Suffix Trees & Information Retrieval

Harry Hochheiser
hsh@cs.umd.edu

Motivating Problem

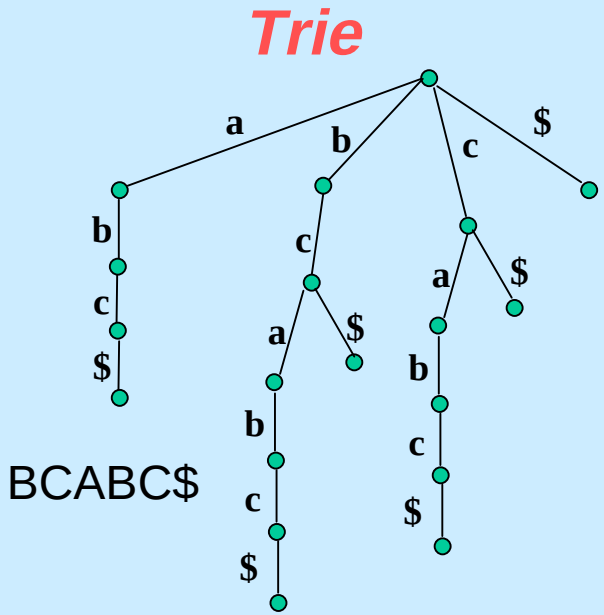
- Understanding user interaction patterns
  - Logs of User Sessions:
    - *WORD, EXPLORER, OUTLOOK, WORD*
  - Use patterns to inform interface design
- Model as string search problem
  - *ABCA…*
  - Find common repeated subsequences of some length $> k$
- Brute-force $O(n^2)$ algorithm based on diagonals of incidence matrix, but…
- We want *approximate* matches:
  - *xxxxxABCxxxxCBAxxxxxBAC*
- Still looking for an answer – I don't think it's an easy problem.

Outline

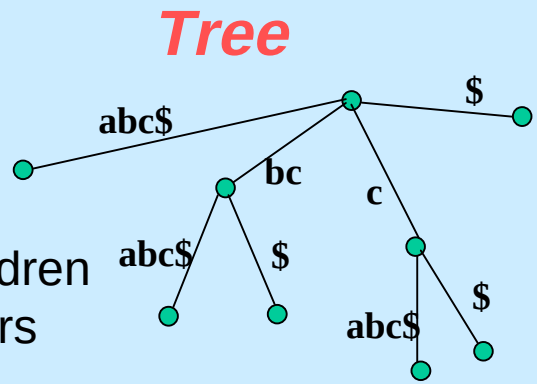
- Basic Suffix Trees & Extensions
  - Definitions
  - Brute-Force
  - McCreight – suffix links
  - Searching
  - Application to Common Subsequences
- Parameterized Suffix Trees
  - Motivations
  - Definitions
  - Searching & Example
  - Parameterized Duplications

Tries & Trees

- *Trie*: Digital Search Tree over strings in alphabet *C*
- Each edge is a symbol, and siblings represent distinct symbols
- Final character of string cannot occur elsewhere in string
  - Add marker symbol (“$”) to alphabet, if needed

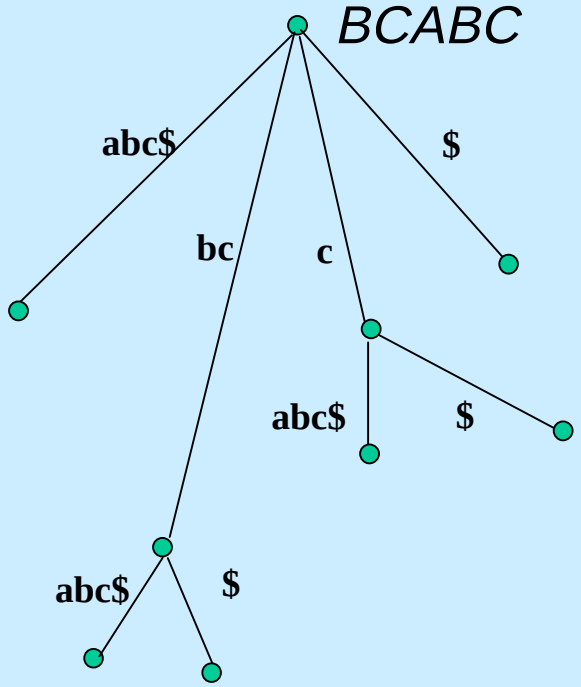


- Inefficient
  - Eliminate Unary Nodes
- ***Suffix Tree***
  - Arcs are non-empty substrings
  - Each non-terminal, non-root has two children
  - Sibling arcs begin with different characters



