String indexing in the Word RAM model, part 3

Paweł Gawrychowski

University of Wrocław

Lempel-Ziv based compression methods

Text $t[1..N]$ is partitioned into disjoint blocks $b_1 b_2 \dots b_n$. Each block is defined in terms of the blocks on its left.

What we exactly mean by "defined" depends on the exact version. The most common are the following two:

LZ77, LZ: the next block b_i is a subword of the already processed prefix concatenated with exactly one new character, `zip,gzip,PNG`

LZ78, LZW: the next block b_i is a block on the left concatenated with exactly one new character. `compress,GIF,TIFF,PDF`

Lempel-Ziv based compression methods

Text $t[1..N]$ is partitioned into disjoint blocks $b_1 b_2 \dots b_n$. Each block is defined in terms of the blocks on its left.

What we exactly mean by "defined" depends on the exact version. The most common are the following two:

LZ77, LZ the next block b_i is a subword of the already processed prefix concatenated with exactly one new character, `zip,gzip,PNG`

LZ78, LZW the next block b_i is a block on the left concatenated with exactly one new character. `compress,GIF,TIFF,PDF`

Lempel-Ziv based compression methods

Text $t[1..N]$ is partitioned into disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on its left.

What we exactly mean by "defined" depends on the exact version. The most common are the following two:

- **LZ77, LZ** the next block b_i is a subword of the already processed prefix concatenated with exactly one new character, `zip,gzip,PNG`
- **LZ78, LZW** the next block b_i is a block on the left concatenated with exactly one new character. `compress,GIF,TIFF,PDF`

Lempel-Ziv based compression methods

Text $t[1..N]$ is partitioned into disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on its left.

What we exactly mean by “defined” depends on the exact version. The most common are the following two:

LZ77, LZ the next block b_i is a subword of the already processed prefix concatenated with exactly one new character, `zip,gzip,PNG`

LZ78, LZW the next block b_i is a block on the left concatenated with exactly one new character. `compress,GIF,TIFF,PDF`

An example of LZW compression:

$$\boxed{\mathrm{ababbababababababababaabbbaa}}$$

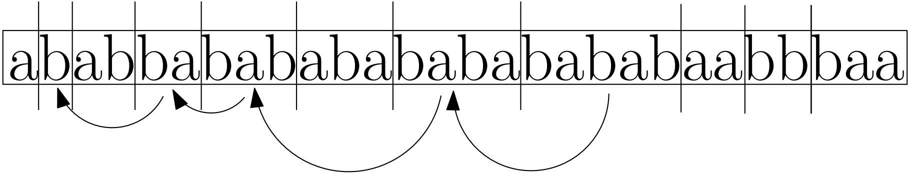
Even though $n \in \Omega(\sqrt{N})$, the compression/decompression are fast and simple, so the method is useful.

An example of LZW compression:

a|b|ab|ba|bab|aba|baba|babab|aa|bb|baa

Even though $n \in \Omega(\sqrt{N})$, the compression/decompression are fast and simple, so the method is useful.

An example of LZW compression:



Even though $n \in \Omega(\sqrt{N})$, the compression/decompression are fast and simple, so the method is useful.

An example of LZ compression:

ababbababaaabbababaabaabbbaa

It is easy to construct an example, where $n = \mathcal{O}(\log N)$.

Well, most probably such example will not occur in practice, but anyway such good compression method is achieved for the Fibonacci words, which are often used as a "benchmark" for text algorithms.

There is also the self-referential variant, where the new block can refer to itself.

An example of LZ compression:

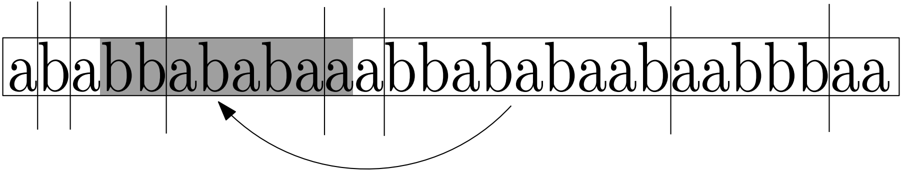
a	b	abb	ababa	aa	bbababaab	aabbb	aa

It is easy to construct an example, where $n = \mathcal{O}(\log N)$.

Well, most probably such example will not occur in practice, but anyway such good compression method is achieved for the Fibonacci words, which are often used as a “benchmark” for text algorithms.

