On the bit-complexity of Lempel-Ziv compression

Rossano Venturini

joint work with Paolo Ferragina and Igor Nitto

Department of Computer Science
University of Pisa
Lempel-Ziv Compression

Lossless data compression by textual substitution (Lempel-Ziv '77):

INPUT string: ........ substring ............

Replaced by a pair <copy distance, length>

Many implementations: `gzip`, `arj`, `.gif`, `pkzip`, `compress`...
Lempel-Ziv 77

Parse input string from left to right and separate it into **phrases**: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: **(distance, length)** or **(0,c)** if single symbol

```
 a b a b a a a a
```
Parse input string from left to right and separate it into phrases: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: \textbf{(distance, length)} or \textbf{(0,c)} if single symbol
Parse input string from left to right and separate it into **phrases**: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: (**distance**, **length**) or (**0**,c) if single symbol

(a) (b) a b a a a

(0,a) (0,b)
Lempel-Ziv 77

Parse input string from left to right and separate it into **phrases**: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: **(distance, length)** or **(0,c)** if single symbol

(a) (b) (a b a) a a

(0,a) (0,b) (2,3)
Parse input string from left to right and separate it into **phrases**: either single symbols or repetitions..

Greedy strategy: it always selects the longest repetition

Backward-References: \((\text{distance}, \text{length})\) or \((0,c)\) if single symbol

\[(a) \ (b) \ (a \ b \ a) \ (a \ a)\]

\[(0,a) \ (0,b) \ (2,3) \ (1,2)\]
Parse input string from left to right and separate it into phrases: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: \((\text{distance, length})\) or \((0,c)\) if single symbol

\[
(a) \ (b) \ (a \ b \ a) \ (a)
\]

\[
(0,a) \ (0,b) \ (2,3) \ (1,2)
\]

Compress: \((f(0),a) \ (f(0),b) \ (f(2),g(3)) \ (f(1),g(2))\)
Lempel-Ziv 77

Parse input string from left to right and separate it into phrases: either single symbols or repetitions.

Greedy strategy: it always selects the longest repetition

Backward-References: \((\text{distance, length})\) or \((0,c)\) if single symbol

\[(a) \ (b) \ (a \ b \ a) \ (a) \ (0,a) \ (0,b) \ (2,3) \ (1,2)\]

Encode distances and lengths with two uniquely decodable encoding functions \(f,g:\mathbb{N} \rightarrow \{0,1\}^*\)

Compress: \((f(0),a) \ (f(0),b) \ (f(2),g(3)) \ (f(1),g(2))\)

Decompression is simple and efficient in practice
Given a string $T$ and fixed two functions $f, g$ to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on $T$. 

**Problem**

Given a string $T$ and fixed two functions $f, g$ to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on $T$. 

Minimize compress size
Given a string T and fixed two functions \( f, g \) to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on T.

Theoretical analysis usually assume that both \( f \) and \( g \) encode any value with a fixed number of bits (\( \log n \) bits where \( n = |T| \)).
Minimize compress size

**Problem**

Given a string $T$ and fixed two functions $f, g$ to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on $T$.

Theoretical analysis usually assume that both $f$ and $g$ encode any value with a fixed number of bits ($\log n$ bits where $n=|T|$)

Nothing to do! It is known that greedy strategy produces the minimum number of pairs. Thus, also the compress size is minimized (wrt $f$ and $g$).
Minimize compress size

Problem

Given a string $T$ and fixed two functions $f, g$ to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on $T$.

Theoretical analysis usually assume that both $f$ and $g$ encode any value with a fixed number of bits ($\log n$ bits where $n = |T|$)

Nothing to do! It is known that greedy strategy produces the minimum number of pairs. Thus, also the compress size is minimized (wrt $f$ and $g$).

Good in practice?
Minimize compress size

**Problem**

Given a string $T$ and fixed two functions $f, g$ to encode distances and lengths respectively, we want to minimize the compress size (in bits) produced by LZ77 on $T$.

Theoretical analysis usually assume that both $f$ and $g$ encode any value with a fixed number of bits ($\log n$ bits where $n = |T|$)

Nothing to do! It is known that **greedy strategy** produces the minimum number of pairs. Thus, also the compress size is minimized (wrt $f$ and $g$).

Good in practice?

<table>
<thead>
<tr>
<th>File</th>
<th>English</th>
<th>Sources</th>
<th>HTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bwt</td>
<td>20.6%</td>
<td>17.3%</td>
<td>3.8%</td>
</tr>
<tr>
<td>LZ-fix</td>
<td>26.1%</td>
<td>24.6%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

Files of 50 Mbytes
Theoretical analysis usually assume that both $f$ and $g$ encode any value with a fixed number of bits ($\log n$ bits where $n=|T|$).

Nothing to do! It is known that greedy strategy produces the minimum number of pairs. Thus, also the compress size is minimized (wrt $f$ and $g$).

Good in practice?

<table>
<thead>
<tr>
<th>File</th>
<th>Bwt</th>
<th>LZ-fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>20.6%</td>
<td>26.1%</td>
</tr>
<tr>
<td>Compression</td>
<td>24.6%</td>
<td>4.9%</td>
</tr>
<tr>
<td>Files of 50 Mbytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Very fast in decompression: 0.8 vs 20 secs
In practice (e.g., gzip) variable-length encoding functions are used. (e.g., Elias' Gamma and Delta, Fibonacci, etc.)
In practice (e.g., gzip) variable-length encoding functions are used. (e.g., Elias' Gamma and Delta, Fibonacci, etc.)

