String Indexing for Patterns with Wildcards

Philip Bille\textsuperscript{1}, Inge Li Gørtz\textsuperscript{1}, Hjalte Wedel Vildhøj\textsuperscript{1}, and Søren Vind\textsuperscript{1}

\textsuperscript{1}Technical University of Denmark, DTU Informatics

SWAT 2012, Helsinki
July 6, 2012
String Indexing for Patterns with Wildcards

Problem Definition

Build an index for a string $t \in \Sigma^*$, that, given a query pattern $p$, quickly can report where $p$ occurs in $t$.

$$p = p_0 \ast p_1 \ast \ldots \ast p_j$$

Example

$$t = \text{combinatorialpatternmatching}$$

$$p = \ast \text{at}\ast\ast\ast\text{n}$$
Two Simple Solutions

Suffix Tree Search

\[ p = \texttt{*na*} \]

\[ t = \texttt{bananas} \]
Two Simple Solutions

Suffix Tree Search

\[ p = \ast \text{na}\ast \]

\[ t = \text{bananas} \]
Two Simple Solutions

Suffix Tree Search

\[ p = \ast \text{na}\ast \]

\[ t = \text{bananas} \]
Two Simple Solutions

Suffix Tree Search

\[ p = *na* \]

\[ t = bananas \]
Two Simple Solutions

Suffix Tree Search

\[ p = *\text{na}* \]

\[ t = \text{bananas} \]
Two Simple Solutions

Suffix Tree Search

\[ p = *na* \]

\[ t = \text{bananas} \]
Two Simple Solutions

Suffix Tree Search

\[ p = *na* \]

\[ t = bananas \]

Time: \( O(\sigma^j m + \text{occ}) \)

Space: \( O(n) \)
Two Simple Solutions

Simple Linear Time Index

\[ \text{Time: } O(m + j + \text{occ}) \]
\[ \text{Space: } O(nk + 1) \]
Two Simple Solutions

Simple Linear Time Index

Time: $O(m + j + \text{occ})$
Space: $O(nk + 1)$
Two Simple Solutions

Simple Linear Time Index

Time: $O(m + j + \text{occ})$

Space: $O(nk + 1)$
Two Simple Solutions

Simple Linear Time Index

```
Time: \(O(m + j + \text{occ})\)
Space: \(O(nk + 1)\)
```
Two Simple Solutions

Simple Linear Time Index

Time: $O(m + j + \text{occ})$
Space: $O(nk + 1)$
Two Simple Solutions

Simple Linear Time Index

$\text{bananas}^2$

$p = *na*$
Two Simple Solutions

Simple Linear Time Index

$P = *na*$

Time: $O(m + j + occ)$
Space: $O(n^{k+1})$
LCP Queries

Let $C_i$ be a set of substrings of the indexed string. Consider the following query on the compressed trie $T(C_i)$ storing the strings in $C_i$.

\[ \text{LCP}(x, i, \ell): \text{The location where the search for } x \in \Sigma^* \text{ stops when starting in location } \ell \in T(C_i). \]

**Example:** $x = \text{angry}$ and $C_i = \text{suff(bananas)}$. 

---

The Longest Common Prefix Data Structure

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

The Longest Common Prefix Data Structure

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

How fast can you answer an LCP query?

- $O(\log \log n)$ time and $O(n \log n)$ space.
  - Index with query time $O(m + \sigma^j \log \log n + \text{occ})$ and space $O(n \log n)$.

- We show that you can also do $O(\log n)$ time and $O(n)$ space.
  - Index with query time $O(m + \sigma^j \log n + \text{occ})$ and space $O(n)$.

---

The Longest Common Prefix Data Structure

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

How fast can you answer an LCP query?

- \( O(\log \log n) \) time and \( O(n \log n) \) space.
  \( \Rightarrow \) Index with query time \( O(m + \sigma^j \log \log n + \text{occ}) \) and space \( O(n \log n) \).

- We show that you can also do \( O(\log n) \) time and \( O(n) \) space.
  \( \Rightarrow \) Index with query time \( O(m + \sigma^j \log n + \text{occ}) \) and space \( O(n) \).

---

Solution 1

An Unbounded Wildcard Index
Using Linear Space

Query Time: \( O(m + \sigma^j \log \log n + \text{occ}) \)
Space Usage: \( O(n) \)
Definition:

- A bottom tree is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the top tree.

Example: A tree with $n = 16$ leaves ($\log n = 4$).
An Unbounded Wildcard Index Using Linear Space

ART Decomposition

Definition:
- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

**Example:** A tree with $n = 16$ leaves ($\log n = 4$).

---

$B_1$

---

$\text{2. S. Alstrup, T. Husfeldt, and T. Rauhe}$

Definition:

- A bottom tree is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the top tree.

Example: A tree with $n = 16$ leaves ($\log n = 4$).

