Burrows-Wheeler Transforms in Linear Time and Linear Bits

Russ Cox (following Hon, Sadakane, Sung, Farach, and others)

18.417 Final Project

BWT in Linear Time and Linear Bits

Three main parts to the result.

- Least common ancestor calculation in constant time after $O(n)$ preprocessing and $O(n)$ bits (already presented), which enables...
- Suffix tree construction for integer alphabets in $O(n)$ time and $O(n \log n)$ bits, which enables...
- Burrows Wheeler transformation in $O(n)$ time and $O(n)$ bits.

Part 1: Least Common Ancestors in Constant Time

Presented by Peter Lee.

Basic idea

- encode node id in $O(\log n)$ bits as path through tree
- find LCA by computing greatest common prefix of node ids
- some encoding tricks to handle lopsided trees

Many encoding tricks; see, for example, [Alstrup, Gavoille, Kaplan, and Rauhe, 2002]

Part 2: Suffix Trees in Linear Time and Linear Bits [Farach]

Compute odd suffix tree of paired text.

- rewrite text in new character-pair alphabet (*catdog* $\Rightarrow$ *catdog*)
- recurse on this tree, half as big, to compute tree of odd suffixes

Compute even suffix tree from odd suffix tree. Merge odd and even suffix trees to produce full suffix tree.

Compute Odd Suffix Tree of Paired Text

Construct paired text

- assume text $T = t_1 t_2 \dots t_n$ is over alphabet $\{1, \dots, k\}$
- two-pass radix sort list of all character pairs $\{t_1 t_2, t_3 t_4, \dots\}$, remove duplicates, and assign mapping to $\{1, 2, \dots, k'\}$

Recursively compute suffix tree of paired text.

Process suffix tree to yield odd suffix tree for original text.

- already mostly there
- node with children ab, ac: create new child a and hang b and c off it

Construct Even Suffix Tree from Odd Suffix Tree

Observe: in-order leaf listing and depth of lca of adjacent leaves is sufficient to reconstruct tree structure.

- reconstruct each separately, then build tree

Construct Even Leaf List from Odd Leaf List

Treat even suffixes as (char, odd-suffix) pairs.

- already have sorted list of odd suffixes
- already know sort order of individual characters
- make pairs from sorted odd suffix list; then stable sort by character

Construct Even Leaf List from Odd Leaf List

Consider *abracadabra*\$.

odd suffix array		rotate		stable sort
a\$abracadabr		r a\$abracadab		\$ abracadabra
abracadabra\$		\$ abracadabra		a bra\$abracad
bra\$abracada	⇒	a bra\$abracad	⇒	a cadabra\$abr
cadabra\$abra		a cadabra\$abr		a dabra\$abrac
dabra\$abraca		a dabra\$abrac		b racadabra\$a
racadabra\$ab		b racadabra\$a		r a\$abracadab

Construct Even LCP List from Odd LCP List

Length of longest common prefix = depth of least common ancestor.

Let suffix $s_i = t_i t_{i+1} \dots t_n$.

$$lcp(s_{2i}, s_{2j}) = \begin{cases} 1 + lcp(s_{2i+1}, s_{2j+1}) & \text{if } t_{2i} = t_{2j} \\ 0 & \text{otherwise} \end{cases}$$

Merge Odd and Even Suffix Trees

Toy example: merging suffix *tries* is trivial

- recurse on commonly-labeled subtries
- add uniquely labeled subtries

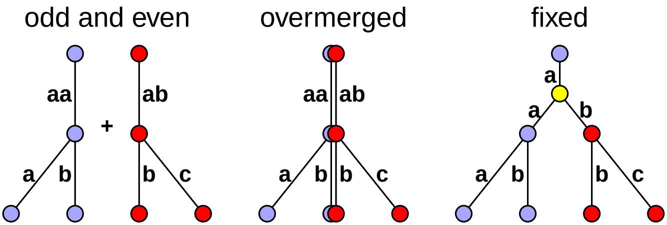
Merging suffix trees is harder

- $O(n^2)$ edges in the equivalent trie
- label comparison in suffix tree is not $O(n)$.

Solution: sloppy merge then fix

Sloppy Merging and Fixing It

Sloppy merge: treat edge labels as identical if first letters match. Overmerged nodes with correctly-merged parents form antichain across tree. Fix tree by unmerging at these points.



Identifying Overmerged Nodes

Reversed suffix links form an overlay tree.

