Suffix Trees

- A suffix tree is a data structure that exposes in detail the internal structure of a string.
- The real virtue of suffix trees comes from their use in linear time solutions to many string problems more complex than exact matching.
- Suffix trees provide a bridge between exact matching problems and matching with various types of errors.

Suffix Trees and pattern matching

- In off-line pattern matching one is allowed to process the text $T = T[0..n-1]$ in time $O(n)$, s.t., any further matching queries with unknown pattern $P = P[0..m-1]$ can be served in time $O(m)$.
- Compact suffix trees provide efficient solution to off-line pattern matching problem.
- Compact suffix trees provide also solution to a number of substring problems, periodicities and regularities.

Compact suffix trees - brief history

- First linear algorithm for constructing compact suffix trees in ‘73 by Weiner.
- More space efficient also linear algorithm was introduced in ‘76 by McCreight.
- An alternative, conceptually different (and easier) algorithm for linear construction of compact suffix trees was proposed by Ukkonen in ‘95.

Tries - trees of strings

- A trie $T$ for a set of strings $S$ over alphabet $A$ is a rooted tree, such that:
  - edges in $T$ are labeled by single symbols from $A$.
  - each string $s \in S$ is represented by a path from the root of $T$ to some leaf of $T$,
  - for some technical reasons (e.g., to handle the case when for some $s, w \in S$, $s$ is a prefix of $w$) every string $s \in S$ is represented in $T$ as $s \#$, where $\#$ is a special symbol that does not belong to $A$. 
**Tries - example**

- Strings in $S = \{a, aba, bba, abba, abab\}$ are replaced by $a\#$, $aba\#$, $bba\#$, $abba\#$, $abab\#$ respectively

**Suffix trees**

- A suffix tree $ST(w)$ is a trie that contains all suffixes of a given word $w$, i.e.,
- Similarly as it happens in tries ends of a suffixes are denoted by the special character $#$ which form leaves in $ST(w)$
- Moreover each internal node of the suffix tree $ST(w)$ represent the end of some substring of $w$

**Suffix Trees - example**

- Take $w = f_5 = babbabab$ (5th Fibonacci word)
- The suffixes of $w$ are
  - $b$ represented in $ST(w)$ as $b\#$
  - $ab$ represented in $ST(w)$ as $ab\#$
  - $bab$ represented in $ST(w)$ as $bab\#$
  - $abab$ represented in $ST(w)$ as $abab\#$
  - $babab$ represented in $ST(w)$ as $babab\#$
  - $bbabab$ represented in $ST(w)$ as $bbabab\#$
  - $abbabab$ represented in $ST(w)$ as $abbabab\#$
Compact suffix trees

- We know that suffix trees can be very large, i.e., quadratic in the size of an input string, e.g., when the input string has many different symbols.
- This problem can be cured if we encode all chains (paths with nodes of degree 2) in the suffix tree by reference to some substring in the original string.
- A suffix tree with encoded chains is called a **compact suffix tree**.

Compact suffix trees - example

**Theorem:** The size of a compact suffix tree constructed for any string \( w = w[0..n-1] \) is \( O(n) \)

- In the (compact) suffix tree there is only \( n \) leaves marked by \#s
- Since each internal node in the compact suffix tree is of degree \( \geq 2 \) there are \( \leq 2n-1 \) edges in the tree
- Each edge is represented by two indexes in the original string \( w \)
- Thus the total space required is **linear** in \( n \).

Longest repeated sequence

- Using a compact suffix tree for any string \( w = w[0..n-1] \) we can find the longest repeated sequence in \( w \) in time \( O(n) \).

```
procedure longest(v:tree; depth: integer);
    if v is not a leaf then
        if (depth > max-depth)
            then max-depth := depth;
        for each u \in v.children do
            longest(u, depth+length(v,u));
    max-depth := 0;
    longest(T.root,0);
    return(max-depth);
```

Find the deepest node in the tree which has degree at least 2

\[
\text{depth} \ x \ w[i+x-1] = w[j+x-1]
\]
Suffix trees for several strings

One can compute joint properties of two (or more) strings $w_1$ and $w_2$ constructing a single compact suffix tree $T$ for string $w_1w_2#$, where:
- Symbol $\$ does not belong neither to $w_1$ nor to $w_2$
- All branches in $T$ are truncated below the special symbol $\$

For example, using similar procedure one can compute the longest substring shared by $w_1$ and $w_2$.