**Property (Increasing Cost):**

An encoding function $e: \mathbb{N} \rightarrow \{0,1\}^*$ satisfies the *increasing cost property* iff:

$$i < j \rightarrow |e(i)| \leq |e(j)|$$
In practice (e.g., gzip) variable-length encoding functions are used. (e.g., Elias' Gamma and Delta, Fibonacci, etc.)

**Property (Increasing Cost):**

An encoding function $e:[n] \rightarrow \{0,1\}^*$ satisfies the *increasing cost property* iff:

$$i < j \rightarrow |e(i)| \leq |e(j)|$$
Variable-length encoding functions

In practice (e.g., gzip) variable-length encoding functions are used. (e.g., Elias' Gamma and Delta, Fibonacci, etc.)

**Property (Increasing Cost):**

An encoding function $e : [n] \rightarrow \{0,1\}^*$ satisfies the *increasing cost property* iff:

$$i < j \rightarrow |e(i)| \leq |e(j)|$$

$Q(e,n)$: number of different encoding lengths of $e$ in $[n]$.

$Q(e,n) = O(\log n)$ for most of practical encoding functions
Greedy strategy wastes space

Assume any value j is encoded by f with O(log j) bits

T

... abacdde ............................ abae .... cdde ........ abacddf ...
Greedy strategy wastes space

Assume any value $j$ is encoded by $f$ with $O(\log j)$ bits

$T$

... $\text{abacdde}$ ...................... $\text{abaed}$ .... $\text{cdde}$ ....... $\text{abacddef}$ ...

$2^k$
Greedy strategy wastes space

Assume any value $j$ is encoded by $f$ with $O(\log j)$ bits

$T \quad \ldots \text{abacdde} \ldots \text{abae} \ldots \text{cdde} \ldots \text{abacddf} \ldots$

$2^k$

Cost $= |f(2^k)| = \log 2^k = k$ bits
Greedy strategy wastes space

Assume any value $j$ is encoded by $f$ with $O(\log j)$ bits

$\text{Cost} = |f(2^k)| = \log 2^k = k \text{ bits}$
Assume any value j is encoded by f with $O(\log j)$ bits

Cost $= |f(2^k)| = \log 2^k = k$ bits

Cost $\leq 2 \cdot |f(2^h)| = 2 \log 2^h = 2 \cdot h$ bits
Greedy strategy wastes space

Assume any value $j$ is encoded by $f$ with $O(\log j)$ bits.

![Diagram showing the comparison between Greedy and Non-greedy strategies]

Cost = $|f(2^k)| = \log 2^k = k$ bits

Cost $\leq 2 \cdot |f(2^h)| = 2 \log 2^h = 2 \cdot h$ bits

Non-greedy strategy is better if $h < k/2$
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

\[ S_k = 0^k 1 2^k 010^2 10^3 10^4 1 \ldots 0^k 1 \]
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

\[ S_k = \theta^k 1^2 \theta 1^2 0^2 1^3 0^4 1 \ldots \theta^k 1 \]

\[ LZ(S_k) = (\theta)(\theta^{k-1})(1^{2^k-1})(\theta 1)(\theta^2 1) \ldots (\theta^k 1) \]

Greedy strategy forces to copy from the beginning
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

\[ S_k = \theta^k 1 0^k 1 0^2 1 0^3 1 0^4 1 \ldots 0^k 1 \]

\[ LZ(S_k) = (\theta)(\theta^{k-1})(1^{2k-1})(\theta 1)(\theta 2 1) \ldots (\theta^k 1) \]

Amplifies distances!

Greedy strategy forces to copy from the beginning
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

\[ S_k = \theta^k 1 \theta 01 \theta 02 \theta 03 \theta 04 \ldots \theta^k 1 \]

\[ \text{LZ}(S_k) = (\theta) (\theta^{k-1}) (1^{2^k-1}) (\theta 1) (\theta 2 1) \ldots (\theta^k 1) \]

Assume any value \( j \) is encoded by \( f \) and \( g \) with \( O(\log j) \) bits.
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

\[ S_k = \theta^k 1^{2^k} \theta 01 \theta 02 \theta 10^3 \theta 10^4 \theta 1 \ldots \theta^k 1 \]

\[ \text{LZ}(S_k) = (\theta)(\theta^{k-1})(1^{2^{k-1}})(\theta 1)(\theta 2)(\theta 1) \ldots(\theta^k 1) \]

Assume any value \( j \) is encoded by \( f \) and \( g \) with \( O(\log j) \) bits

\[ |\text{LZ}(S_k)| > k |f(2^k)| = \Omega(k^2) \text{ \ bits} \]
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

Assume any value $j$ is encoded by $f$ and $g$ with $O(\log j)$ bits

$$S_k = 01021031041...0_k1$$

We can copy closer! The number of pairs is slightly increased but they require much less bits

$$|LZ(S_k)| > k|f(2^k)| = \Omega(k^2) \text{ bits}$$

$$r_{OPT}(S_k) = (\emptyset)(\emptyset^{k-1})(1^{2^k-1})(\emptyset)(1)(\emptyset)(\emptyset^11)...(\emptyset)(\emptyset^{k-1}1)$$
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

$$S_k = \theta^k 1^2 \theta 10^2 1 \theta 10^3 1 \theta 10^4 1 \ldots \theta^k 1$$

$$LZ(S_k) = (\theta)(\theta^{k-1})(1^{2^{k-1}})(\theta 1)(\theta 2 1) \ldots (\theta^k 1)$$