There is also the self-referential variant, where the new block can refer to itself.

An example of LZ compression:

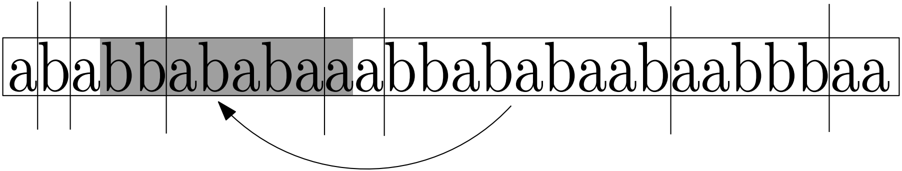


It is easy to construct an example, where $n = \mathcal{O}(\log N)$.

Well, most probably such example will not occur in practice, but anyway such good compression method is achieved for the Fibonacci words, which are often used as a “benchmark” for text algorithms.

There is also the self-referential variant, where the new block can refer to itself.

An example of LZ compression:

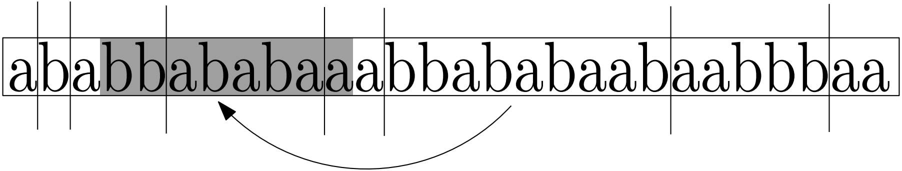


It is easy to construct an example, where $n = \mathcal{O}(\log N)$.

Well, most probably such example will not occur in practice, but anyway such good compression method is achieved for the Fibonacci words, which are often used as a “benchmark” for text algorithms.

There is also the self-referential variant, where the new block can refer to itself.

An example of LZ compression:



It is easy to construct an example, where $n = \mathcal{O}(\log N)$.

Well, most probably such example will not occur in practice, but anyway such good compression method is achieved for the Fibonacci words, which are often used as a “benchmark” for text algorithms.

There is also the self-referential variant, where the new block can refer to itself.

The blocks are described by pairs (in LZW) or triples (in LZ):

...ababbababaaabbababaabaabbbaaa...

$\downarrow$

...,a,b,(1,2,b),(1,4,a),(1,1,a),(4,8,b),(11,4,b),(10,2,a),...

The blocks are described by pairs (in LZW) or triples (in LZ):

$$\ldots,\mathrm{a},\mathrm{b},(1,2,\mathrm{b}),(1,4,\mathrm{a}),(1,1,\mathrm{a}),(4,8,\mathrm{b}),(11,4,\mathrm{b}),(10,2,\mathrm{a}),\ldots$$

$$p = \mathrm{aaab}$$

The blocks are described by pairs (in LZW) or triples (in LZ):

$$\ldots,\mathrm{a},\mathrm{b},(1,2,\mathrm{b}),(1,4,\mathrm{a}),(1,1,\mathrm{a}),(4,8,\mathrm{b}),(11,4,\mathrm{b}),(10,2,\mathrm{a}),\ldots$$

p=aaab

Motivation

We want to store repetitive texts (say, genomic databases) in compressed form, but such that we can search them quickly.

In other words, given a text, build a small structure which allows fast pattern matching.

Pattern matching?

Given $p[1..m]$ we want to find where it occurs **exactly** in text $t[1..n]$. We might want the first occurrence, or all of them, or just a few...

Such structure is called an index. If it also allows retrieving the original text, it is called a self-index.

Motivation

We want to store repetitive texts (say, genomic databases) in compressed form, but such that we can search them quickly.

In other words, given a text, build a small structure which allows fast pattern matching.

Pattern matching?

Given $p[1..m]$ we want to find where it occurs **exactly** in text $t[1..n]$. We might want the first occurrence, or all of them, or just a few...

Such structure is called an index. If it also allows retrieving the original text, it is called a self-index.

Motivation

We want to store repetitive texts (say, genomic databases) in compressed form, but such that we can search them quickly.

In other words, given a text, build a small structure which allows fast pattern matching.

Pattern matching?

Given $p[1..m]$ we want to find where it occurs **exactly** in text $t[1..n]$.
We might want the first occurrence, or all of them, or just a few...

Such structure is called an index. If it also allows retrieving the original text, it is called a self-index.

