# Substring Compression Problems

Graham Cormode
cormode@bell-labs.com

S. Muthukrishnan
muthu@cs.rutgers.edu

# Overview

- Recent applications of compression
- Define “substring compression problems”
- Give exact and approximate algs for substring compression problems under Lempel-Ziv
- Run out of time
- Stop abruptly

# Introduction

- Text compression is part of most algorithms courses
- Basic problem: given text T, produce C(T), compressed version of T, which can be decompressed: D(C(T)) = T
- Some variations have been studied, eg, searching compressed texts, compressed text indexes.
- A variety of recent applications...

# Use 1: Kolmogorov Complexity

Compression programs used as a surrogate for Kolmogorov Complexity:

- Kolmogorov Complexity of a string is smallest possible algorithmic description.
- But this is uncomputable.
- Compressed version of a string attempts to be smallest possible efficiently computable description.
- So in practice use compressed size. [Li and Vitanyi]

# Use 2: Biological Sequences

In Bioinformatics, people have designed compression methods for DNA sequences etc.

Different parts show different compressibility: coding regions are hard to compress, “junk DNA” more compressible.

Methods are either off-the-shelf compressors, or extensions of these to add plausible operations (reverse-copies etc.)

# Use 3: Sequence Comparison

A heuristic idea: given sequences X and Y, compute |C(XY)| - |C(X)| as a measure of similarity of X & Y (Y compressed in context of X)

Applied in practice with some success. [Benedetto, Caglioti, Loreto 02]

Explained in terms of relative Kolmogorov complexity [Li, Chen, Li, Ma, Vitanyi 03] and approximation of combinatorial distances [Ergun, Muthukrishnan, Sahinalp 03]

*Proposed by physicists, used by biologists, explained by computer scientists*

# Substring Applications

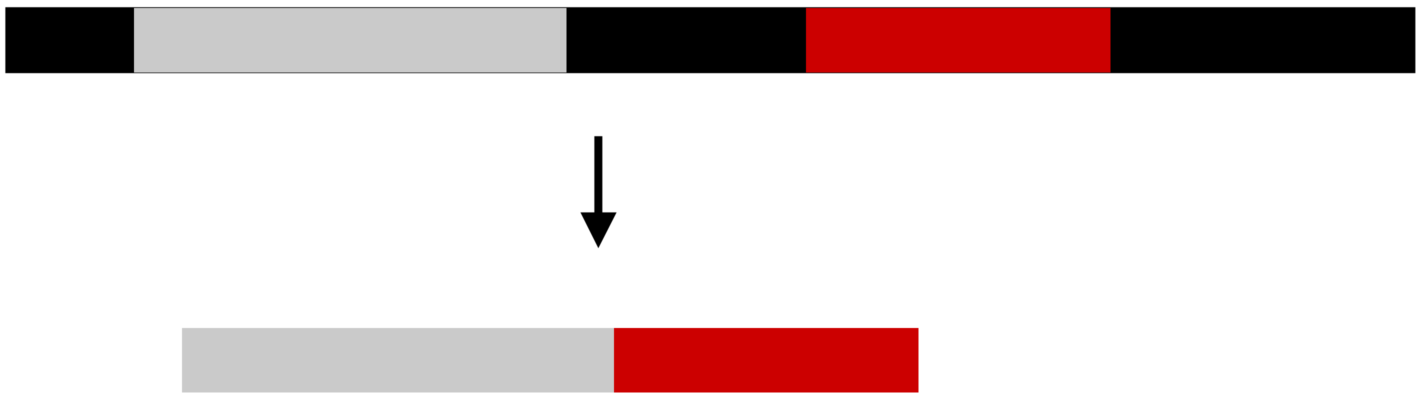
In most previous applications, compression has been applied at whole string level, but can also be used for substrings:

- Estimate Kolmogorov complexity of substrings (find most complex substring)
- Compute compressed version of substrings of Biological sequences (find subsection of interest)
- Find compressed size of substring using another as initial dictionary (gives distance between substrings)

# Substring Compression

Gives a new direction in stringology: substring compression problems.

Fix a compression method C, and given string S, we can ask a variety of questions:



# Substring Compression Query

After efficient preprocessing of string $S$:

**Substring Compression Query (SCQ):** Given ($i$, $j$) compute the compressed representation of $S[i, j]$, $C(S[i,j])$.

**Substring Compression Size Query (SCSQ):** Given ($i$, $j$), compute $|C(S[i,j])|$

**Generalized Substring Compression Query (GCSQ):** Given ($\alpha$, $\beta$, $i$, $j$) compute the compressed version of $S[i, j]$ in the context of $S[\alpha, \beta]$.