- depth in suffix link tree equals length of suffix represented by node
- (suffix length of node = 1 + suffix length of suffix link node)
- overmerged nodes claim excess suffix length

Can compute suffix link tree in $O(n)$ time.

- suppose node has two children ℓ_{2i} and ℓ_{2j-1}
- suffix link is LCA of ℓ_{2i+1} and ℓ_{2j}

Leads to checking procedure for overmerged nodes.

- compute suffix link tree in $O(n)$ time
- compute suffix link depths for tree in $O(n)$ time
- look for incorrect nodes with correct parents and fix them

Total unmerging time $O(n)$

Suffix Trees in Linear Time and Linear Bits

Let time for text of length n be $\mathit{time}(n)$.

Compute odd suffix tree of paired text.

- $\mathit{time}(n/2)$ time, $O(n \log n)$ bits

Compute even suffix tree from odd suffix tree.

- $O(n)$ time, $O(n \log n)$ bits

Overmerge odd and even suffix trees.

- $O(n)$ time, $O(n \log n)$ bits

Unmerge overmerged tree.

- $O(n)$ time, $O(n \log n)$ bits

Total: $O(n)$ time, $O(n \log n)$ bits

Part 3: Burrows-Wheeler Transform in Linear Time and Bits [Hon et al.]

Recurse to compute “odd CSA” of character pairs (Ψ_o) a la Farach.
Build “even CSA” of shifted character pairs (Ψ_e) a la Farach.
Merge odd and even CSAs to construct BWT of full text.
The hard work is in encodings and computation tricks

- clever encoding of Ψ in $O(n)$ bits
- convert between Ψ and BW text C in linear time
- augment Ψ for $O(\log\log|\Sigma|)$ backward step time
- once recursion has reduced text length to $n/\log n$, switch to suffix tree computation (faster than above, but has super-linear space requirement)

Encoding Ψ in $O(n \log |\Sigma|)$ bits

Compressed suffix array $\Psi[i] = SA^{-1}[SA[i] + 1]$.

- link from suffix index to next smaller suffix index

Ψ increasing along runs corresponding to each alphabet character.

- shown in class

Can make completely increasing:
$\Psi'[i] = \rho(t[SA[i]], \Psi[i]) = t[SA[i]]n + \Psi[i]$.

Encode Ψ' in $O(n)$ bits, reconstruct Ψ on demand.

Encoding Ψ' in $O(n \log |\Sigma|)$ bits

Divide each value of Ψ' into $\Psi'[i] = q_i \times |\Sigma| + r_i$.

- q_i has size $\log n$ bits
- r_i has size $\log |\Sigma|$ bits

Have n values in sequence of $q_i \leq n$, monotonically increasing

- encode deltas $q_1, q_2 - q_1, q_3 - q_2, \ldots$ using unary codes $(0 = 1, 1 = 01, 2 = 001, \ldots)$
- requires n 1 bits and $q_1 + q_2 - q_1 + \ldots + q_n - q_{n-1} = q_n$ 0 bits
- total $2n$ bits maximum

Store r_i in simple array

- total $n \log |\Sigma|$ bits

Total $O(n \log |\Sigma|)$ bits.

Constant time access to q_i requires building $O(n / \log \log n)$ auxiliary structure taking $O(n)$ time.

Duality between Ψ and C

Can convert Ψ to C and vice versa in $O(n)$ time.

Ψ is easier to work with; C is easier to compute.

Converting Ψ to C

Applying Ψ to an index of SA yields the suffix array index of the next shortest suffix.

suffixes	**sorted suffixes**	i	Ψ[i]
abracadabra$	$abracadabra	1	4
bracadabra$a	a$abracadabr	2	1
racadabra$ab	abra$abracad	3	7
acadabra$abr	abracadabra$	4	8
cadabra$abra	acadabra$abr	5	9
adabra$abrac	adabra$abrac	6	10
dabra$abraca	bra$abracada	7	11
abra$abracad	bracadabra$a	8	12
bra$abracada	cadabra$abra	9	6
ra$abracadab	dabra$abraca	10	3
a$abracadabr	ra$abracadab	11	2
$abracadabra	racadabra$ab	12	5

Can iterate Ψ to learn suffix array slots of each suffix

- $\Psi^k[1]$ = suffix array slot of $s_k = t_k \dots t_n$
- if the suffix s_k is in slot i in the suffix array, then $C[i] = t_{k-1}$

Iterate Ψ, filling in one C entry at each step.

Converting C to Ψ

Can build Ψ during backward search of text T in C

- on problem set
- use $lo(\sigma w) = block(\sigma) + occ(\sigma, lo(w))$, where $\sigma = C[lo(w)]$

Note that $lo(w)$ is index of string w in sorted suffix array.