Formalities & Definitions

- *Y:* Suffix tree for string *y, |y|=n*
  - *n* leaves
  - Storage *O(n)*
  - Left-Child, Right Sibling Structure
- Edges: substrings *y(k,l)*
- Internal Nodes: longest common prefix of string's suffixes.
- *Locus:* node representing a string
- *Extension:* string with *u* as a prefix
- *Extended Locus:* Locus of shortest extension of *u* that is found in tree
- *Contracted Locus:* Locus of longest prefix of *u* in tree
- $Head_i$ *:* longest prefix of *y(i,n)* which is a prefix of *y(j,n)* for some *j<I*
- $Tail_i$ *: y(i,n)=head$_i$tail$_i$*

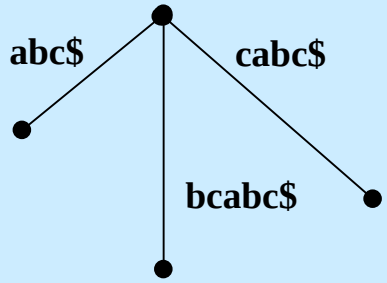


$suf_4 = BC$
$head_4 = BC$
$tail_4 = \$$

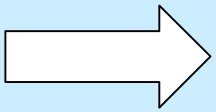
Construction: Brute Force

- Stage *i:* insert *y(i,n)*
  - Find extended locus of $head_i$ in Tree Y_{i-1}
  - If string is not $head_i$, break edge and insert internal node
  - Add a new node for $tail_i$, from locus of $head_i$

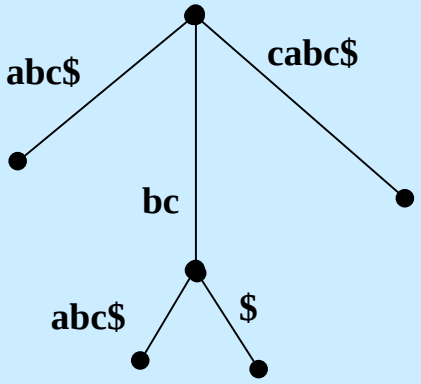
S=bcabc



Y_4 – insert *bc$*



Head = bc
Tail = $



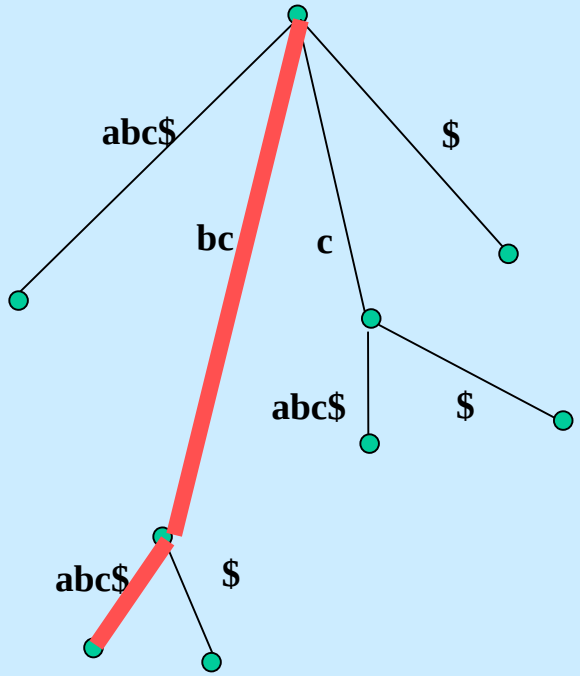
hsh@cs.umd.edu

Analysis

- Key: finding the extended locus
  - Otherwise, *O(1)* time
- Brute-force search: start from root and trace edges
  - Worst-case: *AAAAAAA$.*
  - $|head_i|=n-i$ for $1< i < n$, $n-i$ comparisons for each search
  - $T(n^2)$
- Apostolico and Szpanowski (1992): average case: $O(n \log n)$
  - Average maximum length of the longest common prefix of two suffixes is $O(\log n)$
  - Probabilistic analysis
- McCreight – *suffix links* improve performance

Searching

- Not mentioned in McCreight..
- Follow arcs with matching prefixes?