Motivation

We want to store repetitive texts (say, genomic databases) in compressed form, but such that we can search them quickly.

In other words, given a text, build a small structure which allows fast pattern matching.

Pattern matching?

Given $p[1..m]$ we want to find where it occurs **exactly** in text $t[1..n]$.
We might want the first occurrence, or all of them, or just a few...

Such structure is called an index. If it also allows retrieving the original text, it is called a self-index.

Motivation

We want to store repetitive texts (say, genomic databases) in compressed form, but such that we can search them quickly.

In other words, given a text, build a small structure which allows fast pattern matching.

Pattern matching?

Given $p[1..m]$ we want to find where it occurs **exactly** in text $t[1..n]$. We might want the first occurrence, or all of them, or just a few...

Such structure is called an index. If it also allows retrieving the original text, it is called a self-index.

Problem, more precisely

We are asked to build a self-index for a string $t[1..n]$ whose LZ77 parse consists of z phrases.

Why LZ77?

The number of those phrases is believed to be the **right** measure of how repetitive the text is.

We want to use space proportional to z, not n.

Problem, more precisely

We are asked to build a self-index for a string $t[1..n]$ whose LZ77 parse consists of z phrases.

Why LZ77?

The number of those phrases is believed to be the **right** measure of how repetitive the text is.

We want to use space proportional to z, not n.

Problem, more precisely

We are asked to build a self-index for a string $t[1..n]$ whose LZ77 parse consists of z phrases.

Why LZ77?

The number of those phrases is believed to be the **right** measure of how repetitive the text is.

We want to use space proportional to z, not n.

Solution?

Straight-line program, or grammar representation

Simply a context-free grammar with **exactly** one production per nonterminal.

Rytter 2003, Charikar et al. 2005

A LZ77 parse consisting of z phrases can be converted to a grammar consisting of $g = \mathcal{O}(z \log n)$ words. The grammar is AVL-balanced (Rytter) or weight-balanced (Charikar et al.).

Weight-balanced means that for each production $A \rightarrow BC$ we have that $|B| \approx |C|$.

Extracting an arbitrary substring of length ℓ from a balanced SLP takes $\mathcal{O}(\log n + \ell)$ time by just traversing.

Solution?

Straight-line program, or grammar representation

Simply a context-free grammar with **exactly** one production per nonterminal.

Rytter 2003, Charikar et al. 2005

A LZ77 parse consisting of z phrases can be converted to a grammar consisting of $g = \mathcal{O}(z \log n)$ words. The grammar is AVL-balanced (Rytter) or weight-balanced (Charikar et al.).

Weight-balanced means that for each production $A \rightarrow BC$ we have that $|B| \approx |C|$.

Extracting an arbitrary substring of length ℓ from a balanced SLP takes $\mathcal{O}(\log n + \ell)$ time by just traversing.

Solution?

Straight-line program, or grammar representation

Simply a context-free grammar with **exactly** one production per nonterminal.

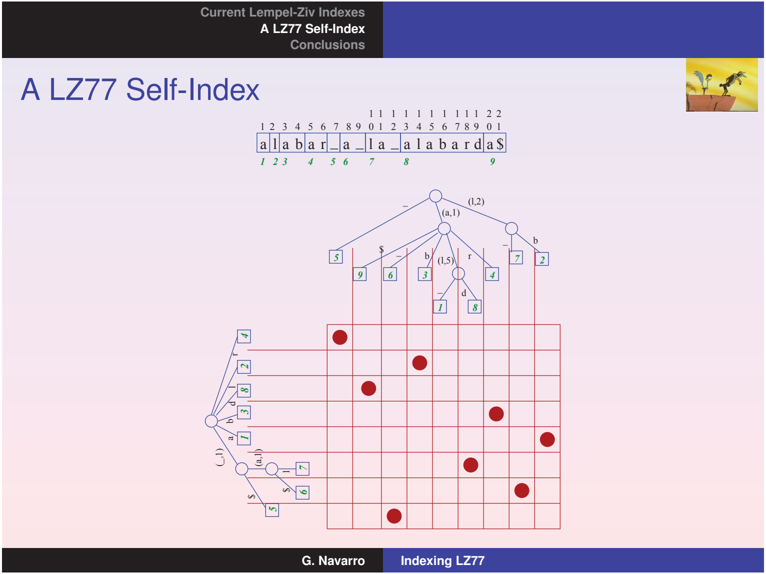
Rytter 2003, Charikar et al. 2005

A LZ77 parse consisting of z phrases can be converted to a grammar consisting of $g = \mathcal{O}(z \log n)$ words. The grammar is AVL-balanced (Rytter) or weight-balanced (Charikar et al.).