# Substring Compression Query

Two trivial solutions for SCQ:

(1) Preprocess all (i, j) pairs and store answer.
Preprocessing $O(|S|^2)$, query time $O(|C(S[i,j])|)$.

(2) Compute compressed version on demand.
Preprocessing: $O(1)$, Query time $O(|S|)$.

Queries need $\Omega(|C(S[i,j])|)$ time to output result

Goal is therefore $o(|S|^2)$ preprocessing, and $o(|S|)$ time for queries.

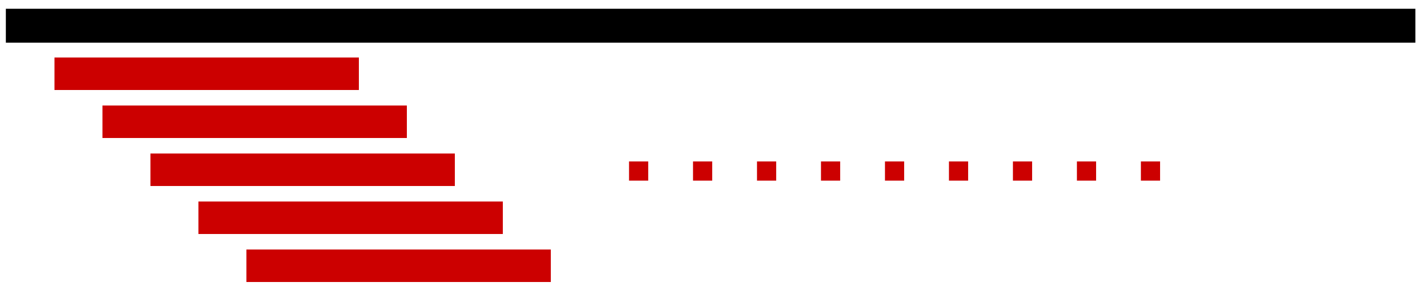
# Least Compressible Substring

Given string $S$ and value $\lambda$:

**Least Compressible Substring (LCS):** Find $i$ so $|C(S[i, i+\lambda-1])| = \max_j |C(S[j, j+\lambda-1)|$

**Generalized Least Compressible Substring (GLCS):** Given $\alpha$, $\beta$ find least compressible substring in context of $S[\alpha, \beta]$.

Most Compressible Substring is similar.



# Compression Method

Choice of compression method is vital.

Simple methods eg Run Length Encoding, Huffman Encoding, have mostly trivial solutions.

We will focus on Lempel-Ziv and variants:

**LZSS**: Given string S, greedily parse left-to-right the longest substring that occurs earlier in string (or single character).

Compressed size counts the number of phrases.

# Our Results

- Exact algorithms for SCQ.
  $O(|S| \log |S|)$ preprocessing, poly-log time to produce each phrase in $C(S[i,j])$.
- Constant factor approximation of LCS in time $O(|S| \lambda / \log \lambda)$.
- Poly-log factor approximation of LCS and SCSQ
  $O(|S| \log^2 |S|)$ preprocessing, $O(1)$ per query