BCAB

Analysis/Hand-Waving:

Height of Tree ~ $O(\log n)$

Size of alphabet: $|\Sigma|$

Search Time ~ $O(|\Sigma| \log n)$?

Suffix Links

- Finding *extended locus* – node to be split - is the hard part
- *$Head_i$:* longest prefix of *y(i,n)* which is a prefix of *y(j,n)* for some *j<I*
- Note: if *$head_{i-1}$=az* then *z* is a prefix of *$head_i$*
- *Use locus of $head_{i-1}$ to find locus of $head_i$ in stage i*
  - Use suffix links to go from *$head_{i-1}$* to *$head_i$*
  - In *i*th tree, only locus of *$head_i$* does not have a valid suffix link
  - In step *i,* contracted locus of *$head_i$* in tree *i-1* is visited
  - Find *$head_{i-1}$=uvw,* s.t. *uv* is the contracted locus of *$head_{i-1}$* in previous tree.
  - Rescan: Follow suffix link from this node and then go down path to extended locus of *vw*
  - Scan:continue downward to find extended locus of *$head_i$*

Suffix Links – Example

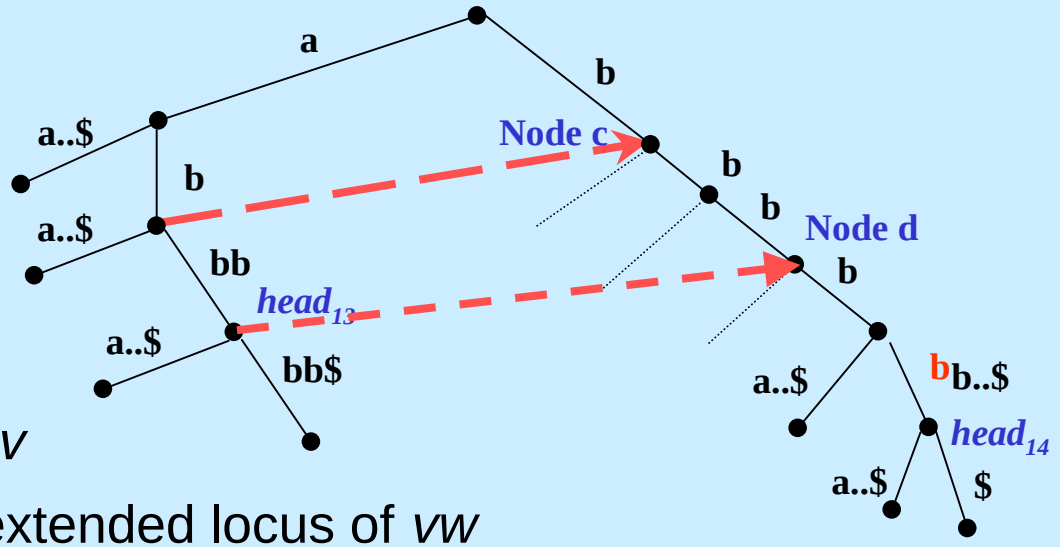
BBBBBABABBBAABBBBB$

Insert suf_{14}=*BBBBB*

$head_{13}$=*ABBB*

u=A,v=B,w=BB

$head_{14}$=*BBBBB=vwz,z=BB*



Step A*: Node c* is suffix link of *uv*

Step B - rescan: *Node d* is extended locus of *vw*

Step C – Scan: Extend from *d* to add node for *z=BB*

From $head_{14}$, we can move on to $head_{15}$….

How do we get to contracted locus for *uv* quickly?

from previous iteration's *Node d?*

Suffix Links: analysis

- Rescan:
  - res_i – shortest suffix of string to be rescanned in step $length(res_{i+1}) <= length(res_i) - int_i$
  - $int_i = |res_i| - |res_{i-1}|$
  - $length(res_n)=0$ & $length(res_0)=n$
  - $\Sigma int_i = n$ – total nodes visited in rescanning
- Scan:
  - Step i, to find $head_i$, must find z
  - scan $length(head_i) - length(head_{i-1}) + 1$.
  - Totals to n
- All other steps are linear.. total time $O(n)$