Weight-balanced means that for each production $A \to BC$ we have that $|B| \approx |C|$.

Extracting an arbitrary substring of length ℓ from a balanced SLP takes $\mathcal{O}(\log n + \ell)$ time by just traversing.

Framework (of Navarro)



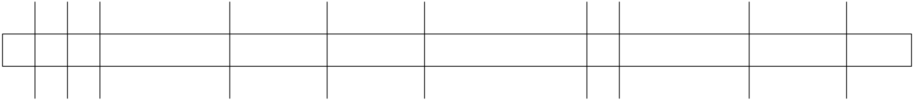
Idea

Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

Idea

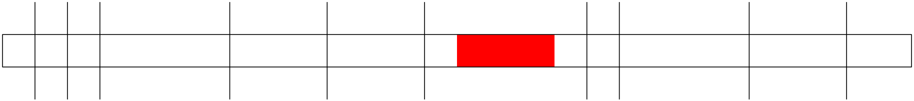


Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

Idea



Secondary occurrence

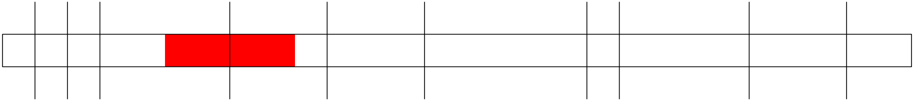
An occurrence is secondary iff it is completely contained in some phrase.

Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

Idea



Primary occurrence

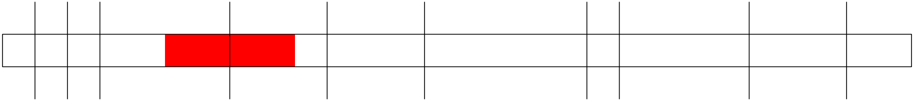
An occurrence is primary iff it crosses some boundary.

Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

Idea



Primary occurrence

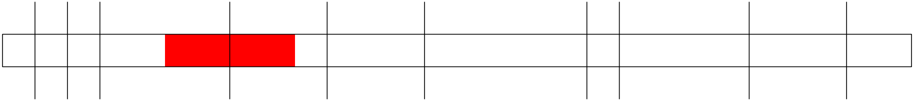
An occurrence is primary iff it crosses some boundary.

Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

Idea



Primary occurrence

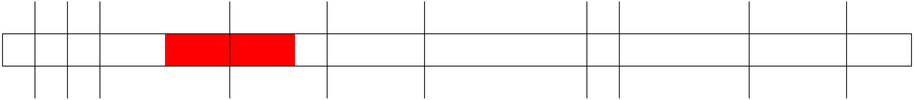
An occurrence is primary iff it crosses some boundary.

Observation (by Kärkkäinen and Ukkonen?)

If the pattern occurs in the text, there is at least one primary occurrence.

Assuming we have all primary occurrences, all secondary occurrences can be found via 2-sided 2D range reporting.

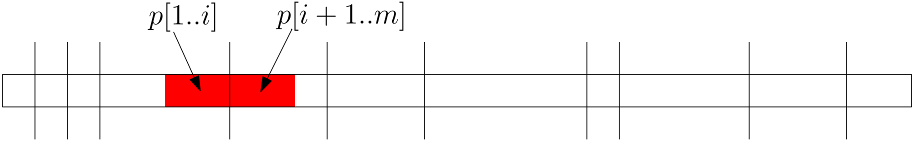
Idea, continued



To find all primary occurrences of $p[1..m]$, for each $1 \leq i \leq m$, we

1. search for $p[i+1..m]$ in the compacted trie of the suffixes starting at phrase boundaries,
2. search for $(p[1..i])^R$ in the compacted trie of the reversed phrases,
3. check the results via random access,
4. use range reporting to find all boundaries preceded by $p[1..i]$ and followed by $p[i+1..m]$.