Assume any value $j$ is encoded by $f$ and $g$ with $O(\log j)$ bits

$$|LZ(S_k)| > k |f(2^k)| = \Omega(k^2) \text{ bits}$$

$$rOPT(S_k) = (\theta)(\theta^{k-1})(1^{2^{k-1}})(\theta)(1)(\theta)(\theta 1 1) \ldots (\theta)(\theta^{k-1} 1)$$

For $j \leq k$, $0^j 1 = (0)(0^{j-1} 1) = O(\log j) \text{ bits}$
Greedy strategy wastes space

There exists a family of strings for which greedy strategy wastes a lot of space!

$$S_k = \theta^k 1^2 \theta 10^2 1 \theta 10^3 1 \theta 10^4 1 \ldots \theta^k 1$$

$$LZ(S_k) = (\theta) (\theta^{k-1}) (1^{2^{k-1}}) (\theta 1) (\theta 2 1) \ldots (\theta^k 1)$$

Assume any value $j$ is encoded by $f$ and $g$ with $O(\log j)$ bits

For $j < k$, $0^j 1 = (0)(0^{j-1} 1) = O(\log j)$ bits
Earlier results are either:
- Space and Time inefficient (both $O(n^2)$) [Schuegraf et al., '74]
- Based on Heuristics [Klein '97, Cohn et al. '96, ...]

Our solution computes the optimal parsing in $O(n \log n)$ time for most of the encoding functions used in practice and $O(n)$ space
Reduction to shortest path on DAG

\[ T = \text{ababaaab} \]

G(T) \[ \begin{array}{ccccccccc}
\text{a} & \text{b} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{b} & \$ \\
\end{array} \]

A vertex for each string position \( \{v_1, \ldots, v_n\} \) where \( n = |T| + \text{extra } \text{eof} \ v_{n+1} \)
Reduction to shortest path on DAG

T = ababaaab

G(T) a → b → a → b → a → a → a → b →$

A vertex for each string position \{v_1, ..., v_n\} where n = |T| + extra eof v_{n+1}

An edge connecting v_i and v_j for any i < j ≤ n+1 if either:
- j = i+1
- T[i...j-1] is repeated in T[1..j-2]
Reduction to shortest path on DAG

\[ T = \text{ababaaab} \]

A vertex for each string position \( \{v_1, ..., v_n\} \) where \( n = |T| + \text{extra \( \text{eof} \) \( v_{n+1} \)} \)

An edge connecting \( v_i \) and \( v_j \) for any \( i < j \leq n+1 \) if either:

- \( j = i+1 \)
  
  or

- \( T[i...j-1] \) is repeated in \( T[1..j-2] \)
Reduction to shortest path on DAG

$T = ababaaab$

An edge connecting $v_i$ and $v_j$ for any $i < j < n+1$ if either:
- $j = i+1$
- $T[i...j-1]$ is repeated in $T[1..j-2]$

A vertex for each string position $\{v_1, \ldots, v_n\}$ where $n = |T| +$ extra eof $v_{n+1}$

Lempel-Ziv parsings of $T \iff$ paths in $G(T)$ from $v_1$ to $v_{n+1}$

Any edge $(v_i, v_j)$ has assigned the label $(d_{ij}, l_{ij})$ meaning that we can copy at distance $d_{ij}$ a repetition of length $l_{ij}$. 
Reduction to shortest path on DAG

T = ababaaab

G(T)

A vertex for each string position \{v_1, \ldots, v_{n+1}\}

An edge connecting \(v_i\) and \(v_j\) for any \(i < j < n+1\)

- \(j = i+1\)
- \(T[i...j-1]\) is repeated in \(T[1..j-2]\)

Any edge \((v_i, v_j)\) has also assigned the cost \(c(i,j) = |f(d_{ij})| + |g(l_{ij})|\).

Lempel-Ziv parsings of T \(\iff\) paths in G(T) from \(v_1\) to \(v_{n+1}\)

d-cost and l-cost

Different labels? Select the one that minimizes the cost!
Reduction to shortest path on DAG

T = ababaaab

G(T)

A vertex for each string position {v₁, …, vₙ}

An edge connecting vᵢ and vⱼ for any i < j < n + 1 if either:
- j = i + 1
- T[i...j-1] is repeated in T[1..j-2]

Lempel-Ziv parsings of T ⇔ paths in G(T) from v₁ to vₙ+1

Any edge (vᵢ, vⱼ) has also assigned the cost c(i, j) = |f(dᵢⱼ)| + |g(lᵢⱼ)|.

d-cost and l-cost

Different labels? Select the one that minimizes the cost!