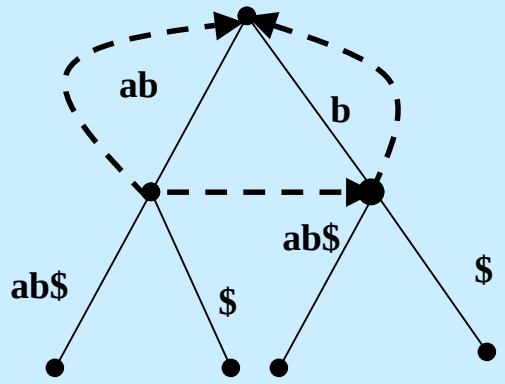
Searching with Suffix Links

- Chang & Lawler: to search for pattern *P* in text *T*, build suffix tree for *P* and compare *T* to it
  - follow path by symbols of *T*
  - use suffix links when a symbol can't be matched
    - rescan to find what we've already seen, then continue
    - report a match when we hit the last symbol that matches

P=ABAB *T=ABACABAB*

Suffix Tree for *P*

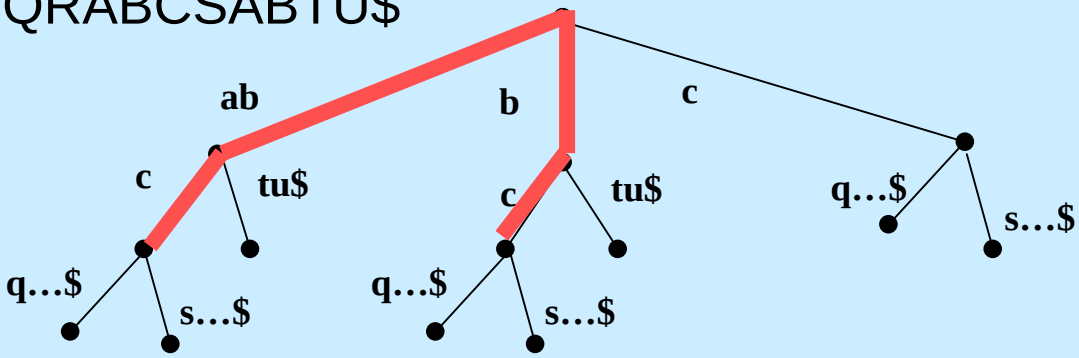
Match($T_1 \ldots T_n$)=aba$
Match($T_2 \ldots T_n$)=ba$
Match($T_3 \ldots T_n$)=a$
Match($T_4 \ldots T_n$)=$
Match($T_5 \ldots T_n$)=abab$



Suffix Trees & Repeated Subsequences

- Easy: track internal nodes – longest substring is given by maximum length internal-node substring
- Other internal nodes give repeated substrings

ABCQRABCSABTU$



Possible Extensions

Common substrings of two strings

Non-overlapping repetitions – requires augmented tree

Miscellany

- McCreight
  - Hash-coded tree for better search performance with large alphabets
  - Dynamic insertion into suffix trees in linear time
- Ukkonen (1992)
  - On-line, linear time construction
  - add successively longer prefixes (instead of shorter suffixes)
- Manber & Myers (1993)
  - *Suffix Arrays:* 3x-5x space savings in practice
- Apostolico & Szpanowski (1992)
  - Probabilistic analysis of brute-force construction

Parameterized Suffix Trees

- Motivation: find duplication in software systems

```
...
*pmin++ = *pmax++;
copy_number(&pmin,&pmax,
    pfi->min_bounds.lbearing,
    pfi->max_bounds.rbearing);
*pmin++ = *pmax++;
...
```

```
...
*pmin++ = *pmax++;
copy_number(&pmin,&pmax,
    pfh->min_bounds.left,
    pfh->max_bounds.left);
*pmin++ = *pmax++;
...
```

Brenda Baker, Bell Labs

`http://cm.bell-labs.com/cm/cs/who/bsb/index.html`

Parameterized Duplications in Strings: Algorithms and an Application to Sotware Maintenance

Siam J. Computing, October 1997.

Definitions

- *S={a,b,c}* - symbol alphabet, *? ={x,y,v}* - parameter alphabet
- *p-strings: (S+ ?)**
- **P1:** p_1 *&* p_2 are *p-matches* iff p_1 can be transformed into p_2 by a one-to-one mapping

  S=axayb

  T=avaxb
- *prev(S)* -
  - leftmost occurrence of each parameter becomes 0
  - subsequent occurrences replaced by difference in position compared to previous occurrence: *parameter pointer*

  prev(abuvabuvu) = ab00ab442
  - *S & S'* are *p-matches* iff *prev(S)=prev(S')*

More Definitions….