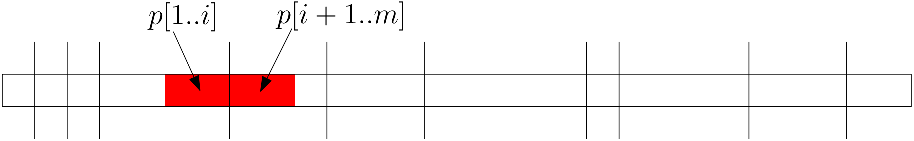
Idea, continued



To find all primary occurrences of $p[1..m]$, for each $1 \leq i \leq m$, we

1. search for $p[i+1..m]$ in the compacted trie of the suffixes starting at phrase boundaries,
2. search for $(p[1..i])^R$ in the compacted trie of the reversed phrases,
3. check the results via random access,
4. use range reporting to find all boundaries preceded by $p[1..i]$ and followed by $p[i+1..m]$.

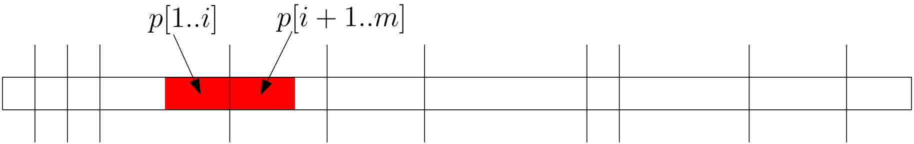
Idea, continued



To find all primary occurrences of $p[1..m]$, for each $1 \leq i \leq m$, we

1. search for $p[i+1..m]$ in the compacted trie of the suffixes starting at phrase boundaries,
2. search for $(p[1..i])^R$ in the compacted trie of the reversed phrases,
3. check the results via random access,
4. use range reporting to find all boundaries preceded by $p[1..i]$ and followed by $p[i+1..m]$.

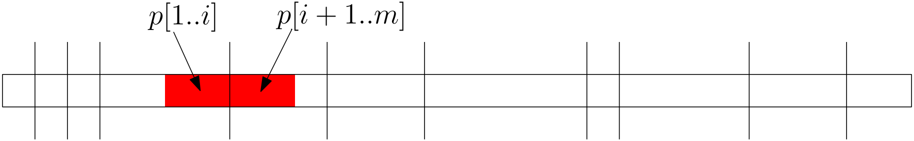
Idea, continued



To find all primary occurrences of $p[1..m]$, for each $1 \leq i \leq m$, we

1. search for $p[i+1..m]$ in the compacted trie of the suffixes starting at phrase boundaries,
2. search for $(p[1..i])^R$ in the compacted trie of the reversed phrases,
3. check the results via random access,
4. use range reporting to find all boundaries preceded by $p[1..i]$ and followed by $p[i+1..m]$.

Idea, continued



To find all primary occurrences of $p[1..m]$, for each $1 \leq i \leq m$, we

1. search for $p[i+1..m]$ in the compacted trie of the suffixes starting at phrase boundaries,
2. search for $(p[1..i])^R$ in the compacted trie of the reversed phrases,
3. check the results via random access,
4. use range reporting to find all boundaries preceded by $p[1..i]$ and followed by $p[i+1..m]$.

Bookmarking

Because we know that we will extract characters from the phrase boundaries, we can replace $\mathcal{O}(\log n + \ell)$ with the following bound:

Lemma

Given a balanced SLP for S with g rules and integers b and L, we can store $2\log g + \mathcal{O}(\log L)$ bits such that later, given $\ell \leq L$, we can extract $t[b-\ell..b+\ell]$ in $\mathcal{O}(\log L + \ell)$ time.

Corollary

Given b, we can store $\mathcal{O}(\log^* z)$ words such that, given any ℓ, we can extract $t[b-\ell..b+\ell]$ in $\mathcal{O}(\ell)$ time.

Bookmarking

Because we know that we will extract characters from the phrase boundaries, we can replace $\mathcal{O}(\log n + \ell)$ with the following bound:

Lemma

Given a balanced SLP for S with g rules and integers b and L, we can store $2\log g + \mathcal{O}(\log L)$ bits such that later, given $\ell \leq L$, we can extract $t[b-\ell..b+\ell]$ in $\mathcal{O}(\log L + \ell)$ time.

Corollary

Given b, we can store $\mathcal{O}(\log^* z)$ words such that, given any ℓ, we can extract $t[b-\ell..b+\ell]$ in $\mathcal{O}(\ell)$ time.