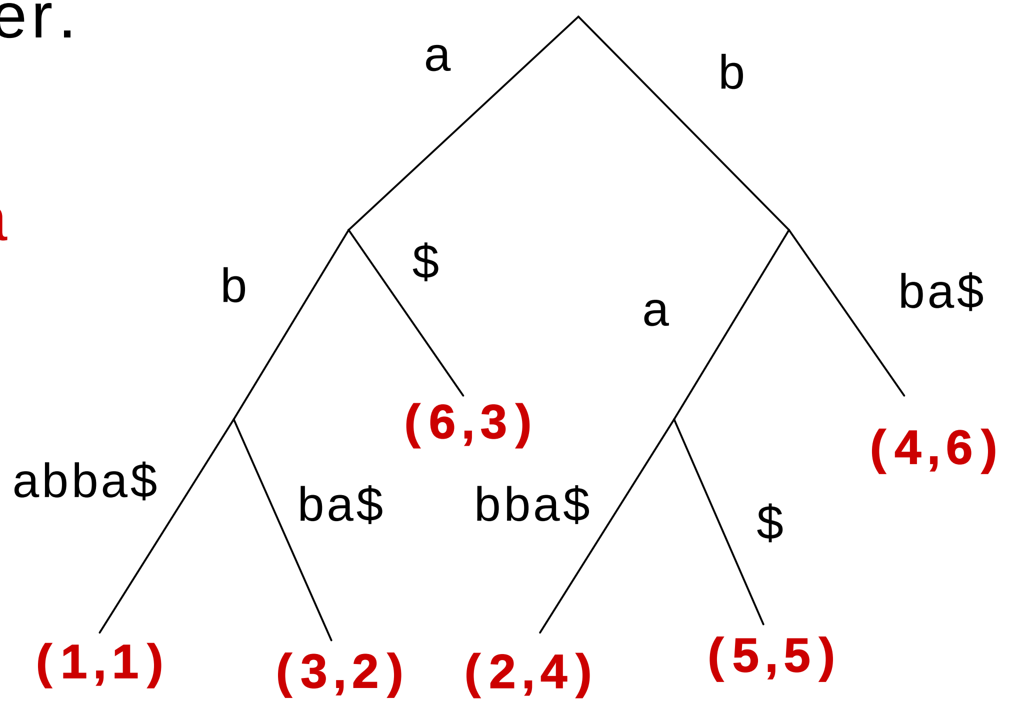
# Exact Solutions for SCQ

Build the suffix tree for S$.

Note that there is a bijection between suffixes $S_j$ = S[j, |S|] and the leaves of the suffix tree.

Label the leaf for $S_j$ with j and its position in the lexicographic order.

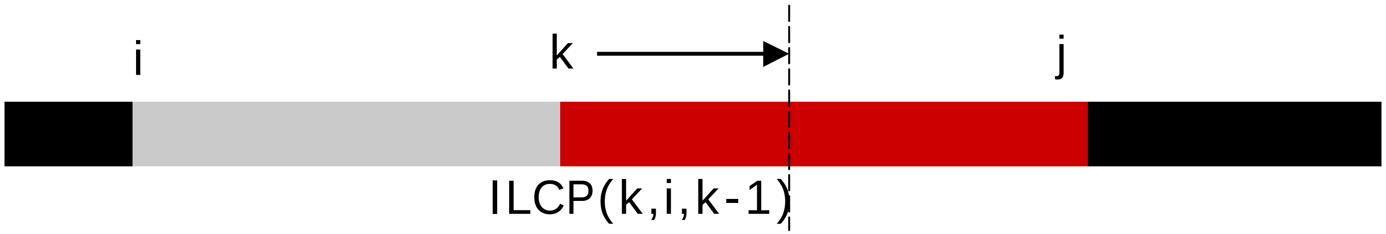
S=ababba



# Interval Longest Common Prefix

We define the Interval Longest Common Prefix (ILCP) as the longest common prefix of $S_k$ and suffixes $S_l$ ... $S_m$ ($l < m$)

Using ILCP repeatedly, answer SCQ(i,j):




```
k=i;
  repeat
        ILCP = ILCP(k,i,k-1)
        output ILCP
        k ← k + |ILCP|
  until k>j
```

# Reduction

Split ILCP into two parts:

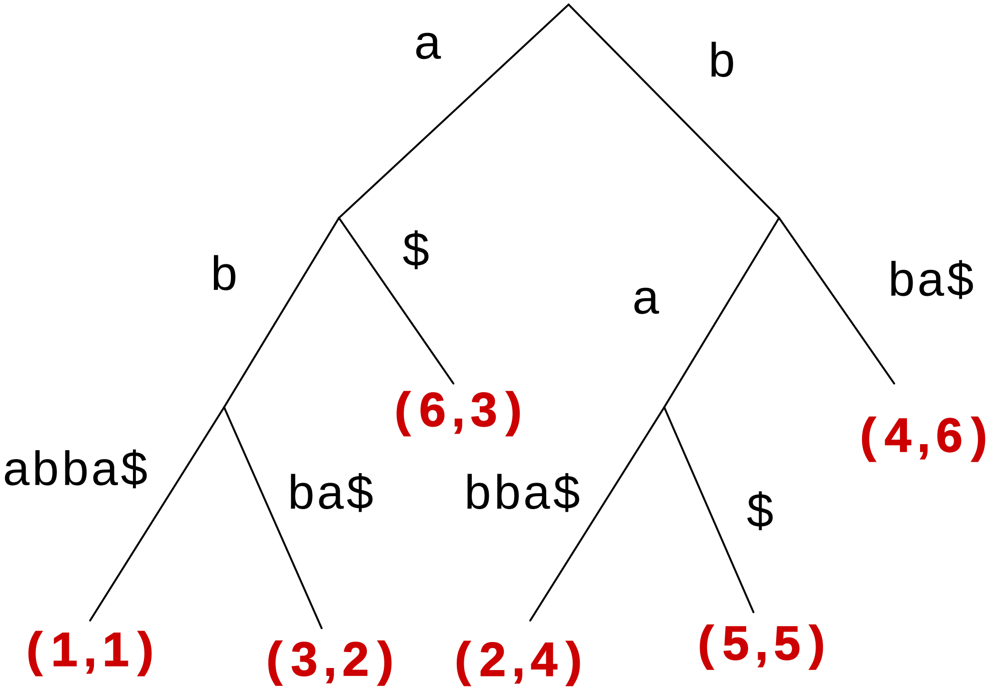
- ILCP that is (lexicographically) greater than $S_k$
- ILCP that is smaller than $S_k$

Focus on the latter, since former is symmetric.

Suppose $S_k$ is labeled (k,p). The longest matching suffix is the one labeled (a, b) where $a \in [l, m]$ and b is as large as possible but $< p$.

Range searching: query for pairs $\in ([l, m], [b, p])$, binary search on b to find greatest. Use least common ancestor (LCA) in tree to find length.

# Example




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | X | | | | | |
| 2 | | | X | | | |
| 3 | | | | | | X |
| 4 | | X | | | | |
| 5 | | | | | X | |
| 6 | | | | X | | |

↓

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | X | | | | | |
| 2 | | | X | | | |
| 3 | | | | | | X |
| 4 | | X | | | | |
| 5 | | | | | X | |
| 6 | | | | X | | |

ILCP(5,2,4) answered by range searching for pair (x, y) with y < 5, x ∈ [2, 4].

Solution is (2, 4) whose LCA with (5,5) is ba = $S_2[2]$.

# Cost

Preparing data structures for ILCP:

Build Suffix Tree, LCA $O(|S| \log |\Sigma|)$

Range search structure $O(|S| \log |S|)$

Each ILCP costs $O(\log |S|)$ range queries.

Total number of ILCPs = $|C(S[i,j])|$.

Overall cost per SCQ:

$O(|C(S[i,j])| \log|S| \log \log|S|)$

ie poly-log factor over optimal
(for small $|C(S[i,j])|$ )

# Approximate Solutions

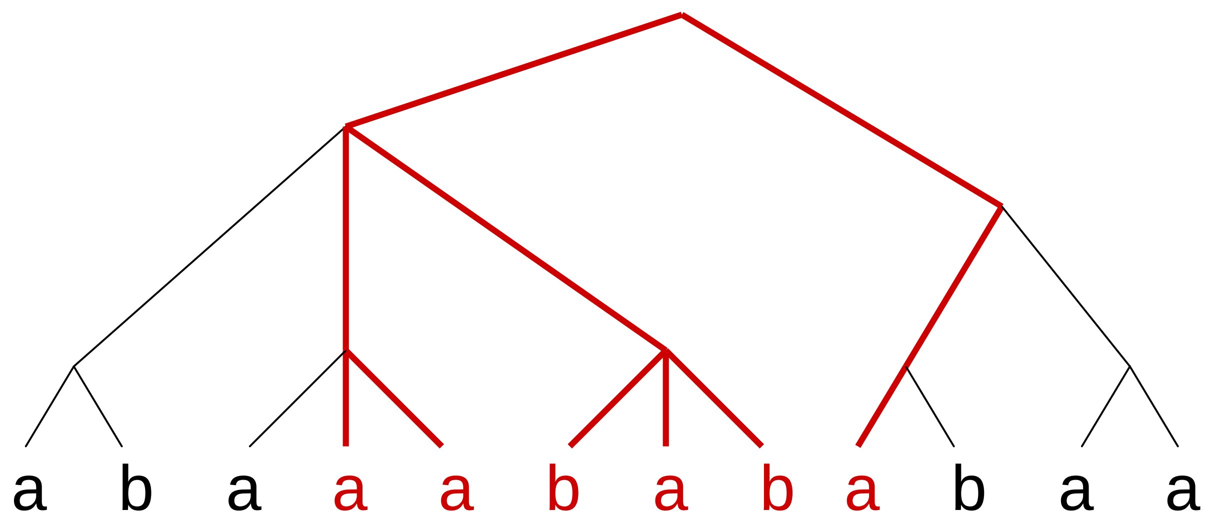
We can find approximate solutions to substring compression problems: either approximating the length of SCQ, or finding a substring which is approximately the LCS.

