Outline

1. Introduction
   - Suffix Trees

2. Bundled Suffix Trees
   - Encoding Approximate Information
   - Definition
   - Size and Construction

3. An application
   - Computing Surprise Measures
   - Summary
A Suffix Tree is a data structure revealing the internal structure of a string. They occupy $O(n)$ space and can be built in $O(n)$ time.

They are efficient for:
- **Exact String Matching**
- **Longest Exact Common Substring Problem**
- **Identifying Exactly Repeated Patterns**
Limitations of Suffix Trees

Suffix Trees cannot deal naturally with approximate string matching problems. (Hamming or Edit distance)

Two difficult problems:
- Longest Common Approximate Substring Problem
- Extraction of approximately repeated patterns


Extending Suffix Trees

THE TARGET

*Extending Suffix Trees* in order to solve *in a simple way* some classes of *approximate string matching problems.*

Bundled Suffix Trees

*Bundled Suffix Trees* extend suffix Trees.

- They incorporate *approximate information*;
- They can be used *like Suffix Trees* for:
  - Longest Common Approximate Substring Problem
  - Extraction of approximately repeated patterns
Approximate Matching

Character matching is a relation among letters (in fact, it is the equality relation)

We model *approximate matching* as a non-transitive relation among letters:

two strings “match” if all their letters are in relation.
Approximate Matching

Character matching is a relation among letters (in fact, it is the equality relation).

We model *approximate matching* as a non-transitive relation among letters:

*two strings “match” if all their letters are in relation.*
Non-Transitive Relation: An Example

Modeling a relation based on Hamming Distance

- Start from a basic alphabet (e.g. binary: \( A = \{0, 1\} \))
- Construct an alphabet composed of macrocharacters (e.g. \( \overline{A} = \{00, 01, 10, 11\} \))
- Two letters \( x, y \in \overline{A} \) are in relation if and only if \( d_H(x, y) \leq D \) (e.g. \( D = 1 \)).

The Relation Graph

- Relation is non-transitive
- It encapsulates a (restricted) form of distance.
Bundled Suffix Tree: An Example

We start from the suffix tree for the string.
Let’s compare suffix 3 and suffix 1:

After bcabb in the tree, we put a red node with label 3.
Due to symmetry, there is also a red node with label 1 after abbab.
Bundled Suffix Tree: An Example

We start from the suffix tree for the string.

Let’s compare suffix 3 and suffix 1:

```
| b | c | a | b | b | a | b | c |
```

After `bcabb` in the tree, we put a red node with label 3.

Due to symmetry, there is also a red node with label 1 after `abbab`.
Introduction

Bundled Suffix Trees

An application

Bundled Suffix Tree: An Example

We start from the suffix tree for the string.

Let’s compare suffix 3 and suffix 1:

After bcabb in the tree, we put a red node with label 3.

Due to symmetry, there is also a red node with label 1 after abbab.
Bundled Suffix Tree: An Example

We start from the suffix tree for the string.

Let’s compare suffix 3 and suffix 1:

After bcabb in the tree, we put a red node with label 3.

Due to symmetry, there is also a red node with label 1 after abbab.
Bundled Suffix Tree: An Example

If we do this process for every couple of suffixes, we build a Bundled Suffix Tree!

Note that this data structure is in the middle between a suffix tree and a suffix trie.
Bundled Suffix Tree: An Example

Bundled Suffix Trees can be used to:

- solve the Longest Common Approximate Substring Problem with respect to a given relation (just find the lowest red node).
- extract information about approximately repeated patterns.
The number of red nodes inserted depends on:
- the relation
- the structure of the text.

In the worst case, the number of red nodes is quadratic in the length of the text $S$. (Example)

On average, the number of red nodes is limited by

$$m^{1+\delta}, \quad \delta = \log_{1/p^+} C.$$  

( $m$ is the length of the text, $p^+$ is the normalized frequency of the most common letter in $S$, $C$ depends on the relation)

$1 + \delta$ is slightly greater than one! (Example)
How Fast?

Naive Algorithm

- The naive algorithm for building a BuST tries to “match” every suffix of the text along every branch of the suffix tree, until a “mismatch” is found.
- It can be quadratic in the worst case.
- An analysis based on the average shape of a suffix tree shows that its average complexity is bounded by $m^{1+\delta'}$ ($\delta'$ just slightly greater that $\delta$).

Faster

Efficient Algorithm
We found an “McCreight-like” algorithm that is linear in the size of the output.

Intuitions
- It processes the suffixes backwards.
- It is based on the concept of inverse suffix links.
- It identifies the red nodes for suffix $i$ by processing the red nodes for suffix $i + 1$. 
Experimental Results

- We have implemented the naive algorithm for the construction of BuST.
- We have tested it with relations induced by hamming distance, defined over DNA-macrocharacters.
- With macrocharacters of size 4 ($X \leftrightarrow Y \iff d_H(X, Y) \leq 1$) the algorithm can process texts of length 100K in few seconds.
- The number of red nodes grows tamely.
Measures of surprise: exact case

**z-score**

\[ \delta(\alpha) = \frac{f(\alpha) - E(\alpha)}{N(\alpha)} \]

- \(f(\alpha)\) is the observed frequency of \(\alpha\)
- \(E(\alpha)\) is the expected frequency of \(\alpha\)
- \(N(\alpha)\) is a normalization factor (e.g. the variance or its first-order approximation).