Bookmarking

Because we know that we will extract characters from the phrase boundaries, we can replace $\mathcal{O}(\log n + \ell)$ with the following bound:

Lemma

Given a balanced SLP for S with g rules and integers b and L, we can store $2 \log g + \mathcal{O}(\log L)$ bits such that later, given $\ell \leq L$, we can extract $t[b - \ell..b + \ell]$ in $\mathcal{O}(\log L + \ell)$ time.

Corollary

Given b, we can store $\mathcal{O}(\log^* z)$ words such that, given any ℓ, we can extract $t[b - \ell..b + \ell]$ in $\mathcal{O}(\ell)$ time.

Space bounds (in words)

Patricia trees	$\mathcal{O}(z)$
bookmarks	$\mathcal{O}(z \log^* z)$
4-sided 2D range reporting	$\mathcal{O}(z \log \log z)$
2-sided 2D range reporting	$\mathcal{O}(z)$
	$\mathcal{O}(z \log \log z)$

Time bounds

searching in compacted tries (with perfect hashing if necessary)	$\mathcal{O}(m^2)$
extracting from bookmarks	$\mathcal{O}(m^2)$
4-sided 2D range reporting	$\mathcal{O}(m \log\log n)$
2-sided 2D range reporting	$\mathcal{O}(occ \log\log n)$
	$\mathcal{O}(m^2 + (m + occ)\log\log n)$

A simple trick to remove the $m \log \log n$:

For each node of the compacted trie (of the prefixes), store a 1D range reporting structure with the pre-orders in the other compacted trie:

Alstrup, Brodal, Rauhe STOC 2001

1D range reporting on z points can be solved in $\mathcal{O}(z)$ space and optimal $\mathcal{O}(1 + occ)$ query time.

If $m \leq \log \log n$ then use the 1D range reporting structure, otherwise m^2 subsumes the $m \log \log n$ anyway.

A simple trick to remove the $m \log \log n$:

For each node of the compacted trie (of the prefixes), store a 1D range reporting structure with the pre-orders in the other compacted trie:

Alstrup, Brodal, Rauhe STOC 2001

1D range reporting on z points can be solved in $\mathcal{O}(z)$ space and optimal $\mathcal{O}(1 + occ)$ query time.

If $m \leq \log \log n$ then use the 1D range reporting structure, otherwise m^2 subsumes the $m \log \log n$ anyway.

Final result

Theorem

Given a balanced SLP for a string $t[1..n]$ whose LZ77 parse consists of z phrases, we can add $\mathcal{O}(z \log\log z)$ words such that, given a pattern $p[1..m]$, we can find all occ occurrences of p in $\mathcal{O}(m^2 + occ \log\log n)$ time.

Can we decrease m^2?

Final result

Theorem

Given a balanced SLP for a string $t[1..n]$ whose LZ77 parse consists of z phrases, we can add $\mathcal{O}(z \log\log z)$ words such that, given a pattern $p[1..m]$, we can find all occ occurrences of p in $\mathcal{O}(m^2 + occ \log\log n)$ time.

Can we decrease m^2?

Theorem

We can store a string $t[1..n]$ *whose LZ77 parse consists of* z *phrases in* $\mathcal{O}(z \log n)$ *space, so that later, given a pattern* $p[1..m]$, *we can find all* occ *occurrences of* p *in* s *in* $\mathcal{O}(m \log m + occ \log \log n)$ *time.*

A bit technical, but let's try!

Theorem

We can store a string $t[1..n]$ *whose LZ77 parse consists of* z *phrases in* $\mathcal{O}(z \log n)$ *space, so that later, given a pattern* $p[1..m]$*, we can find all* occ *occurrences of* p *in* s *in* $\mathcal{O}(m \log m + occ \log\log n)$ *time.*

A bit technical, but let's try!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi(s) = \sum_{k=1}^{|s|} S[k]\sigma^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi(s) = \sum_{k=1}^{|s|} S[k]\sigma^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi_r(s) = \sum_{k=1}^{|s|} s[k] r^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi_r(s) = \sum_{k=1}^{|s|} s[k] r^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi_r(s) = \sum_{k=1}^{|s|} s[k] r^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

Karp-Rabin fingerprints

Checking if two strings are equal can be done by comparing their fingerprints:

Karp-Rabin-style fingerprints

$$\phi_r(s) = \sum_{k=1}^{|s|} s[k] r^{|s|-k} \bmod p$$

Lemma

For a prime p and $r \in \{1, 2, \ldots, p-1\}$ chosen uniformly at random, the probability that $\phi_r(s) = \phi_r(s')$ even though $s \neq s'$ is at most $\frac{|s|}{p-1}$.