- *psuffix(s,i) = prev($S_i S_{i+1} \ldots S_n$)*

  S=abxvabuvu prev(S)=ab00ab442

  psuffix(S,5)=ab002

- **P2:** if *P* is a *p-string* pattern, and *S* is a *p-string* text, *P* p-matches at position *i* iff *prev(P)* is a prefix of *psuffix(S,i)*

  P=abu

  prev(P)=ab0

  psuffix(S,1)=ab00ab442

  psuffix(S,2)=b00ab442

  psuffix(S,3)=00ab442

  psuffix(S,4)=0ab042

  psuffix(S,5)=ab002

Brute-force Construction

- Like McCreight, but add succesive *p-suffix* entries
- *S={a,b,c,$},? ={v,x,y}*

  S=xbyybx$

psuffix(S,1)=0b01b5$

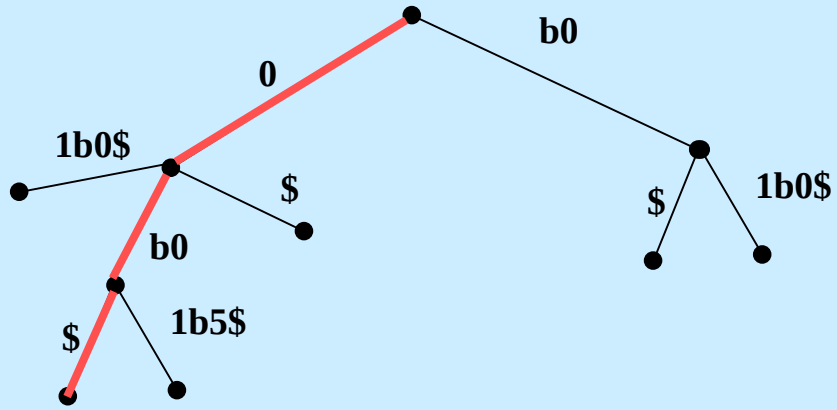
(S,2)=b01b0$

(S,3)=01b0$

(S,4)= 1b0$

(S,5)=b0$

(S,6)=0$



Searching:

follow symbols from *prev(p)*

p=vbx$

prev(p)=0b0$

Time: $O(|P|\log(|S|+|?|)$

P-Trees & Suffix Links

- Baker: Why do suffix links work?
  - Common Prefix Property: if *aS=bT* then *S=T*
  - Distinct Right Context: if *aS=bT* and *aSc ≠ bTd* then *Sc≠Td*
- Distinct Right Context does not hold for *p-strings*