**Monotonicity**

- If \(f(\alpha) = f(\alpha\beta)\) then \(\delta(\alpha) \leq \delta(\alpha\beta)\).
- \(\delta\) needs to be computed only for *maximal strings* at a fixed frequency. These are exactly the strings ending at nodes of the Suffix Tree.
Computing the z-score

Using a Suffix Tree, we can compute and store the z-score for all “interesting” substrings of a given text in linear time and space (given that we can compute $E$ and $N$ in linear time and space).

Let’s consider as occurrences of $\beta$ in $\alpha$ all the substrings $\beta'$ that are in relation with $\beta$.

Reasoning as in the exact case, we can use a BuST to compute the z-score for all interesting substrings of $\alpha$ in time and space proportional to the BuST’s size.
If we use an **Hamming-like relation** built on macrocharacters, we are counting all the occurrences of a string with *distance bounded by a threshold proportional to the string’s length*.

### Pros and Cons

**Pros:**
- the algorithm runs in time proportional to the number of maximal substrings (w.r.t. $\delta$).
- BuST provides a compact way to store and retrieve this information.

**Cons:**
- the macrocharacters introduce **rigidity** (we can count compute the z-score only for strings of length multiple of the macrocharacter’s size).
- the distance must be **distributed evenly** among macrocharacters.
Conclusions

- We have introduced **Bundled Suffix Trees**, a new data structure extending suffix trees.
- Given a relation among characters encoding some sort of approximate information, a BuST reveals the inner structure of the strings w.r.t. this relation (all this information is internal w.r.t. the processed string).
- BuST can be used for all the problems related to the inner structure of the string, like computation of approximated frequency.
- The structure is based on a very general concept of non-transitive relation among characters. The use of Hamming-like relation on tuples is just a possible example.
- Its size is slightly more than linear on average, and there’s a fast (McCreight-like) algorithm to build it.
Let’s consider the text
\[ \underbrace{a\ldots a}_{m} \underbrace{c\ldots c}_{m} \underbrace{b\ldots b}_{2m}, \]
over \( \{a, b, c, d\} \), with

\[ a \leftrightarrow b \]
\[ d \leftrightarrow c \]

The number of nodes surrounded by the red box is quadratic in \( m \)!
Let’s consider the text

\[
\underbrace{a \ldots a}_{m} \underbrace{c \ldots c}_{m} \underbrace{b \ldots b}_{2m},
\]

over \(\{a, b, c, d\}\), with

\[
\begin{align*}
a & \leftrightarrow b \\
\uparrow & \quad \uparrow \\
d & \leftrightarrow c
\end{align*}
\]

The number of nodes surrounded by the red box is quadratic in \(m\)!
Let’s consider the text 

\[ a \ldots a \ c \ldots c \ b \ldots b, \]

over \( \{a, b, c, d\} \), with

\( a \leftrightarrow b \)

\( d \leftrightarrow c \)

The number of nodes surrounded by the red box is quadratic in \( m \)!
The exponent $\delta$
The exponent $\delta$
Number of macrocharacters of length 4 over DNA alphabet. Test strings are generated according to a uniform p.d.
A crucial role in the fast construction of suffix trees is played by **suffix links**.

- Suffix links are pointers from nodes with path label $x\alpha$ to nodes with path label $\alpha$.
- Whenever there is a node with path label $x\alpha$, there’s also a node with path label $\alpha$. 
Inverse Suffix Links

- Inverse suffix links are pointers from nodes with path label $\alpha$ to positions in the tree labeled $x\alpha$, for each $x$ in the alphabet such that $x\alpha$ is a substring of $S$.
- They can point in the middle of an arc.
- If a ISL takes from $\alpha$ to $x\alpha$, it is labeled with $x$. 
The Algorithm

- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix* $S[i+1]$, using *Inverse Suffix Links*.

- Suppose a red node for suffix $S[i+1]$ is just under a “black” node with path label $\alpha$.

- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.

- With a skip and count trick, we can identify the positions of red nodes for $S[i]$. 
The Algorithm

- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix* $S[i + 1]$, using *Inverse Suffix Links*.

- Suppose a red node for suffix $S[i + 1]$ is just under a “black” node with path label $\alpha$.

- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.

- With a skip and count trick, we can identify the positions of red nodes for $S[i]$. 

---

**Efficient Algorithm**

- Dimension of BuST
The Algorithm

- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix $S[i + 1]$*, using *Inverse Suffix Links*.
- Suppose a red node for suffix $S[i + 1]$ is just under a “black” node with path label $\alpha$.
- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.
- With a *skip and count trick*, we can identify the positions of red nodes for $S[i]$. 