The cool property is that given the fingerprints of s and t we can compute the fingerprint of st!

z-fast tries

We want to preprocess a compacted trie T on n nodes for navigating with a query string x. In this application, it is enough to find the unique (implicit or explicit) node of T that corresponds to the whole x, if such a node exists, and otherwise return any node. However, the procedure will in fact do a bit more.

2-fattest number

The 2-fattest number in a nonempty interval of positive integers is the number in the interval whose binary representation has the highest number of trailing zeros

For every edge of T, we choose the implicit node on the edge whose string depth is the 2-fattest number in the corresponding range, and make it explicit.

z-fast tries

We want to preprocess a compacted trie T on n nodes for navigating with a query string x. In this application, it is enough to find the unique (implicit or explicit) node of T that corresponds to the whole x, if such a node exists, and otherwise return any node. However, the procedure will in fact do a bit more.

2-fattest number

The 2-fattest number in a nonempty interval of positive integers is the number in the interval whose binary representation has the highest number of trailing zeros

For every edge of T, we choose the implicit node on the edge whose string depth is the 2-fattest number in the corresponding range, and make it explicit.

z-fast tries

We want to preprocess a compacted trie T on n nodes for navigating with a query string x. In this application, it is enough to find the unique (implicit or explicit) node of T that corresponds to the whole x, if such a node exists, and otherwise return any node. However, the procedure will in fact do a bit more.

2-fattest number

The 2-fattest number in a nonempty interval of positive integers is the number in the interval whose binary representation has the highest number of trailing zeros

For every edge of T, we choose the implicit node on the edge whose string depth is the 2-fattest number in the corresponding range, and make it explicit.

Search in a z-fast trie

This can be done in only $\mathcal{O}(\log |x|)$ iterations:

Algorithm 1 Querying the probabilistic z-fast trie (represented by the function T).

input $x \in u$
$i \leftarrow \lceil \log w \rceil - 1$
$\ell, r \leftarrow 0, w$
while $r - \ell > 1$ **do**
  if $\exists b$ such that $2^i b \in (\ell \,..\, r)$ **then**
    $\{2^i b$ is the 2-fattest number in $(\ell \,..\, r)\}$
    $q \leftarrow$ prefix of x of length $2^i b$
    $\langle g, s \rangle \leftarrow T(q)$
    if $g \leq |x|$ and s is the signature of the prefix of x of length g **then**
      $\ell \leftarrow g$ {Move from $(\ell \,..\, r)$ to $(g \,..\, r)$}
    else
      $r \leftarrow 2^i b$ {Move from $(\ell \,..\, r)$ to $(\ell \,..\, 2^i b)$}
    end if
  end if
  $i \leftarrow i - 1$
end while
return ℓ

For a given suffix $p[i..m]$, this allows us to find the unique (implicit or explicit) node of the compacted trie.

Fingerprinting

Given a balanced SLP of size g, we can store $\mathcal{O}(g)$ words of extra information such that we can compute the fingerprint of any substring in $\mathcal{O}(\log n)$ time.

Bookmarked fingerprinting

Given a balanced SLP of size g and an integer b, we can store $\mathcal{O}(\log \log n)$ words of extra information such that later, given ℓ, we can compute the fingerprint of any $t[b..b+\ell)$ in $\mathcal{O}(\log \ell)$ time.

For a given suffix $p[i..m]$, this allows us to find the unique (implicit or explicit) node of the compacted trie.

Fingerprinting

Given a balanced SLP of size g, we can store $\mathcal{O}(g)$ words of extra information such that we can compute the fingerprint of any substring in $\mathcal{O}(\log n)$ time.

Bookmarked fingerprinting

Given a balanced SLP of size g and an integer b, we can store $\mathcal{O}(\log\log n)$ words of extra information such that later, given ℓ, we can compute the fingerprint of any $t[b..b+\ell)$ in $\mathcal{O}(\log \ell)$ time.

For a given suffix $p[i..m]$, this allows us to find the unique (implicit or explicit) node of the compacted trie.

Fingerprinting

Given a balanced SLP of size g, we can store $\mathcal{O}(g)$ words of extra information such that we can compute the fingerprint of any substring in $\mathcal{O}(\log n)$ time.

