈m/2l⌉ − 1 do
 4                  (l_l, r_l, l_r, r_r) ← (2il, (2i+1)l), (2i+1)l, min(2(i+1)l, m))
 5                  b_l ← |{j | l_l ≤ j < r_l and A[j]&b = b}|
 6                  o_r ← |{j | l_r ≤ j < r_r and A[j]&b ≠ b}|
 7                  REVERSE(A[r_l − b_l .. r_l − 1])
 8                  REVERSE(A[l_r .. l_r + o_r − 1])
 9                  REVERSE(A[r_l − b_l .. l_r + o_r − 1])
10          l ← 2l
```

Transposition

Transform

$$a_0 b_0 a_1 b_1 \ldots a_{m-1} b_{m-1}$$

to

$$a_0 a_1 \ldots a_{m-1} b_0 b_1 \ldots b_{m-1}$$

is special case of stable bit key sorting. Assign key 0 to a_i and key 1 to b_i. Runtime $O(m \log_2 m)$.

Partitioning

Theorem

String s of length n can be partitioned in place ($O(\log n + \delta)$ additional bits) in time $O(n \log_2 n)$.

Proof.

First sort s while keeping keys in place, then apply transposition. Result is $b(s)f_0(s)f_1(s)$, runtime is $O(n \log n)$, additional space used is $O(\log n + \delta)$ bits. □

Using additional space

Use space $O(\sqrt{\delta n}\log_2(\delta n))$ bits to partition in $O(n)$ time. Approach is similar to linear time sorting by Kärkkäinen, Sanders and Burkhardt (see Linear Work Suffix Array Construction, JACM 53(6), 2006). Two steps:

- Use bucket sorting to fill buckets of $\sqrt{\delta n}$ bits.
  - Use 3 external buckets (one for b, one for f_0 and one for f_1) of size $O(\sqrt{\delta n})$ bits
  - Whenever an external bucket runs full copy it to a free internal bucket and store where it should be in the final array (block pointer of $O(\log_2(\delta n)$ bits)
- When all input was treated by bucket sort use permutation stored in block pointers to produce final output (follow cycles in permutation)

Approach can be split into sorting and transposition under same time and space constraints.

Table of contents

Breadth first

- Construct tree level for level
- On each level
  - First perform sorting for each node
  - Second perform transposition for complete level
  - Start and end positions of nodes can be computed by considering output already produced. This takes time $O(nk)$ for each node on level k, which is $O(n\delta\sigma)$ (last level is worst case).

  Sorting and transposition take time $O(n \log n)$ using $O(\log_2 n + \delta)$ additional space and and $O(n)$ using additional space of $O(\sqrt{n\delta} \log_2(n\delta))$ bits.
- Total runtime $O(\delta n \log_2 n + n\delta^2\sigma)$ using $O(\log_2 n + \delta)$ bits of additional space or $O(\delta n + n\delta^2\sigma)$ using $O(\sqrt{n\delta} \log_2(n\delta))$ additional bits.

Depth first

- Construct tree in depth first pre-order
- Perform partitioning operation for each node
- Use stack of depth δ storing pointers into input/output and depth in $O(\delta \log_2(\delta n))$ bits
- As above partitioning takes time $O(n \log n)$ using $O(\log_2 n + \delta)$ additional space bits and and $O(n)$ using additional space of $O(\sqrt{n\delta} \log_2(n\delta))$ bits.
- Total runtime $O(\delta n \log_2 n)$ using $O(\delta \log_2(\delta n))$ bits of additional space or $O(\delta n)$ using $O(\sqrt{n\delta} \log_2(n\delta))$ additional bits.

Huffman shaped wavelet tree and reconstruction of string

- Approach can be extended to Huffman shaped wavelet tree construction
- Method is reversible. We can compute the string from a wavelet tree in same time/space constraints as for forward direction.

Thank you