- so $\Psi[lo(C[lo(w)]w)] = lo(w)$
- so $\Psi[block(C[i]) + occ(C[i], i)] = i$

Can compute all Ψ values in order of i, filling in *occ* row as we go.

compute *block* array ($O(n)$ time)
set *occ* array to zeros
for $i = 1$ to n
    record $\Psi[block(C[i]) + occ(C[i])] = i$
    increment $occ(C[i])$

(Skipping details about storing Ψ' using $O(n)$-bits encoding during construction to avoid $O(n \log n)$ work space.)

Augmented Ψ for Improved Backward Search

Can preprocess Ψ into a handful of complicated multi-level tables

- levels and levels and levels and levels
- G. Jacobson. “Space-efficient Static Trees and Graphs.” FOCS 1989.
- J. I. Munro. “Tables.” Conf. on Foundations of Software Technology and Theoretical Computer Science. 1996.
- T. Hagerup, P. B. Miltersen, and R. Pagh. “Deterministic Dictionaries.” J. Algs 41(1), 2001.
- section 3 of Hon et al. paper

Net result is $O(\log \log |\Sigma|)$ backward steps.

Computing C_e from Ψ_o

Remember *abracadabra*\$.

odd suffix array		rotate		stable sort
`a$abracadabr`		`r a$abracadab`		`$ abracadabra`
`abracadabra$`		`$ abracadabra`		`a bra$abracad`
`bra$abracada`	$\Rightarrow$	`a bra$abracad`	$\Rightarrow$	`a cadabra$abr`
`cadabra$abra`		`a cadabra$abr`		`a dabra$abrac`
`dabra$abraca`		`a dabra$abrac`		`b racadabra$a`
`racadabra$ab`		`b racadabra$a`		`r a$abracadab`

Compute stable sort order of first column, permuting last two columns in same way. (Requires $O(n \log |\Sigma|)$ space.)

Have to use Ψ to extract columns for sort.

- just like using Ψ to extract C (last column of odd suffix array)

Merging Ψ_o and Ψ_e

Backward search for T_o on Ψ_o and Ψ_e.

- at each step have rank of suffix of T_o among odd and even suffixes of T
- set $C[\text{sum of ranks}] =$ char preceding current suffix

Backward search for T_e on Ψ_o and Ψ_e.

- same

From C, compute Ψ.
$O(n)$ backward steps of $O(\log\log|\Sigma|)$ time each.
Merging: $O(n \log\log|\Sigma|)$ time.

BWT in Linear Time and Bits

Let $i = \lceil \log \log_{|\Sigma|} n \rceil$.
Apply BWT recursion to depth i, then call suffix tree construction.
Suffix tree runs on text of length $n \log_{|\Sigma|} n / \log n = O(n / \log n)$.

- $o(n)$ time and $O(n \log_{|\Sigma|} n)$ bits

Recursion runs on texts of size
$n, n/2, n/2^2, \ldots, n/2^j, \ldots, n/2^i = n/\log_{|\Sigma|} n$.

- $O(n/2^j \log |\Sigma|^j)$ bits in each step
- $O(n/2^j + |\Sigma|^j)$ time for odd to even
- $O(n/2^j \log \log |\Sigma|^j + |\Sigma|^j)$ time for merge

Total is $O(n \log \log |\Sigma|)$ time and $O(n \log |\Sigma|)$ space.
Q.E.D.

Take-Home Messages

Can do LCA in constant time.

Can compute suffix trees of integer alphabets in $O(n)$ time with $O(n \log n)$ bits.

Can compute Burrows-Wheeler transform in $O(n)$ time and $O(n)$ bits.

- seems more theoretical than practical
- assumes constant $|\Sigma|$ (unlike suffix tree result!)
- isn't self-supporting (requires suffix tree result)

References

Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. "Identifying nearest common ancestors in a distributed environment," Tech. Rep. IT-C Series 2001-6, ISSN 1600-6100, The IT University of Copenhagen, Aug. 2001.
http://www.it-c.dk/people/stephen/Papers/ITU-TR-2001-6.ps

Martin Farach. "Optimal Suffix Tree Construction with Large Alphabets." FOCS 1997.

http://portal.acm.org/citation.cfm?id=796326

Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. "Breaking a Time-and-Space Barrier in Constructing Full-Text Indices." FOCS 2003.
http://tcslab.csce.kyushu-u.ac.jp/~sada/focs03.ps