Statistical Encoding of Succinct Data Structures

Rodrigo González[1] Gonzalo Navarro[1]

[1]Department of Computer Science
Universidad de Chile

Combinatorial Pattern Matching, 2006

Outline

1. Background
   - Motivation
   - The k-th order empirical entropy
   - Statistical encoding
2. Entropy-bound succinct data structure
   - Idea
   - Data structures
   - Decoding Algorithm
   - Space requirement
   - Supporting appends
3. Application to full-text indexing
   - Succinct full-text self-indexes
   - The Burrows-Wheeler Transform
   - The wavelet tree

Outline

Outline

1. Background
   - Motivation
   - The k-th order empirical entropy
   - Statistical encoding
2. Entropy-bound succinct data structure
   - Idea
   - Data structures
   - Decoding Algorithm
   - Space requirement
   - Supporting appends
3. Application to full-text indexing
   - Succinct full-text self-indexes
   - The Burrows-Wheeler Transform
   - The wavelet tree

Outline

Motivation

Previous work

- In recent work, Sadakane and Grossi [SODA'06] introduced a scheme to represent any sequence S using $nH_k(S) + O(\frac{n}{\log_\sigma n}((k+1)\log\sigma + \log\log n))$ bits of space.
- The representation permits us to extract any substring of size $\Theta(\log_\sigma n)$ in constant time, and thus it completely replaces S under the RAM model.
- This permits converting any succinct structure using $o(n\log\sigma)$ bits of space on top of S, into a compressed structure using $nH_k(S) + o(n\log\sigma)$ bits overall, for any $k = o(\log_\sigma n)$.

Motivation

Previous work

- In recent work, Sadakane and Grossi [SODA'06] introduced a scheme to represent any sequence S using $nH_k(S) + O(\frac{n}{\log_\sigma n}((k+1)\log\sigma + \log\log n))$ bits of space.
- The representation permits us to extract any substring of size $\Theta(\log_\sigma n)$ in constant time, and thus it completely replaces S under the RAM model.
- This permits converting any succinct structure using $o(n\log\sigma)$ bits of space on top of S, into a compressed structure using $nH_k(S) + o(n\log\sigma)$ bits overall, for any $k = o(\log_\sigma n)$.

Motivation

Previous work

- In recent work, Sadakane and Grossi [SODA'06] introduced a scheme to represent any sequence S using $nH_k(S) + O(\frac{n}{\log_\sigma n}((k+1)\log\sigma + \log\log n))$ bits of space.
- The representation permits us to extract any substring of size $\Theta(\log_\sigma n)$ in constant time, and thus it completely replaces S under the RAM model.
- This permits converting any succinct structure using $o(n\log\sigma)$ bits of space on top of S, into a compressed structure using $nH_k(S) + o(n\log\sigma)$ bits overall, for any $k = o(\log_\sigma n)$.

Motivation

Our work

- We extend previous works, by obtaining slightly better space complexity and the same time complexity using a simpler scheme based on statistical encoding.
- We show that the scheme supports appending symbols in constant amortized time.
- We prove some results on the applicability of the scheme for full-text self-indexing.

Motivation

Our work

- We extend previous works, by obtaining slightly better space complexity and the same time complexity using a simpler scheme based on statistical encoding.
- We show that the scheme supports appending symbols in constant amortized time.
- We prove some results on the applicability of the scheme for full-text self-indexing.

Motivation

Our work

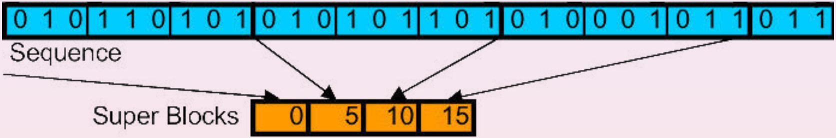
- We extend previous works, by obtaining slightly better space complexity and the same time complexity using a simpler scheme based on statistical encoding.
- We show that the scheme supports appending symbols in constant amortized time.
- We prove some results on the applicability of the scheme for full-text self-indexing.

Example: a simple rank structure

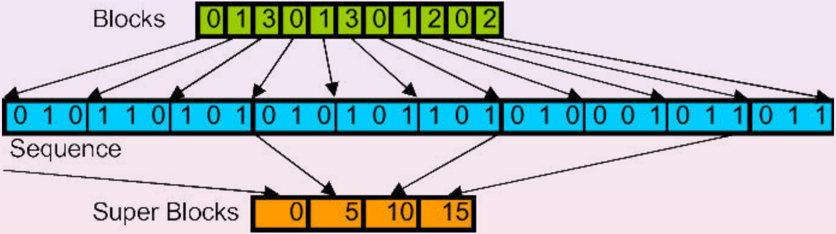
Definition

$rank_1(S, i)$ = number of ones in $S[1 \ldots i]$.