  S=xabxyabz

  prev(xabx)=0ab3 *prev(yabz)=0ab0*

  0ab=0ab *(aS=bT)*

  prev(xabx) ≠ prev(yabz) *(aSc ≠ bTd)*

  prev(abx)=ab0=prev(abz) *(Sc=Td)*

- *xabx* - repetition of parameter

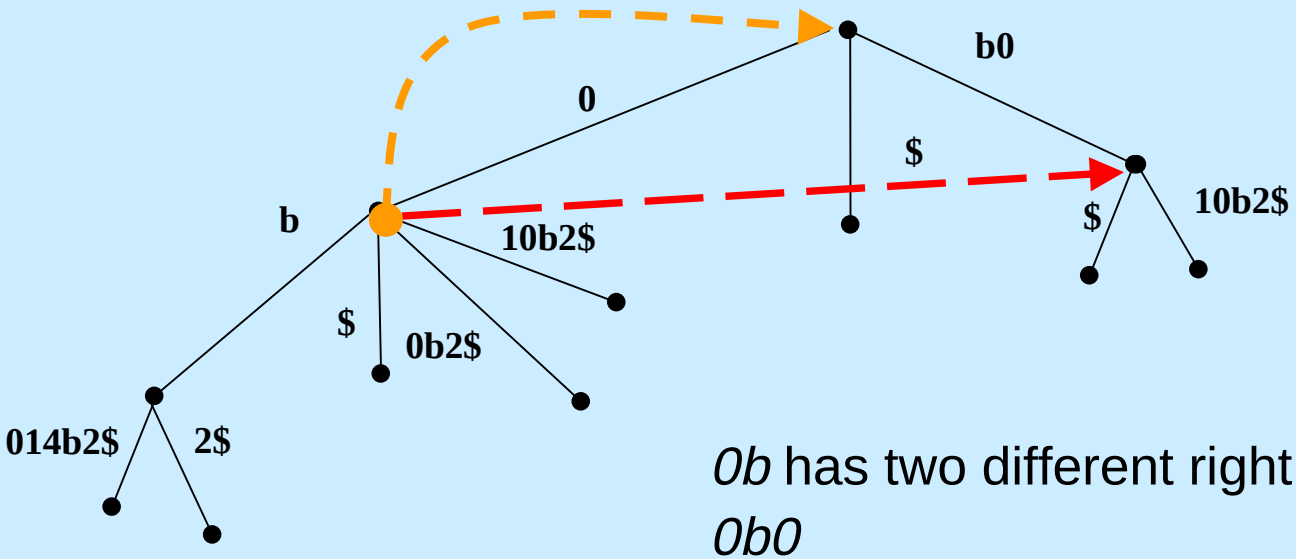
  yabz - new parameter

  abx, abz - context lost

P-Strings & Suffix Links….

- Can’t always point from *N* to node representing appropriate suffix
  - might end up pointing to node with shortest extension of suffix instead.
- Use “best available” suffix link

S=xbyyxbx



0b has two different right contexts..

0b0

0b2

Fixes

- Must get suffix links pointing to the right place
- Fix them up as we go along – lazy or eager
- *eager:* keep track of *min* - node of shortest length with a bad pointer pointing to *N*

  Six phases:
  - Set temporary suffix link for previous head
  - Scan $head_i$
  - Add new internal node if needed
    - call *update* to fix suffix links and *min* pointers
  - Fix *min* pointers of $head_{i-1}$
  - Create new node with appropriate suffix
  - update *oldhd* and *oldchild* pointers
- *O(n(|? |+*log(*| ? |+|S))))*, improve to *O(n* log*n):* concatenable queues & dynamic trees.

Searching for a pattern

- Chang & Lawler: to search for pattern *P* in text *T*, build suffix tree for *P* and compare *T* to it
  - follow path by symbols of *T*
  - use suffix links when a symbol can't be matched
    - rescan to find what we've already seen, then continue
    - report a match when we hit the last symbol of *P*
- For suffix trees - slight difference
  - when we hit a mismatch, we may need to go back up to a parent node to find the right suffix link to follow
- with appropriate optimizations,