Longest shared substring

Initially, for each node $v \in T$ we compute attribute $shares$, which says whether $v$ is an ancestor of leaves $\$ and $\#

function sharing(v:tree): set of {$, #}$
if $v$ is a leaf then
  return(v.symbol)
else
  set $\{\}$;
  for each $u \in v.children$ do
    set $\setunion sharing(u);$
    $v.shares \leftarrow set;$
    return($v.shares);$

... sharing($T.root);$

Looking for substrings and reversals

In some biological applications we are interested in finding substrings and their reverses (reversed strings).
E.g., in prediction of potential hair-pin loops, stems, pseudo-knots, etc in secondary structure of RNA sequences we focus on close substrings and their complementary reverses.
The problem can be solved via search for shared substrings in a compact suffix tree constructed for the string $w$co-w, where $w$ is any RNA sequence and co-w is w in which A,T and G,C are swapped in pairs respectively.
Lowest common ancestor - LCA

- A node \( z \) is the lowest common ancestor of any two nodes \( u, v \) in the tree \( T \) rooted in the node \( r \), \( z = \text{lca}_T(u,v) \), iff:
  1) node \( z \) belongs to both paths from \( u \) to \( r \) and from \( v \) to \( r \)
  2) node \( z \) is the deepest node in \( T \) with property 1)

\[ z = \text{lca}(u,v) \]

Theorem: Any tree of size \( n \) can be preprocessed in time \( O(n) \), such that, the lowest common ancestor query \( \text{lca}(u,v) \), for any two nodes \( u, v \) in the tree can be served in \( O(1) \) time.

- For example, we can preprocess any suffix tree in linear time and then compute the longest prefix shared by any two suffixes in \( O(1) \) time.
- LCA queries have also many other applications.

Pattern matching with \( k \) mismatches

- So far we discussed algorithmic solutions either for exact pattern matching or pattern matching with don’t care symbols, where the choice of text symbols was available at fixed pattern positions
- In pattern matching with \( k \) mismatches we say that an occurrence of the pattern is acceptable if there is at most \( k \) mismatches between pattern symbols and respective substring of the text

acceptable pattern occurrence

at most \( k \) mismatches

Pattern \( P \)

Text \( T \)

matching substrings
Pattern matching with k mismatches

- As many other instances of pattern matching also in this case one can provide an easy solution with time complexity $O(m \cdot n)$. However we are after faster solution.
- The search stage in pattern matching with $k$ mismatches is preceded by the construction of a compact suffix tree $ST$ for the string $P$T#
- The tree $ST$ is later processed for LCA queries which will allow to fast recognition of matching substrings $S_i$
- Both steps are performed in linear time

Suffix arrays

- One of the very attractive alternatives to compact suffix trees is a suffix array
- For any string $w = w[0..n-1]$ the suffix array is an array of length $n$ in which suffixes (namely their indexes) of $w$ are sorted in lexicographical order
- The space required to compute and store the suffix arrays is smaller, the construction is simpler, and the use/properties are comparable with suffix trees

Pattern matching with k mismatches

- During the search stage each text position is tested for potential approximate occurrence of the pattern $P$
- Consecutive blocks $S_i$ are recovered in $O(1)$ time via LCA queries in preprocessed $ST$ tree at most $k$ times, which gives total complexity $O(kn)$.

Suffix arrays - example

Original string $w = b \ a \ b \ b \ a \ b \ a \ b$
Suffix array $w = 6 \ 4 \ 1 \ 7 \ 5 \ 3 \ 0 \ 2$

- Suffix arrays provide tools for off-line pattern matching in time $O(m \cdot \log n)$, where $n$ is the length of the text and $m$ is the length of the pattern
- There exists linear transformation between suffix trees and suffix arrays
- Suffix arrays provide simple and efficient mechanism for several text compression methods