Example: a simple rank structure

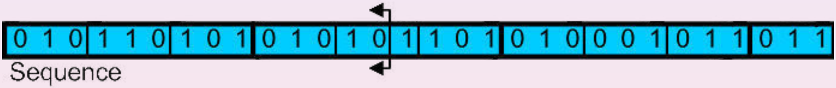


Example: a simple rank structure



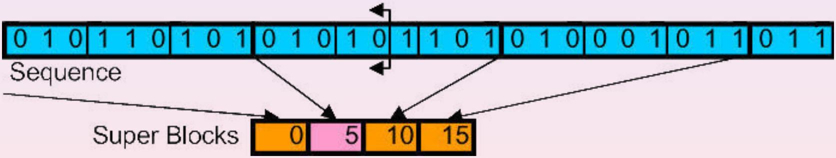
Example: a simple rank structure

rank(14)=

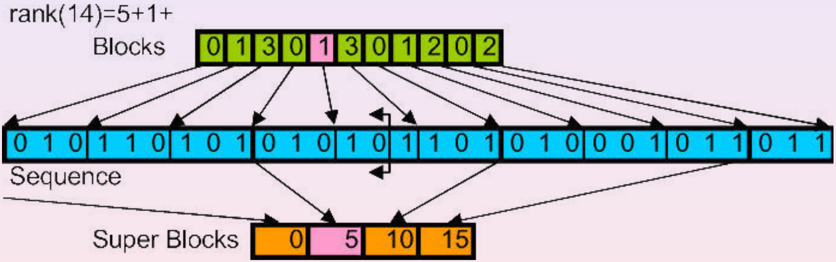


Example: a simple rank structure

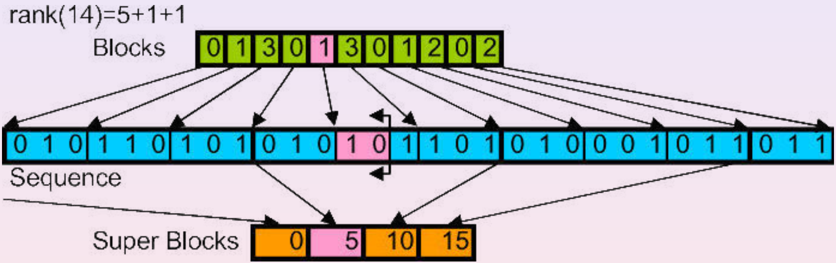
rank(14)=5+



Example: a simple rank structure



Example: a simple rank structure



$$rank_1(S, 14) = 5 + 1 + 1.$$

Outline

The k-th order empirical entropy

Definition

- The empirical entropy is defined for any string S and can be used to measure the performance of compression algorithms without any assumption on the input.
- The k-th order empirical entropy captures the dependence of symbols upon their context. For $k \geq 0$, $nH_k(S)$ provides a lower bound to the output of any compressor that considers a context of size k to encode every symbol of S.

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| H_0(w_S). \qquad (1)$$

The k-th order empirical entropy

Definition

- The empirical entropy is defined for any string S and can be used to measure the performance of compression algorithms without any assumption on the input.
- The k-th order empirical entropy captures the dependence of symbols upon their context. For $k \geq 0$, $nH_k(S)$ provides a lower bound to the output of any compressor that considers a context of size k to encode every symbol of S.

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| H_0(w_S) . \quad (1)$$

Outline

Semi-static Statistical encoding

Descriptions

- Given a k-th order modeler, which will yield the probabilities $p_1, p_2, \ldots, p_n$ for the symbols, we will encode the successive symbols of S trying to use $-\log p_i$ bits for s_i. If we reach exactly $-\log p_i$ bits, the overall number of bits produced will be $nH_k(S) + O(k \log n)$.
- Different encoders provide different approximations to the ideal $-\log p_i$ bits (Huffman coding, Arithmetic coding).
- Given a statistical encoder E and a semi-static modeler over sequence $S[1, n]$, we call $E(S)$ the bitwise output of E. We call $f_k(E, S)$ the extra space in bits needed to encode S using E, on top of $nH_k(S)$.