Techniques rely on relating compressed size of substrings to other combinatorial measures which are easier to manipulate.

# Parsing Methods

Preprocess S by generating a tree parsing using methods based on Deterministic Coin Tossing [Sahinalp, Vishkin 96, Muthukrishnan Sahinalp 00, Cormode Muthukrishnan 02].

Any substring induces a subtree of the parse tree:



# Parsing Methods for LCS

The number of unique nodes in the induced subtree (nodes representing substrings) approximates LZ compressed size of substring.

Approximate Least Compressible Substring by walking over tree, adding and removing nodes to represent sliding substring.

**Result:** Approximate LCS in time $O(|S| \log |S|)$ up to factor of $O(\log |S| \log^* |S|)$.

Naïve alg costs $O(|S| \lambda)$.

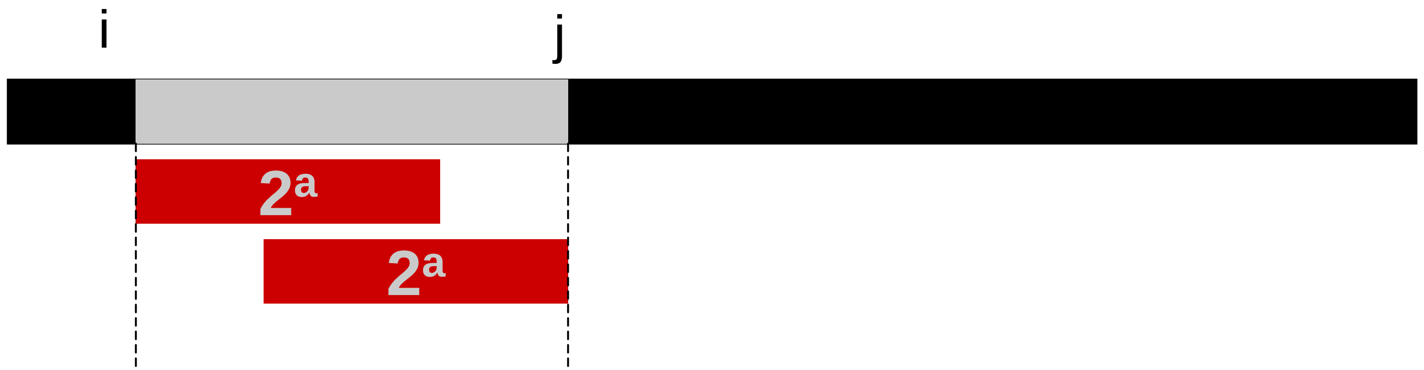
# Parsing Methods for SCSQ

Compute number of unique nodes for all substrings of length $2^a$. Represent any substring by two overlapping substrings of length $2^a$.

Compute estimate of SCSQ by summing number of distinct nodes (giving 2-factor approx).

**Result:** $O(|S| \log^2 |S|)$ preprocessing.

Approximate SCSQ to $O(\log |S| \log^* |S|)$ in time $O(1)$ per query



# Approximation of GLCS

From [Ergun, Muthukrishnan, Sahinalp 03], can show compressed size of concatenated substrings approximates "block edit distance" between them.

Bounding the change in block edit distance allows us to "skip over" substrings with similar compressed size, and only compute compression of small number of substrings.

**Result:** $O(1)$ approximation of GLCS in time $O(|S|\ \lambda\ /\ \log \lambda)$. Naïve alg costs $O(|S|\ \lambda)$.

# Open Problems

Consider other compression techniques:

- Prediction by Partial Matching (PPM) ?
- Grammar-based compression methods

Can Burrows-Wheeler transform be analyzed?

- Some results possible for eg BWT+RLE.
- Other combinations still unstudied
  eg. BWT+MTF (+HUFFMAN / +ARITHMETIC)

Stop abruptly.