Shortest paths identify optimal parsings!
Reduction to shortest path on DAG

\[ T = ababaaab \]

Any edge connecting \( v_i \) and \( v_j \) for any \( i < j < n+1 \) if either:

- \( j = i+1 \)
- \( T[i...j-1] \) is repeated in \( T[1..j-2] \)

Lempel-Ziv parsings of \( T \) \( \iff \) paths in \( G(T) \) from \( v_1 \) to \( v_{n+1} \)

A vertex for each string position \( \{v_1, \ldots, v_{n+1}\} \) where \( n = |T| \) + extra eof \( v_{n+1} \)

An edge connecting \( v_i \) and \( v_j \) has also assigned the cost \( c(i,j) = |f(d_{ij})| + |g(l_{ij})| \).

\( d \)-cost and \( l \)-cost

Different labels? Select the one that minimizes the cost!

Shortest paths identify optimal parsings!

Problem: How many edges?

\( O(n^2) \) in worst case! (e.g., \( T = a^n \))
Reduction to shortest path on DAG

\[ T = \text{ababaaaab} \]

A vertex for each string position \( \{v_1, \ldots, v_{n+1}\} \)

An edge connecting \( v_i \) and \( v_j \) for any \( i < j \) if either:
- \( j = i+1 \)
- \( T[i...j-1] \) is repeated in \( T[1..j-2] \)

\[ Lempel-Ziv \text{ parsings of } T \iff \text{paths in } G(T) \text{ from } v_1 \text{ to } v_{n+1} \]

Any edge \((v_i, v_j)\) has also assigned the cost \( c(i,j) = |f(d_{ij})| + |g(l_{ij})| \).

\textbf{Shortest paths identify optimal parsings!}

\textbf{Pruning!}

Problem: How many edges? \( O(n^2) \) in worst case! (e.g., \( T=a^n \))
Structural properties of G(T)

Property (Nesting)

If \((v_i, v_j) \in E_T\) then, for any \(i < t < j\):

1) \((v_i, v_t) \in E_T\) and \(|f(d')| \leq |f(d)|, |g(l')| \leq |g(l)|\), \(c(i,t) \leq c(i,j)\)

2) \((v_t, v_j) \in E_T\) and \(|f(d'')| \leq |f(d)|, |g(l'')| \leq |g(l)|\), \(c(t,i) \leq c(i,j)\)
Structural properties of G(T)

**Property (Nesting)**

If \((v_i, v_j) \in E_T\) then, for any \(i < t < j\):

1) \((v_i, v_t) \in E_T\) and \(|f(d')| \leq |f(d)|, |g(l')| \leq |g(l)|, c(i, t) \leq c(i, j)\)

2) \((v_t, v_j) \in E_T\) and \(|f(d'')| \leq |f(d)|, |g(l'')| \leq |g(l)|, c(t, i) \leq c(i, j)\)

**Proof of 1)**

\(T[i...j-1]\) repetition \(\rightarrow\) \(T[i...t-1]\) repetition.

(d',l') \quad (d,l)

\(v_i\) \quad \(v_t\) \quad \(v_j\)
Structural properties of G(T)

**Property (Nesting)**

If \((v_i, v_j) \in E_T\) then, for any \(i < t < j\):

1) \((v_i, v_t) \in E_T\) and \(|f(d')| \leq |f(d)|, |g(l')| \leq |g(l)|\), \(c(i, t) \leq c(i, j)\)

2) \((v_t, v_j) \in E_T\) and \(|f(d'')| \leq |f(d)|, |g(l'')| \leq |g(l)|\), \(c(t, i) \leq c(i, j)\)

**Proof of 1)**

\(T[i...j-1]\) repetition → \(T[i...t-1]\) repetition.

Each copy of \(T[i...j-1]\) contains a copy of \(T[i...t-1] \rightarrow d' \leq d\) and \(l' < l\).
Structural properties of $G(T)$

**Property (Nesting)**

If $(v_i, v_j) \in E_T$ then, for any $i < t < j$:

1) $(v_i, v_t) \in E_T$ and $|f(d')| \leq |f(d)|$, $|g(l')| \leq |g(l)|$, $c(i, t) \leq c(i, j)$

2) $(v_t, v_j) \in E_T$ and $|f(d'')| \leq |f(d)|$, $|g(l'')| \leq |g(l)|$, $c(t, i) \leq c(i, j)$

**Proof of 1)**

$T[i...j-1]$ repetition $\rightarrow$ $T[i...t-1]$ repetition.

Each copy of $T[i...j-1]$ contains a copy of $T[i...t-1] \rightarrow d' \leq d$ and $l' < l$.

Thus, $|f(d')| \leq |f(d)|$ and $|g(l')| \leq |g(l)| \rightarrow c(i, t) \leq c(i, j)$ by Increasing cost property of $f$ and $g$. 
Definition of maximal edge

An edge \((v_i, v_j)\) is said **maximal** iff either \(c(v_i, v_j) < c(v_i, v_{j+1})\) or \((v_i, v_{j+1}) \not\in E_T\)

or simply pick the longest one among equal cost edges outgoing from \(v_i\).
Definition of maximal edge

An edge \((v_i, v_j)\) is said **maximal** iff either \(c(v_i, v_j) < c(v_i, v_{j+1})\) or \((v_i, v_{j+1}) \notin E_T\).

or simply pick the longest one among equal cost edges outgoing from \(v_i\).

