1 Suffix Trees

Weiner 73 “Linear Pattern-matching algorithms” IEEE conference on automata and switching theory
McCreight 76 “A space-economical suffix tree construction algorithm” JACM 23(2) 1976
Chen and Seifras 85 “Efficient and Elegegant Suffix tree construction” in Apostolico/Galil Combninatorial Algorithms on Words

Another “search” structure, dedicated to strings.

Basic problem: match a “pattern” (of length \( m \)) to “text” (of length \( n \))

- goal: decide if a given string (“pattern”) is a substring of the text
- possibly created by concatenating short ones, eg newspaper
- application in IR, also computational bio (DNA seqs)
- if pattern available first, can build DFA, run in time linear in text
- if text available first, can build suffix tree, run in time linear in pattern.
- applications in computational bio.

First idea: binary tree on strings. Inefficient because run over pattern many times.

- fractional cascading?
- realize only need one character at each node!

Tries:

- used to store dictionary of strings
- trees with children indexed by “alphabet”
- time to search equal length of query string
- insertion ditto.
- optimal, since even hashing requires this time to hash.
- but better, because no “hash function” computed.
- space an issue:
  - using array increases storage cost by \( |\Sigma| \)
  - using binary tree on alphabet increases search time by \( \log |\Sigma| \)
– ok for “const alphabet”
– if really fussy, could use hash-table at each node.

• size in worst case: sum of word lengths (so pretty much solves “dictionary” problem.

But what about substrings?

• Relevance to DNA searches
• idea: trie of all $n^2$ substrings
• equivalent to trie of all $n$ suffixes.
• put “marker” at end, so no suffix contained in other (otherwise, some suffix can be an internal node, “hidden” by piece of other suffix)
• means one leaf per suffix
• Naive construction: insert each suffix
• basic alg:
  – text $a_1 \cdots a_n$
  – define $s_i = a_i \cdots a_n$
  – for $i = 1$ to $n$
  – insert $s_i$
• time, space $O(n^2)$

Better construction:

• note trie size may be much smaller: $aaaaaaa$
• algorithm with time $O(|T|)$
• idea: avoid repeated work by “memoizing”
• also shades of finger search tree idea—use locality of reference
• suppose just inserted $aw$
• next insert is $w$
• big prefix of $w$ might already be in trie
• avoid traversing: skip to end of prefix.

Suffix links:

• any node in trie corresponds to string
• arrange for node corresp to $ax$ to point at node corresp to $x$
• suppose just inserted \(aw\).
• walk up tree till find suffix link
• follow link (puts you on path corresp to \(w\))
• walk down tree (adding nodes) to insert rest of \(w\)

Memoizing: (save your work)
• can add suffix link to every node we walked up
• (since walked up end of \(aw\), and are putting in \(w\) now).
• charging scheme: charge traversal up a node to creation of suffix link
• traversal up also covers (same length) traversal down
• once node has suffix link, never passed up again
• thus, total time spent going up/down equals number of suffix links
• one suffix link per node, so time \(O(|T|)\)

half hour up to here.
Amortization key principles:
• Lazy: don’t work till you must
• If you must work, use your work to “simplify” data structure too
• force user to spend lots of time to make you work
• use charges to keep track of work—earn money from user activity, spend it to pay for excess work at certain times.

Linear-size structure:
• problem: maybe \(|T|\) is large \(n^2\)
• compress paths in suffix trie
• path on letters \(a_i \cdots a_j\) corresp to substring of text
• replace by edge labelled by \((i, j)\) (implicit nodes)
• Example: tree on \(abab\)
• gives tree where every node has indegree at least 2
• in such a tree, size is order number of leaves = \(O(n)\)
• terminating \(\$\) char now very useful, since means each suffix is a node
• Wait: didn’t save space; still need to store characters on edge!
• **see if someone with prompting can figure out**: characters on edge are substring of pattern, so just store start and end indices. Look in text to see characters.

Search still works:

• preserves invariant: *at most* one edge starting with given character leaves a node
• so can store edges in array indexed by first character of edge.
• walk down same as trie
• called “slowfind” for later

Construction:

• obvious: build suffix trie, compress
• drawback: may take $n^2$ time and intermediate space
• better: use original construction idea, work in compressed domain.
• as before, insert suffixes in order $s_1, \ldots, s_n$
• compressed tree of what inserted so far
• to insert $s_i$, walk down tree
• at some point, path diverges from what’s in tree
• may force us to “break” an edge (show)
• tack on *one* new edge for rest of string (cheap!)