Bookmarked fingerprinting

Given a balanced SLP of size g and an integer b, we can store $\mathcal{O}(\log\log n)$ words of extra information such that later, given ℓ, we can compute the fingerprint of any $t[b..b+\ell)$ in $\mathcal{O}(\log \ell)$ time.

We use bookmarked fingerprinting to check if the found node has the same fingerprint as $p[i..m]$ (of course, this might be a false positive). Then, we proceed as before.

We now have queries in $\mathcal{O}(m \log m + (m + occ) \log \log n)$ time and $\mathcal{O}(z \log n)$ space. How to remove $m \log \log n$)?

1. If $m \geq \log n$, it is subsumed by $m \log m$.
2. If $m < \log n$, we store extra information at the top $\log n$ levels of the compacted trie.

We use bookmarked fingerprinting to check if the found node has the same fingerprint as $p[i..m]$ (of course, this might be a false positive). Then, we proceed as before.

We now have queries in $\mathcal{O}(m \log m + (m + occ) \log \log n)$ time and $\mathcal{O}(z \log n)$ space. How to remove $m \log \log n$)?

1. If $m \geq \log n$, it is subsumed by $m \log m$.
2. If $m < \log n$, we store extra information at the top $\log n$ levels of the compacted trie.

We use bookmarked fingerprinting to check if the found node has the same fingerprint as $p[i..m]$ (of course, this might be a false positive). Then, we proceed as before.

We now have queries in $\mathcal{O}(m \log m + (m + occ) \log \log n)$ time and $\mathcal{O}(z \log n)$ space. How to remove $m \log \log n$)?

1. If $m \geq \log n$, it is subsumed by $m \log m$.
2. If $m < \log n$, we store extra information at the top $\log n$ levels of the compacted trie.

We use bookmarked fingerprinting to check if the found node has the same fingerprint as $p[i..m]$ (of course, this might be a false positive). Then, we proceed as before.

We now have queries in $\mathcal{O}(m \log m + (m + occ) \log \log n)$ time and $\mathcal{O}(z \log n)$ space. How to remove $m \log \log n$)?

1. If $m \geq \log n$, it is subsumed by $m \log m$.
2. If $m < \log n$, we store extra information at the top $\log n$ levels of the compacted trie.

We use bookmarked fingerprinting to check if the found node has the same fingerprint as $p[i..m]$ (of course, this might be a false positive). Then, we proceed as before.

We now have queries in $\mathcal{O}(m \log m + (m + occ) \log\log n)$ time and $\mathcal{O}(z \log n)$ space. How to remove $m \log\log n$)?

1. If $m \geq \log n$, it is subsumed by $m \log m$.
2. If $m < \log n$, we store extra information at the top $\log n$ levels of the compacted trie.

The most interesting trick is derandomization.

Bille, Cording, Gørtz, Sach, Vildhøj, Vind WADS 2013

After $\mathcal{O}(n \log n)$ time preprocessing, we can be sure that there are no collisions among subtrings of length 2^k.

Do you see how to use this?

The most interesting trick is derandomization.

Bille, Cording, Gørtz, Sach, Vildhøj, Vind WADS 2013

After $\mathcal{O}(n \log n)$ time preprocessing, we can be sure that there are no collisions among subtrings of length 2^k.

Do you see how to use this?

What's next?

Let r be the number of runs in the BWT of the text.

Gagie, Navarro, Prezza JACM 2020

An index taking $\mathcal{O}(r \log(n/r))$ words and generating all *occ* occurrences in $\mathcal{O}(m + occ)$ time.

Let $\delta = \max_{\ell=1}^{n} d_\ell/\ell$, where d_ℓ is the number of distinct length-ℓ substrings of the text.

Kempa and Kociumaka FOCS 2023

An index taking $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ words and allowing suffix array and inverse suffix array queries in $\mathcal{O}(\log^{4+\epsilon} n)$ time.

What's next?

Let r be the number of runs in the BWT of the text.

Gagie, Navarro, Prezza JACM 2020

An index taking $\mathcal{O}(r\log(n/r))$ words and generating all *occ* occurrences in $\mathcal{O}(m + occ)$ time.

Let $\delta = \max_{\ell=1}^{n} d_\ell/\ell$, where d_ℓ is the number of distinct length-ℓ substrings of the text.

Kempa and Kociumaka FOCS 2023

An index taking $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ words and allowing suffix array and inverse suffix array queries in $\mathcal{O}(\log^{4+\epsilon} n)$ time.

Questions?