A maximal edge \((v_i, v_j)\) must be either

- **d-maximal**: \(|f(d_{ij})| < |f(d_{ij+1})|\)
- **l-maximal**: \(|g(l_{ij})| < |g(l_{ij+1})|\)
Definition

An edge \((v_i, v_j)\) is said **maximal** iff either \(c(v_i, v_j) < c(v_i, v_{j+1})\) or \((v_i, v_{j+1}) \notin E_T\).

Or simply pick the longest one among equal cost edges outgoing from \(v_i\).

A maximal edge \((v_i, v_j)\) must be either

- **d-maximal**: \(|f(d_{ij})| < |f(d_{i,j+1})|\)
- **l-maximal**: \(|g(l_{ij})| < |g(l_{i,j+1})|\)

Property

There are no more than \(Q(f,n) + Q(g,n)\) maximal edges outgoing from any vertex \(v_i\).
A pruning theorem

**Theorem**

There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges.
Theorem

There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges

Proof: (by contradiction)
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path $P$ having the longest initial subpath that traverses only maximal edges.
A pruning theorem

**Theorem**
There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges

**Proof: (by contradiction)**
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path $P$ having the longest initial subpath that traverses only maximal edges.

$e_v$ is the first non maximal edge
A pruning theorem

**Theorem**
There exists a Shortest Path from \(v_1\) to \(v_{n+1}\) in \(G(T)\) traversing only maximal edges.

**Proof: (by contradiction)**
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path \(P\) having the longest initial subpath that traverses only maximal edges.

\[ e_v \text{ is the first non maximal edge} \]

Consider the maximal edge \(m\), s.t. \(c(m) = c(e_v)\).
A pruning theorem

**Theorem**
There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges.

**Proof: (by contradiction)**
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path $P$ having the longest initial subpath that traverses only maximal edges.

Consider the maximal edge $m$, s.t. $c(m) = c(e_v)$.
We can replace $e_v$ with $m$ without increasing the cost.
Proof: (by contradiction)
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path $P$ having the longest initial subpath that traverses only maximal edges.

There exists $c(r,w) \leq c(e_u)$ (by Nesting Property)
We can replace $e_u$ with the new edge without increasing the cost
A pruning theorem

**Theorem**
There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges

**Proof: (by contradiction)**
Assume that every shortest path contains at least a non maximal edge. Among all the shortest paths, we pick a path $P$ having the longest initial subpath that traverses only maximal edges.

There exists $c(r, w) < c(e_u)$ (by Nesting Property)
We can replace $e_u$ with the new edge without increasing the cost

Contradiction! We created another shortest path having a longer initial subpath that traverses only maximal edges.

A pruning theorem

**Theorem**

There exists a Shortest Path from \( v_1 \) to \( v_{n+1} \) in \( G(T) \) traversing only maximal edges

**Pruned Subgraph** \( GP(T) \)

Subgraph of \( G(T) \) where non-maximal edges are removed

**Corollary**

Bit-Optimal Parsing \( \Leftrightarrow \) Shortest-Path computation in \( GP(T) \)
A pruning theorem

**Theorem**

There exists a Shortest Path from \( v_1 \) to \( v_{n+1} \) in \( G(T) \) traversing only maximal edges

**Pruned Subgraph** \( GP(T) \)

Subgraph of \( G(T) \) where non-maximal edges are removed

**Corollary**

Bit-Optimal Parsing \( \iff \) Shortest-Path computation in \( GP(T) \)

**Property**

There are no more than \( Q(f,n) + Q(g,n) \) maximal edges outgoing from any vertex \( v_i \)
### Theorem
There exists a Shortest Path from $v_1$ to $v_{n+1}$ in $G(T)$ traversing only maximal edges.

### Pruned Subgraph $GP(T)$
Subgraph of $G(T)$ where non-maximal edges are removed.

### Corollary
Bit-Optimal Parsing $\iff$ Shortest-Path computation in $GP(T)$

### Property
There are no more than $Q(f,n)+Q(g,n)$ maximal edges outgoing from any vertex $v_i$.

Given $GP(T)$, $O(n \cdot (Q(f,n)+Q(g,n))$ time!
We consider the problem of generating d-maximal edges.  

l-maximal edges can be easily computed in $O(n \cdot Q(n,g))$. See paper for more details.

We restrict our attention to d-maximal edges of cost $c$. The procedure can be repeated to find the d-maximal edges for all the $Q(f,n)$ different costs.
We consider the problem of generating d-maximal edges of cost $c$. 

Practical solution!
We consider the problem of generating d-maximal edges of cost $c$.

We need:
- suffix array of $T$;
- to compute $lcp$ among any two suffixes in $O(1)$ time.

Preprocess: $O(n)$ time  
Occupancy: $O(n)$ words

Practical solution!
We consider the problem of generating d-maximal edges of cost $c$.

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$. 

Practical solution!
We consider the problem of generating d-maximal edges of cost $c$

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$
We consider the problem of generating d-maximal edges of cost $c$.

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$.

We are looking for position $p \in W_c(i)$ such that $T[p, ...]$ shares the longest common prefix with $T[i, ...]$ among the other suffixes $T[j, ...]$ with $j \in W_c(i)$.
Computing GP(T)

We consider the problem of generating d-maximal edges of cost $c$

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$

We are looking for position $p \in W_c(i)$ such that $T[p, ...]$ shares the longest common prefix with $T[i, ...]$ among the other suffixes $T[j, ...]$ with $j \in W_c(i)$.