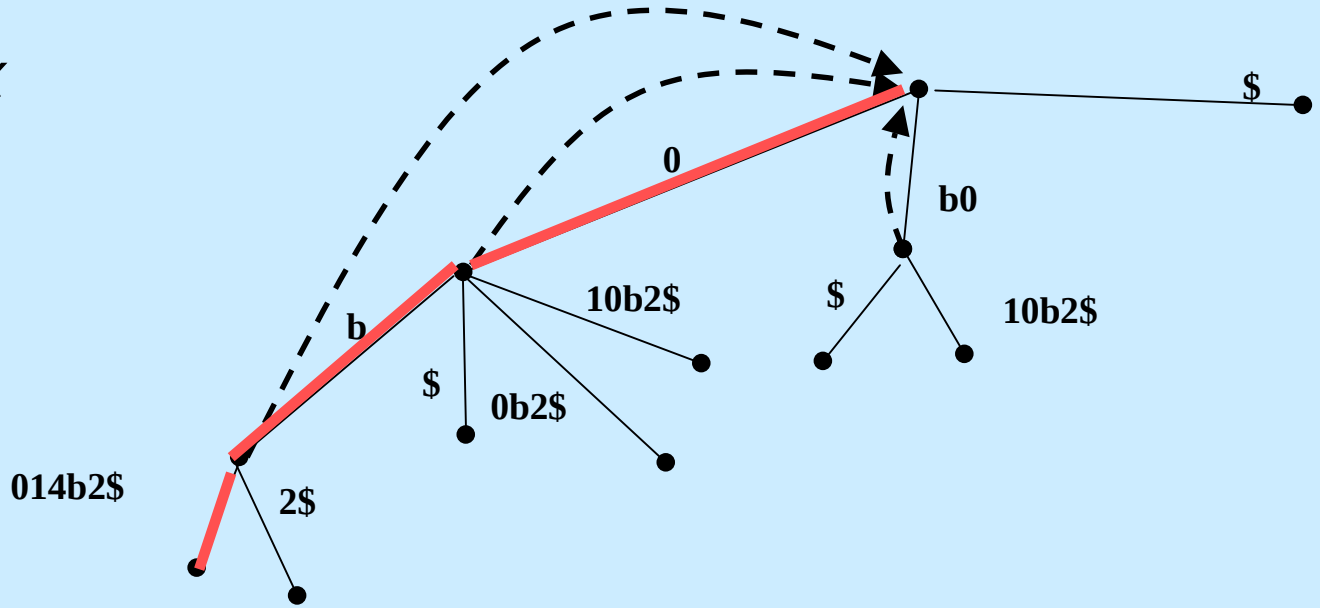
  $O(|P|)$ space and

  $O(|T| \log(|\ ?\ |+|S|))$ time

Chang & Lawler Search in *p-suffix* Trees

Search on *prev(T)* instead of *T*

S=xbyyxbx



T=vazbybxxyby

0a0b0b014b2

a0b0b014b2

0b0b014b2

b0b014b2

0b014b2

hsh@cs.umd.edu

Finding parameterized duplications

- Maximal *p-match* - cannot be extended in either direction
- Node *N* - *p-match* that can't be extended to the right
  - look at preceeding symbols to see if maximal
- *p-strings* - parameters in common prefix have different meanings

  0 - next occurrence of a parameter proceeding this suffix,or

  earlier occurrence of a parameter

  first occurrence of a parameter

  S=xabcx *T=yabcz*

  prev(abcx)=prev(abcz)=abc0

  prev(xabcx)=0abc3 *prev(yabcz)=0abc0*

  - Check previous symbols - if they are parameters, check to see whether next occurrences of those parameters match

Parameterized Duplications...

- Use $A=(prev(S^r))^r$ to find left matches

  $S=xbxzbzzyby$

  $S^r=ybyzzbzxbx$

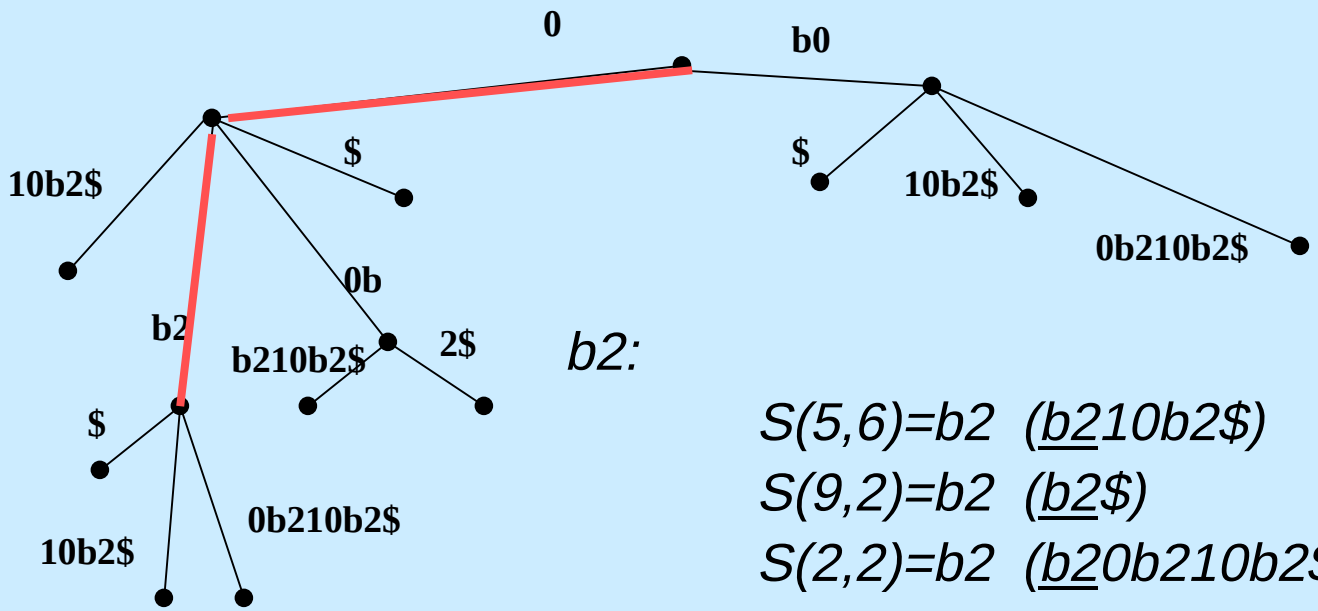
  $prev(S^r)=0b201b20b2$ $A=(prev(S^r))^r=2b02b102b0$
- Only matches with appropriate values in A can be left-extended:

  if $S_i...S_{i+k}$ matches $S_j....S_{j+k}$,

  $A_{i-1}=A_{j-1}$ means that we have a match

Parameterized Duplications, example

S=xbxzbzzyby A=(prev(S^r))^r=2b02b102b0



b2:

*S(5,6)=b2 (b2**1**0b2$)*

*S(9,2)=b2 (b2**2**$)*

*S(2,2)=b2 (b2**0**b210b2$)*

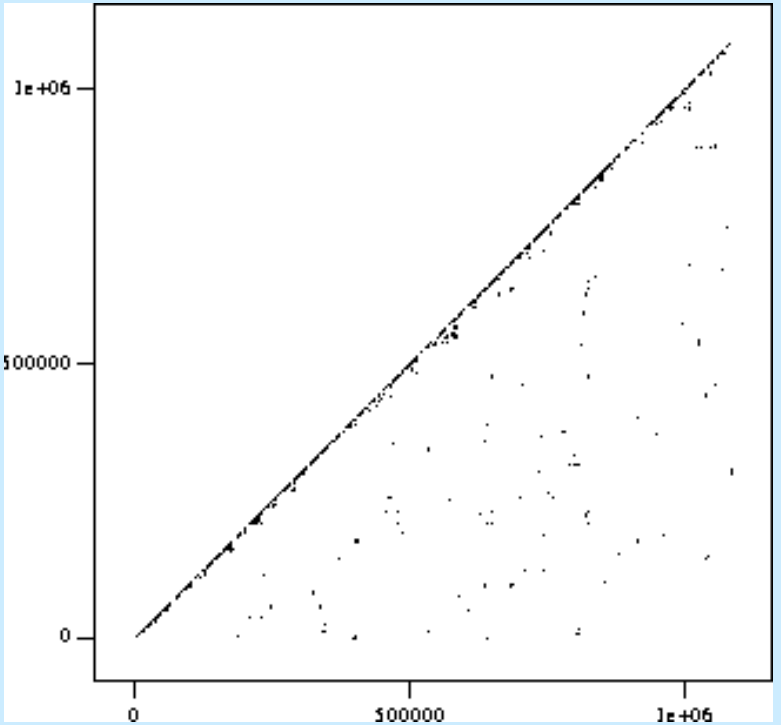
$A_4=A_8=A_1=2$, so all matches can be extended

O(n+m(t,S)) - m(t,S) is number of matches of length at least *t*

hsh@cs.umd.edu

Parameterized Duplication in Software

- Identifiers & constants become parameters
- Hash each line of code into a sybmol of S + 0 or more parameters
- Use hashing-based suffix tree
- Linear time: *O(|T|+m(t,T))*, but *m(t,T) < |T|* - O(|T|).
- With post-processing, 10^6 in 7 minutes
  - 20% involved in parameterized duplication of ≥ 30 lines



References

- Apostolico, A., Szpanowski, W. (1992).**Self-alignments in words and their applications** *Journal of Algorithms, 13,* pp. 446-67.
- Baker, B. (1993) **A theory of parameterized pattern matching: Algorithms and applications (Extended Abstract)** *Proceedings 25th ACM Symposium on Theory of Computing,* 1993, pp. 71-80
- Baker, B. (1997) **Parameterized Duplications in Strings: algorithms and an Application to Software Maintenance** *SIAM J. Computing* 25(6), October 1997. pp. 1343-1352
- Chang, W.I. & Lawler, E.L.(1990) **Approximate string matching in sublinear expected time** *Proc. Of 31st Symposium on Foundations of Computer Science,* pp. 116-124.
- Manber U. & Myers, E.W. (1993) **Suffix arrays: A new method for on-line string Searches** *SIAM Journal on Computing* (October 1993), pp. 935-948
- McCreight, E.M. (1976). **A space-economical suffix tree construction algorithm** *Journal of the ACM, 23, 2,* pp. 262-272, April 1976.
- Stephen, G. (1994) **String Searching Algorithms** London: World Scientific Press.

References, continued

- Ukkonen, E. (1992) **Constructing suffix-trees online in linear time** in Leeuwen, J. van (ed.) *Algorithms, Software, Architecture: Information Processing 92*, 1, p. 484-92. Amsterdam: Elsevier.