---

2. S. Alstrup, T. Husfeldt, and T. Rauhe
An Unbounded Wildcard Index Using Linear Space

ART Decomposition

Definition:
- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

**Example:** A tree with $n = 16$ leaves ($\log n = 4$).

**Property:** The top tree has $O\left(\frac{n}{\log n}\right)$ leaves.

---

2 S. Alstrup, T. Husfeldt, and T. Rauhe
Obtaining the Index

- Use the ART decomposition to decompose the suffix tree into a number of logarithmic sized bottom trees and a single top tree containing $O\left(\frac{n}{\log n}\right)$ leaves.
- Store the top and bottom trees in LCP data structure.
- On the top tree $T'$: Add support for $O(\log \log n)$ time LCP queries using the method by Cole et al.\(^3\)
  - This requires space $O(|T'| \log |T'|) = O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) = O(n)$.
- On the bottom trees $T(C_1), \ldots, T(C_q)$: Add support for $O(\log n)$ time LCP queries using our new method.
  - This requires $O\left(\sum_{i=1}^{q} |C_i|\right) = O(n)$ space.
  - The query time becomes $O(\log |C_i|) = O(\log \log n)$.

This gives an unbounded wildcard index using $O(n)$ space with query time $O(m + \sigma^j \log \log n + \text{occ})$.

---

\(^3\) R. Cole, L. Gottlieb, and M. Lewenstein. 
Solution 2

A Time-Space Trade-Off for $k$-Bounded Wildcard Indexes

Query Time: $O(m + \beta^j \log \log n + occ)$

Space Usage: $O(n \log_{\beta}^{k-1}(n) \log n)$
General Idea

Reduce the branching factor of the suffix tree search from $\sigma$ to $\beta$ by creating wildcard trees. Query time: $O(m + \beta^j \log \log n + \text{occ})$ when using the LCP data structure.
A Time-Space Trade-Off for Bounded Wildcard Indexes

General Idea

Reduce the branching factor of the suffix tree search from $\sigma$ to $\beta$ by creating wildcard trees. Query time: $O(m + \beta^j \log \log n + occ)$ when using the LCP data structure.
General Idea

Reduce the branching factor of the suffix tree search from $\sigma$ to $\beta$ by creating wildcard trees. Query time: $O(m + \beta^j \log \log n + \text{occ})$ when using the LCP data structure.
A Time-Space Trade-Off for Bounded Wildcard Indexes

General Idea

Reduce the branching factor of the suffix tree search from $\sigma$ to $\beta$ by creating wildcard trees. Query time: $O(m + \beta^j \log \log n + \text{occ})$ when using the LCP data structure.
Each string in $T(C)$ gives rise to at most $\text{lightdepth}(x) \leq \log_\beta n$ strings on the next level. So the number of strings in a $k$-level index is at most

$$\sum_{i=0}^{k} n \log^i_\beta n = O(n \log^k_\beta n) .$$

By using the LCP data structure to support LCP queries on every subtrie, we obtain a $k$-bounded wildcard index with query time $O(m + \beta^j \log \log n + \text{occ})$ using space $O(n \log^{k-1}_\beta (n) \log n)$.
Solution 3
A $k$-Bounded Wildcard Index with Linear Query Time

Query Time: $O(m + j + \text{occ})$
Space Usage: $O(n \sigma k^2 \log^k \log n)$
A \( k \)-Bounded Wildcard Index with Linear Query Time

**General Idea**

Consider the previously described unbounded wildcard index \( \mathcal{A} \) with

- linear space usage, and
- query time \( O(m + \sigma^j \log \log n + \text{occ}) \).

Suppose the pattern is restricted to contain a maximum of \( k \) wildcards.

- If \( m + j > \sigma^k \log \log n > \sigma^j \log \log n \), (i.e., the query pattern is long) the query time becomes linear: \( O(m + j + \text{occ}) \).
- If \( m + j \leq \sigma^k \log \log n \), we query a special wildcard index \( \mathcal{B} \) for short patterns with query time \( O(m + j + \text{occ}) \).

In any case the query time is \( O(m + j + \text{occ}) \). The space used by the index is \( O(|\mathcal{A}| + |\mathcal{B}|) \).
A $k$-Bounded Wildcard Index with Linear Query Time

A Special Index for Patterns Shorter than $\sigma^k \log \log n$

$G = \sigma^k \log \log n$

$T^k_1(\operatorname{pref}_G(C))$ contains at most $n$ strings. Consider a string $x$ in one of the subtries. At most $|x| \leq G$ suffixes of $x$ appear in tries on the next level. Consequently, the number of strings in $T^k_1(\operatorname{pref}_G(C))$ is bounded by

$$\sum_{i=0}^{k} nG^i = O(n(\sigma^k \log \log n)^k) = O(n\sigma^{k^2} \log^k \log n).$$

**Result:** A $k$-bounded wildcard index with linear query time $O(m + j + \text{occ})$ using space $O(n\sigma^{k^2} \log^k \log n)$. 
Conclusions

- Three new solutions for string indexing for patterns with wildcards:
  - The fastest linear space index.
  - A trade-off for $k$-bounded wildcard indexes.
  - The first non-trivial linear time index.

- All solutions generalize to string indexing for patterns with variable length gaps.
Conclusions

- Three new solutions for string indexing for patterns with wildcards:
  - The fastest linear space index.
  - A trade-off for $k$-bounded wildcard indexes.
  - The first non-trivial linear time index.

- All solutions generalize to string indexing for patterns with variable length gaps.

Thank you!