CS 261: Data Structures
Week 10: Strings and graphs
Lecture 10a: Strings

David Eppstein
University of California, Irvine

Spring Quarter, 2022
Tries
If you enter a quoted string in Google, it will search for documents containing that phrase.

How does it do it? Not hashing. There are too many different phrases for it to hash all of them.

We need a data structure that can:
- Index an enormous text document or documents
- Quickly find all copies of a string (appearing without gaps)
- Uses small space, linear in document length
Some basic definitions

**Alphabet**: any finite set
- For text searching, might be ASCII or unicode characters
- For genomics, might be the four bases adenine, thymine, cytosine, and guanine (A, T, C, G) that encode genetic information in DNA
- For Google’s search problem, might be the words (rather than the individual letters) appearing in its documents

**Symbol**: a member of the given alphabet

**String**: a finite sequence of symbols

**Substring**: a connected subsequence of the sequence (might be empty, might be the whole sequence)
A simpler warm-up problem

Given a “dictionary”, a list of $n$ strings over some alphabet, handle queries that either ask whether a query string $q$ belongs to the dictionary, or that look up some extra information associated with the string (its definition).

Binary search tree: query time $O(\log n) \times \text{length}(q)$

Hash table: query time $\text{length}(q)$, randomized
Trie (or digital search tree)

Nodes = prefixes of given strings (substrings that start at the start)
Parent of a node = prefix with one less symbol
Leaves are words but not all words are leaves:
Is “pea” a word in this dictionary?
Separately storing each prefix would take too much space
Representing a trie

Add extra end-of-word symbol “$” to all words.

Label vertices by single symbol added to parent’s prefix (not by whole prefix).

For large alphabets, nodes can store hash table mapping alphabet symbols to corresponding children.
Querying a tree

Start at root of tree (empty prefix)
For each symbol $c$ of query $q + \$: 
   Go to the child whose label is $c$
   If no such child, $q$ is not in the dictionary
If search succeeded, $q$ is in the dictionary, and current leaf node is its representative in the tree

Time $= \text{length}(q)$

Space $= \text{number of tree nodes}$
$= \text{total length of all dictionary words}$
Compressed tries

When nodes have exactly one child, merge them with their child

Tree has \( n \) leaves, no nodes with one child \( \Rightarrow \leq 2n - 1 \) nodes

Represent substrings by pointer to string they come from + position and length \( \Rightarrow O(1) \) words of information per node
Searching a compressed trie

- Current node = root of compressed trie
- Current position in substring = 0
- For each symbol \( c \) in query \( q + $ \):
  - If current position < length of substring in current node, check that \( c \) matches the symbol at that position in the substring, and add 1 to current position
  - Otherwise, look for the child whose substring begins with \( c \) (by scanning all children or by using a hash table at the node mapping symbols to children), set current node to that child, and set current position = 0
- If any check fails or any child is not found, \( q \) is not in the dictionary; otherwise, it is represented in the dictionary by the current node
Suffix trees
Suffix trees

Suffix = substring of input string that includes the final dollar sign
Suffix tree = compressed trie of all suffixes

E.g., for input = “Mississippi”:

The suffixes in order by length are $, i$, pi$, ppi$, ippi$, sippi$, ssippi$, issippi$, sissippi$, ssissippi$, ississippi$, sissippi$, Mississipp$i$.

Tree leaves represent these same suffixes in alphabetical order.
Searching a suffix tree

Given a query string $q$, search character-by-character, just like in any other compressed trie.

But do not search for the $\$ after the end of $q$.

Search succeeds $\Rightarrow q$ is a substring of the input string.

The positions where $q$ appears in the input are the starting positions of the suffixes whose leaves are descendants of the node where the search finishes.
Possible solution to Google’s exact phrase search

Build suffix tree where input = all known web pages concatenated together, alphabet = words in each web page

Tree size: total number of words in all known web pages

Why words rather than characters: to reduce suffix tree size

Query time ≈ length of query phrase

Search result ⇒ positions where q appears in concatenation ⇒ web pages containing the given phrase
Other uses of suffix trees

Many other algorithmic problems on strings can be solved using suffix trees, for instance:

To find longest matching substrings from starting positions $x$ and $y$ in given string, find lowest common ancestor of leaves for suffixes starting at $x$ and $y$

To sort the suffixes of an input (used in Burrows–Wheeler data compression algorithm), traverse the tree using sorted order of characters at each node

To find longest common substring of two strings $s$ and $t$, build a suffix tree of $s$t and look for a node whose leaf descendants include suffixes starting in both strings and whose distance from the root (measured by string length) is maximum
How to construct the suffix tree?

Can be done in linear time + time to sort the alphabet


Details are very complicated. Rough outline:

- Recursively build compressed trie $T_o$ of suffixes starting at odd positions (almost the same as suffix tree for string of length $n/2$ whose alphabet is the set of consecutive pairs of symbols from the given string, sorted using radix sort)
- Represent suffixes at even positions as pairs $(c, s)$ where $c$ is a single character and $s$ is an odd suffix, and radix sort them
- Build compressed trie $T_e$ of even suffixes non-recursively, using sorted ordering of its leaves + tree navigation in $T_o$ to find heights of common ancestors of consecutive leaves
- Use tree navigation to merge $T_o$ and $T_e$ into a single tree