**Fact**

If there exists a maximal edge having d-cost $c$ outgoing from $v_i$, then it must be $(v_i, v_{i+t})$ where $t = \text{lcp}(i, p)$
We consider the problem of generating d-maximal edges of cost $c$

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$

We are looking for position $p \in W_c(i)$ such that $T[p,...]$ shares the longest common prefix with $T[i,...]$ among the other suffixes $T[j,...]$ with $j \in W_c(i)$.

Keep positions $W_c(i)$ in a Balanced Binary Tree $BST_c(i)$ sorted by lexicographic rank (known from Suffix Array of the text)
Computing $\text{GP}(T)$

We consider the problem of generating $d$-maximal edges of cost $c$.

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$.

We are looking for position $p \in W_c(i)$ such that $T[p, \ldots]$ shares the longest common prefix with $T[i, \ldots]$ among the other suffixes $T[j, \ldots]$ with $j \in W_c(i)$.

- Keep positions $W_c(i)$ in a Balanced Binary Tree $\text{BST}_c(i)$ sorted by lexicographic rank (known from Suffix Array of the text).
- Query $\text{BST}_c(i)$ and find pred/succ in lexicographic order of $T[i, \ldots]$ among $W_c(i)$ and return the one that maximizes lcp with $T[i, \ldots]$. $O(\log n)$ time.

Practical solution!
We consider the problem of generating $d$-maximal edges of cost $c$

$I_c = [a, b]$ represents the interval of integers that can be represented by

We are looking for position $p \in W_c(i)$ such that $T[p, ..]$ shares the
longest common prefix with $T[i, ..]$ among the other suffixes $T[j, ..]$ with
$j \in W_c(i)$.

- Query $\text{BST}_c(i)$ (pred/succ) $O(\log n)$ time
- Two lcp computations $O(1)$ time
- Build $\text{BST}_c(i+1)$ from $\text{BST}_c(i)$ $O(\log n)$ time

Compute all the maximal edges of cost $c$ in $O(n \log n)$:

- Query $\text{BST}_c(i)$ (pred/succ) $O(\log n)$ time
- Two lcp computations $O(1)$ time
- Build $\text{BST}_c(i+1)$ from $\text{BST}_c(i)$ $O(\log n)$ time

Practical solution!

Keep positions $W_c(i)$ in a Balanced
Binary Tree $\text{BST}_c(i)$ sorted by
lexicographic rank (known from
Suffix Array of the text)

Query $\text{BST}_c(i)$ and find pred/succ in
lexicographic order of $T[i, ..]$ among $W_c(i)$
and return the one that maximizes lcp
with $T[i, ..]$. $O(\log n)$ time.
Computing GP(T)

We consider the problem of generating d-maximal edges of cost \( c \) in a text \( T \).

**I** \( = [a, b] \): interval of integers that can be represented by \( f \) with cost \( c \).

We are looking for position \( p \in W_c(i) \) such that \( T[p, \ldots] \) shares the longest common prefix with \( T[i, \ldots] \) among the other suffixes \( T[j, \ldots] \) with \( j \in W_c(i) \).

Keep positions \( W_c(i) \) in a Balanced Binary Tree BST \( c(i) \) sorted by lexicographic rank (known from Suffix Array of the text).

Compute all the maximal edges of cost \( c \) in \( O(n \log n) \):

- Query BST \( c(i) \) (pred/succ) \( O(\log n) \) time
- Two lcp computations \( O(1) \) time
- Build BST \( c(i+1) \) from BST \( c(i) \) \( O(\log n) \) time

Query BST \( c(i) \) and find pred/succ in lexicographic order of \( T[i, \ldots] \) among \( W_c(i) \) and return the one that maximizes lcp with \( T[i, \ldots] \). \( O(\log n) \) time.

Repeated in “parallel” for any possible cost \( c \).

Computation of GP(T) requires \( O(n Q(f,n) \log n + n Q(g,n)) \) time and \( O(n) \) space.

\( O(n \log^2 n) \) for most of functions.

Practical solution!

\( T \)

\( i - b \) \( p \) \( i - a \) \( \ldots \) \( i \)
We consider the problem of generating d-maximal edges

\[ I_c = [a, b] : \text{interval of integers that can be represented by } f \text{ with cost } c \]

“Batched” generation process:

We conceptually partition the text in blocks of size \(|I_c|\).
We compute maximal edges for positions in any block \(A\) in \(O(|A|)\) (i.e., \(O(1)\) amortized time) observing that:
More efficient computation of GP(T)

We consider the problem of generating d-maximal edges

$I_c = [a, b]$: interval of integers that can be represented by $f$ with cost $c$

“Batched” generation process:

We conceptually partition the text in block of size $|I_c|$. We compute maximal edges for positions in any block $A$ in $O(|A|)$ (i.e., $O(1)$ ammortized time) observing that:
We consider the problem of generating d-maximal edges

\[ I_c = [a, b] : \text{interval of integers that can be represented by } f \text{ with cost } c \]

"Batched" generation process:

We conceptually partition the text in block of size \(|I_c|\).
We compute maximal edges for positions in any block \(A\) in \(O(|A|)\) (i.e., \(O(1)\) ammortized time) observing that:
More efficient computation of \( \text{GP}(T) \)

