Exact pattern matching

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

\[
\begin{array}{c}
\text{T} \\
\text{a b c d a b a b a c b a} \\
\hline
\text{4} \\
\text{6} \\
\text{10}
\end{array}
\quad
\begin{array}{c}
\text{P} \\
\text{a b c d a b a b a c b a} \\
\hline
\text{8} \\
\text{6}
\end{array}
\]

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 \( P[j] = T[i + j] \)

- 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...

\[
\begin{array}{c}
\text{T} \\
\text{a b c d a b a b a c b a} \\
\hline
\text{4} \\
\text{6} \\
\text{10}
\end{array}
\]

After preprocessing, a query is a pattern \( P \) (length \( m \)), the output is a list of all matches in \( T \):

\[
\begin{array}{c}
\text{P} \\
\text{a b c d a b a b a c b a} \\
\hline
\text{8} \\
\text{6}
\end{array}
\]

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

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
  - matches occur at the leaves of the subtree

We can decide whether \( P \) matches somewhere in \( O(m) \) time

(we’ll worry about outputting the matches later)
There are at most \( n \) leaves.

Unfortunately there can be lots of internal nodes.

- 7 characters, 23 nodes. That’s not so bad, right?
- 9 characters, 36 nodes. This is far too big!

**Main Idea**: replace each non-branching path with a single edge.
- Edges are now labelled with substrings (instead of single characters).

**Compacted suffix trees**

1. There are at most \( n \) leaves.
2. Every internal node has two or more children.
   - So there are \( O(n) \) edges.
3. Don’t the edges take up lots of space?
   - We only store the end points.
   - We actually store \((4, 6)\).

**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.

**Uses** \( O(n) \) space.

**Normally just called a suffix tree**

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

**Uses** \( O(n) \) space.

**Searching in a compacted suffix tree**

To find a pattern in \( P \):

1. Start at the root and walk down the tree.
2. Matches occur at the leaves of the subtree.
3. How big is this subtree?
   - \( O(\text{occ}) \) because it has \( \text{occ} \) leaves.

We can find all the matches in \( O(m + \text{occ}) \) time (by looking at the whole 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 (as if you were matching a pattern)
- Add a new edge and leaf for the new suffix (this may require you to break an edge in two)

This takes $O(n)$ time per suffix... so $O(n^2)$ time in total.

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 $O(m + \text{occ})$
  where $\text{occ}$ is the number of matches
- We saw how to build a suffix tree in $O(n^2)$ time
  in fact they can be build in $O(n)$ time - but the method is much more involved
  we assumed that the alphabet contained a constant number of symbols

Multiple text indexing

How can we index multiple texts?
- 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 $\text{occ}$ is the number of matches