Pattern Matching part one

Suffix Trees

Benjamin Sach

Exact pattern matching

Input A text string $T$ (length $n$) and a pattern string $P$ (length $m$)

Goal: Find all the locations where $P$ matches in $T$

$P$ matches at location $i$ iff

for all $0 \leq j \leq m$ we have that $P[j] = T[i+j]$  

(tool strings are zero-indexed)

- A naive algorithm takes $O(nm)$ time
- Many $O(n)$ time algorithms are known (for example KMP)

Text indexing

Preprocess a text string $T$ (length $n$) to answer pattern matching queries...

After preprocessing, a query is a pattern $P$ (length $m$),

the output is a list of all matches in $T$.

- A naive algorithm takes $O(n)$ query time (using KMP)
- We want a query time which depends only on $m$ and occ
  - occ is the number of occurrences (matches)
- We also want $O(n)$ space and fast preprocessing (prep.) time

The atomic suffix tree

- The suffix tree contains every suffix of $T$ as a root to leaf path
- Every edge is labelled with a character from $T$
- No two edges leaving the same node have the same label
- Each leaf corresponds to a suffix (so there are $n$ leaves)

Searching in an atomic suffix tree

How do you find a pattern?

start at the root and walk down the tree

... matches occur at the leaves of the subtree

How do you find a pattern?

start at the root and walk down the tree

... matches occur at the leaves of the subtree
We can decide whether $P$ matches somewhere in $O(n)$ time (we’ll worry about outputting the matches later).

This may be fine in some applications (english text or DNA for example)

We can remove the assumption via the magic of hashing

An atomic suffix tree can have $(n/2) + 1^2$ nodes

this is far too big!
Compacted Suffix Tree of T
- A rooted tree with n leaves
- Every internal node has two or more children
- Every edge is labelled with a substring
- No two edges leaving the same node have the same first character
- Each leaf is labelled with a location in T
- Any root-to-leaf path spells out the corresponding suffix

Step one: Add a $ (unique symbol) to T

This is normally just called a suffix tree

Searching in a compacted suffix tree

How do you find a pattern?
- start at the root and walk down the tree
- ... matches occur at the leaves of the subtree

Naively constructing a compacted suffix tree

Insert the suffixes one at a time (longest first)
- Search for the new suffix in the partial suffix tree
- Add a new edge and leaf for the new suffix (this may require you to break an edge in two)

Suffix tree summary
- The (compacted) suffix tree of a (length n) text uses $O(n)$ space
- Finding all matches of a pattern $P$ of length m takes $O(m + occ)$ where occ is the number of matches
- Suffix trees can be built in $O(n)$ time
  but we have only seen the $O(n^2)$ time method

we assumed that the alphabet contained a constant number of symbols
The suffix array - a sneak preview

T
0 1 2 3 4 5 6
Sort the suffixes lexicographically

Suffix Array 1 3 5 0 2 4 6

The suffix array is much smaller than the suffix tree (in terms of constants)

Constructing the Suffix Array from the Suffix Tree

T
0 1 2 3 4 5 6

Suffix Array

recall that we added a unique symbol $\$\$ to make sure the tree exists
- the $\$\$ is the smallest symbol in the alphabet
To get the Suffix array perform a depth-first search (in lexicographical order)
this takes $O(n)$ time

Suffix tree summary

T
1 2 3 4 5 6

• The (compacted) suffix tree of a (length $n$) text uses $O(n)$ space
• Finding all matches of a pattern $P$ of length $m$ takes $O(m + \text{occ})$
  where occ is the number of matches
• Suffix trees can be built in $O(n)$ time
  but we have only seen the $O(n^2)$ time method

we assumed that the alphabet contained a constant number of symbols

Multiple text indexing

T
1 2 3 4 5 6

• build a generalised suffix tree in $O(n_1 + n_2)$ space
• using the linear time method (which we omitted), this takes $O(n_1 + n_2)$ time
• Finding all matches of a pattern $P$ of length $m$ still takes $O(m + \text{occ})$ time
  where occ is the number of matches