We consider the problem of generating \( \text{d-maximal edges} \)

\[ I_c = [a, b]: \text{interval of integers that can be represented by } \mathcal{f} \text{ with cost } c \]

“Batched” generation process:

We conceptually partition the text in block of size \( |I_c| \).
We compute maximal edges for positions in any block \( A \) in \( O(|A|) \) (i.e., \( O(1) \) ammortized time) observing that:

- \( |A \cup W_c(l) \cup W_c(r)| = O(|I_c|) \)
More efficient computation of \( GP(T) \)

We consider the problem of generating d-maximal edges

\[ I_c = [a, b] : \text{interval of integers that can be represented by } f \text{ with cost } c \]

“Batched” generation process:

We conceptually partition the text in block of size \( |I_c| \).
We compute maximal edges for positions in any block \( A \) in \( O(|A|) \) (i.e., \( O(1) \) ammortized time) observing that:

- \( |A \cup W_c(l) \cup W_c(r)| = O(|I_c|) \)
- Build and perform (suitable) a visit of the compact-trie of all suffixes in \( A \cup W_c(l) \cup W_c(r) \) in \( O(|I_c|) \) time.

\[ W_c(l) \cup W_c(r) \]

\[ T \]

\[ l - b \quad l - a \quad r - a \quad \ell \quad r \]
More efficient computation of \( GP(T) \)

We consider the problem of generating d-maximal edges

\[ I_c = [a, b] : \text{interval of integers that can be represented by } f \text{ with cost } c \]

"Batched" generation process:

We conceptually partition the text in block of size \( |I_c| \).
We compute maximal edges for positions in any block \( A \) in \( O(|A|) \) (i.e., \( O(1) \) ammortized time) observing that:

- \( |A \cup W_c(l) \cup W_c(r)| = O(|I_c|) \)
- Build and perform (suitable) a visit of the compact-trie of all suffixes in \( A \cup W_c(l) \cup W_c(r) \) in \( O(|I_c|) \) time.

See paper for more details
In summary

Given:
A string $T[1..n]$ whose symbols belong to an alphabet of size $O(n^c)$
A pair of encoding functions $f, g$ satisfying the increasing-cost property

The $(f, g)$ bit-optimal parsing of $T$ can be computed in:

- $O(n)$ space  and  $O(n \times (Q(f,n)+Q(g,n)))$ time
  (vs. $O(n^2)$ previous solutions)

where $Q(e,n) = \# \text{ distinct equal-cost class for } e \text{ in } [n]$

... $O(n \log n)$ time for most practical choices of $f$ and $g$

For generic alphabet of size $\sigma$, we need additional $T_{\text{sort}}(n, \sigma)$ time
<table>
<thead>
<tr>
<th>File</th>
<th>English</th>
<th>Sources</th>
<th>HTML</th>
<th>Dec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>37.5%</td>
<td>23.2%</td>
<td>20.0%</td>
<td>0.7 s</td>
</tr>
<tr>
<td>bzip2</td>
<td>28.4%</td>
<td>19.7%</td>
<td>10.6%</td>
<td>6.3 s</td>
</tr>
<tr>
<td>Bwt</td>
<td>20.6%</td>
<td>17.3%</td>
<td>3.8%</td>
<td>20.2 s</td>
</tr>
<tr>
<td>LZ-fix</td>
<td>26.1%</td>
<td>24.6%</td>
<td>4.9%</td>
<td>0.8 s</td>
</tr>
<tr>
<td>LZ-OPT</td>
<td>21.6%</td>
<td>17.6%</td>
<td>3.8%</td>
<td>0.9 s</td>
</tr>
</tbody>
</table>

Files of 50 Mbytes
### Experiments

<table>
<thead>
<tr>
<th>File</th>
<th>English</th>
<th>Sources</th>
<th>HTML</th>
<th>Dec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>37.5%</td>
<td>23.2%</td>
<td>20.0%</td>
<td>0.7 s</td>
</tr>
<tr>
<td>bzip2</td>
<td>28.4%</td>
<td>19.7%</td>
<td>10.6%</td>
<td>6.3 s</td>
</tr>
<tr>
<td>Bwt</td>
<td>20.6%</td>
<td>17.3%</td>
<td>3.8%</td>
<td>20.2 s</td>
</tr>
<tr>
<td>LZ-fix</td>
<td>26.1%</td>
<td>24.6%</td>
<td>4.9%</td>
<td>0.8 s</td>
</tr>
<tr>
<td>LZ-OPT</td>
<td>21.6%</td>
<td>17.6%</td>
<td>3.8%</td>
<td>0.9 s</td>
</tr>
</tbody>
</table>

Files of 50 Mbytes

- **BWT compression ratio**
- **Gzip decompression speed!**
Thank you!
Optimize Greedy Strategy: the right-most references

Given a string $T$, there exist several algorithms to compute the LZ77 parsing (with greedy strategy) of $T$ in $O(|T|)$ time and space.

All of them use the Suffix Array of $T$ and LCP array.
Optimize Greedy Strategy: the right-most references

Given $f$ and $g$ satisfying the *Increasing Cost Property* and assuming to use the greedy strategy, how can we minimize the $|\text{compress}|$?

Essentially, for any phrase $w_i$, what is the pair $(d_i, l_i)$ having the smallest encoding?
Optimize Greedy Strategy: the right-most references

Given $f$ and $g$ satisfying the *Increasing Cost Property* and assuming to use the greedy strategy, how can we minimize the $|\text{compress}|$?