MacReight 1976

• use suffix link idea of up-link-down
• problem: can’t suffix link every character, only explicit nodes
• want to work proportional to *real* nodes traversed
• need to skip characters inside edges (since can’t pay for them)
• introduced “fastfind”
  – idea: fast alg for descending tree if *know* string present in tree
  – just check first char on edge, then skip number of chars equal to edge “length”
  – may land you in middle of edge (specified offset)
  – cost of search: number of *explicit* nodes in path
Amortized Analysis:

- suppose just inserted string $aw$
- sitting on its leaf, which has parent
- Parent is only node that was (possibly) created by insertion:
  - As soon as walk down preexisting tree falls off tree, create parent node and stop
- invariant: every internal node except for parent of current leaf has suffix link to another explicit node
- plausible?
  - i.e., is there an explicit node for that suffix link to point at?
  - suppose $v$ was created as parent of $s_j$ leaf when it diverged from $s_k$
  - (note this is only way nodes get created)
  - claim $s_{j+1}$ and $s_{k+1}$ diverge at suffix($v$), creating another explicit node.
  - only problem if $s_{k+1}$ not yet present
  - happens only if $k$ is current suffix
  - only blocks parent of current leaf.

- insertion step:
  - suppose just inserted $s_i$
  - consider parent $p_i$ and grandparent (parent of parent) $g_i$ of current node
  - $g_i$ to $p_i$ link has string $w_1$
  - $p_i$ to $s_i$ link $w_2$
  - go up to grandparent
  - follow suffix link (exists by invariant)
  - fastfind $w_1$
  - claim: know $w_1$ is present in tree!
  - $p_i$ was created by $s_i$ split from a previous edge (or preexisted)
  - so $aww_1$ was in tree before $s_i$ inserted (prefix of earlier suffix)
  - so $ww_1$ is in tree after $s_i$ inserted
  - create suffix link from $p_i$ (preserves invariant)
  - slowfind $w_2$ (stopping when leave current tree)
  - break current edge if necessary (may land on preexisting node)
— add new edge for rest of $w_2$

Analysis:

• First, consider work to reach $g_{i+1}$ (not suf($g_i$))
  — Mix of fastfind and slowfind, but no worse then cost of doing pure slowfind
  — This is it most $|g_{i+1}| - |g_i| + 1$ (explain length notation)
  — So total is $O(\sum |g_{i+1}| - |g_i| + 1) = O(n)$
  — Wait: maybe $g_{i+1} - g_i + 1 < 0$, and I am cheating on sum?
    * Consider after inserting $s_{i+1} = \text{suf}(s_i)$
    * then $p_i$ can’t point at $s_{i+1}$
    * so must point at ancestor
    * so $g_i$ must point at ancestor of ancestor
    * i.e., $g_i$ points at $g_{i+1}$ or something higher
    * so $|g_{i+1}| \geq |g_i| - 1$

• Remaining cost: to reach $p_{i+1}$ (possibly implicit) from $g_{i+1}$.
  — If get there during fastfind, costs at most one additional step from $g_{i+1}$
  — If get there during slowfind, means fastfind stopped at or before $g_{i+1}$.
  — So suf($p_i$) is not below $g_{i+1}$.
  — So remaining cost (from $g_{i+1}$, not suf($p_i$)) is
    $$|g_{i+1}| - |p_{i+1}| \leq |\text{suf}(p_i)| - |p_{i+1}| \leq |p_i| - |p_{i+1}| + 1$$
    — telescopes as before to $O(n)$
  — we mostly analyzed as if used slowfind. when was fastfind important?
    * in case when $p_{i+1}$ was reached on fastfind step from $g_{i+1}$
    * ie, $p_{i+1} = \text{suf}(p_i)$
    * in that case, could not have afforded to do slowfind from $g_{i+1}$ to $p_{i+1}$
    * since slowfind analysis only covered fslowfind to $g_{i+1}$ and from $p_{i+1}$
    * however, don’t know that the case occurred until after the fact.

Ukonnen online version.
Suffix arrays: many of same benefits as suffix trees, but save pointers:
  • lexicographic ordering of suffixes
• represent as list of integers: $b_1$ is (index in text of) lexicographically first suffix, $b_2$ is (index of) lexicographically second, etc.

• search for pattern via binary search on this sequence

• some clever tricks (and some more space) let you avoid re-checking characters of pattern.

• So linear search (with additive $\log m$ for binary search.

• space usage: $3m$ integers (as opposed to numerous pointers and integers of suffix tree).

Applications:

• preprocess bottom up, storing first, last, num. of suffixes in subtree

• allows to answer queries: what first, last, count of $w$ in text in time $O(|w|)$.

• enumerate $k$ occurrences in time $O(w + |k|)$ (traverse subtree, binary so size order of number of occurrences (compare to rabin-karp).

• longest common subsequence is probably on homework.