Essentially, for any phrase $w_i$, what is the pair $(d_i, l_i)$ having the smallest encoding?

$l_i = |w_i|$ is fixed by greedy strategy
Given $f$ and $g$ satisfying the Increasing Cost Property and assuming to use the greedy strategy, how can we minimize the $|\text{compress}|$?

Essentially, for any phrase $w_i$, what is the pair $(d_i, l_i)$ having the smallest encoding?

Minimize the encoding of $d_i$.

There may be multiple copies of $w_i$ in the prefix of $T$
Given $f$ and $g$ satisfying the *Increasing Cost Property* and assuming to use the greedy strategy, how can we minimize the $|\text{compress}|$? 

Essentially, for any phrase $w_i$, what is the pair $(d_i, l_i)$ having the smallest encoding? 

Minimize the encoding of $d_i$. 

There may be multiple copies of $w_i$ in the prefix of $T$.
Given \( f \) and \( g \) satisfying the *Increasing Cost Property* and assuming to use the greedy strategy, how can we minimize the \(|\text{compress}|\)?

Essentially, for any phrase \( w_i \), what is the pair \((d_i,l_i)\) having the smallest encoding?

Minimize the encoding of \( d_i \).

There may be multiple copies of \( w_i \) in the prefix of \( T \)

Select the right-most copy (i.e., the one that is closer to \( p_i \) and starts before it)

\[ T \]

\[ w_1 \quad w_2 \quad w_3 \quad \ldots \quad w_i \quad \ldots \quad w_j \]

\( d_i \) must be as small as possible
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

SA

Range of positions in SA of all suffixes prefixed by $w_i$

Left-most copy of $w_i$ reduces to

$\text{RMQ} =$ find the smallest position in the range
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copy computation seems to be harder.

Left-most copy of $w_i$ reduces to

RMQ = find the smallest position in the range

$O(n)$ time to preprocess SA to solve $O(n)$ RMQs in $O(n)$ time
Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Right-most copy of $w_i$ reduces to

Y-Max Query = find the largest position which is smaller than $p_i$
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Right-most copy of \( w_i \) reduces to

\[ \text{Y-Max Query} = \text{find the largest position which is smaller than } p_i \]

\( O(n \log n) \) time to preprocess SA to solve

\( O(n) \) queries in \( o(n \log n) \) time.
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Our solution: Right-most copies can be computed in $O(n(1+\log \sigma / \log \log n))$ time where $\sigma$ is the alphabet size.

$w_1$ ...

$w_i$ ...

$w_j$ ...

Range of positions in SA of all suffixes prefixed by $w_i$
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Our solution: Right-most copies can be computed in $O(n (1 + \log \sigma / \log \log n)$ time where $\sigma$ is the alphabet size.

$O(n)$ for $\sigma = \text{polylog}(n)$
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Our solution: Right-most copies can be computed in $O(n \log \sigma / \log \log n)$ time where $\sigma$ is the alphabet size. We resort to:

- Suffix tree whose nodes are collected in groups of $O(\sigma)$ nodes each.

$\sigma = \text{polylog}(n)$ for

Range of positions in SA of all suffixes prefixed by $w_i$
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Our solution: Right-most copies can be computed in $O(n \cdot (1 + \log \sigma / \log \log n)$ time where $\sigma$ is the alphabet size. We resort to:

- Suffix tree whose nodes are collected in groups of $O(\sigma)$ nodes each.
- Global algorithm plus dynamic data structures to solve RMQ in each group

$O(n)$ for $\sigma = \text{polylog}(n)$
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

Our solution: Right-most copies can be computed in $O(n \cdot (1 + \log \sigma / \log \log n)$ time where $\sigma$ is the alphabet size. We resort to:

1. Suffix tree whose nodes are collected in groups of $O(\sigma)$ nodes each.
2. Global algorithm plus dynamic data structures to solve RMQ in each group.

See paper for more details.
How to compute right-most copy

Previous known algorithms are able to compute only left-most copies. Right-most copies computation seems to be harder.

In practice, it compresses better, but still worse than BWT
## Experiments

<table>
<thead>
<tr>
<th>File</th>
<th>English</th>
<th>Sources</th>
<th>HTML</th>
<th>Dec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>37.5%</td>
<td>23.2%</td>
<td>20.0%</td>
<td>0.7 s</td>
</tr>
<tr>
<td>bzip2</td>
<td>28.4%</td>
<td>19.7%</td>
<td>10.6%</td>
<td>6.3 s</td>
</tr>
<tr>
<td>Bwt</td>
<td>20.6%</td>
<td>17.3%</td>
<td>3.8%</td>
<td>20.2 s</td>
</tr>
<tr>
<td>LZ-fix</td>
<td>26.1%</td>
<td>24.6%</td>
<td>4.9%</td>
<td>0.8 s</td>
</tr>
<tr>
<td>LZ-RM</td>
<td>23.8%</td>
<td>20.2%</td>
<td>4.3%</td>
<td>0.9 s</td>
</tr>
<tr>
<td>LZ-OPT</td>
<td>21.6%</td>
<td>17.6%</td>
<td>3.8%</td>
<td>0.9 s</td>
</tr>
</tbody>
</table>

Files of 50 Mbytes