Semi-static Statistical encoding

Descriptions

- Given a k-th order modeler, which will yield the probabilities $p_1, p_2, \ldots, p_n$ for the symbols, we will encode the successive symbols of S trying to use $-\log p_i$ bits for s_i. If we reach exactly $-\log p_i$ bits, the overall number of bits produced will be $nH_k(S) + O(k \log n)$.
- Different encoders provide different approximations to the ideal $-\log p_i$ bits (Huffman coding, Arithmetic coding).
- Given a statistical encoder E and a semi-static modeler over sequence $S[1, n]$, we call $E(S)$ the bitwise output of E. We call $f_k(E, S)$ the extra space in bits needed to encode S using E, on top of $nH_k(S)$.

Semi-static Statistical encoding

Descriptions

- Given a k-th order modeler, which will yield the probabilities $p_1, p_2, \ldots, p_n$ for the symbols, we will encode the successive symbols of S trying to use $-\log p_i$ bits for s_i. If we reach exactly $-\log p_i$ bits, the overall number of bits produced will be $nH_k(S) + O(k \log n)$.
- Different encoders provide different approximations to the ideal $-\log p_i$ bits (Huffman coding, Arithmetic coding).
- Given a statistical encoder E and a semi-static modeler over sequence $S[1, n]$, we call $E(S)$ the bitwise output of E. We call $f_k(E, S)$ the extra space in bits needed to encode S using E, on top of $nH_k(S)$.

Semi-static Statistical encoding

Encoders

- Arithmetic coding essentially expresses S using a number in $[0,1)$ which lies within a range of size $P = p_1 \cdot p_2 \cdots p_n$. We need $-\log P = -\sum \log p_i$ bits to distinguish a number within that range (plus two extra bits for technical reasons).
- These are usually some limitations to the near-optimality achieved by Arithmetic coding in practice. They are scaling, very low probabilities and adaptive encoding. None of them is a problem in our scheme.

Semi-static Statistical encoding

Encoders

- Arithmetic coding essentially expresses S using a number in $[0,1)$ which lies within a range of size $P = p_1 \cdot p_2 \cdots p_n$. We need $-\log P = -\sum \log p_i$ bits to distinguish a number within that range (plus two extra bits for technical reasons).
- These are usually some limitations to the near-optimality achieved by Arithmetic coding in practice. They are scaling, very low probabilities and adaptive encoding. None of them is a problem in our scheme.

Outline

Entropy-bound succinct data structure

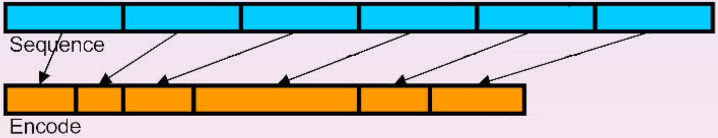
Idea

- Given a sequence $S[1, n]$ over an alphabet A of size σ, we encode S into a compressed data structure S' within entropy bounds. To perform all the original operations over S under the RAM model, it is enough to allow extracting any $b = \frac{1}{2} \log_\sigma n$ consecutive symbols of S, using S', in constant time.

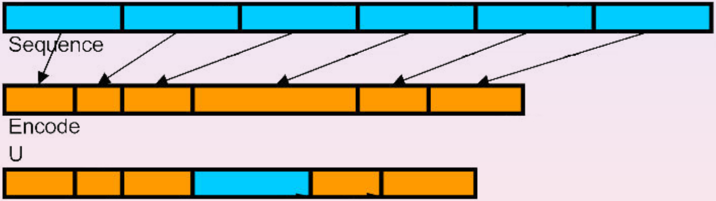
Outline

Data structures

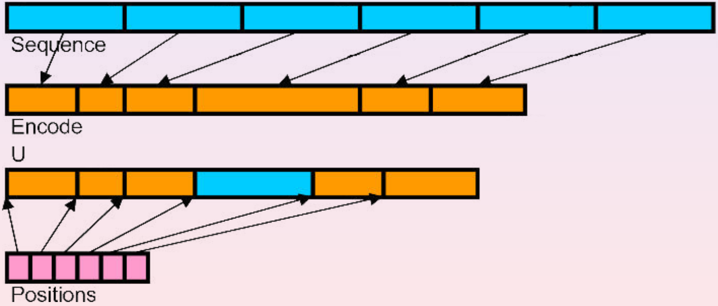
Data structures



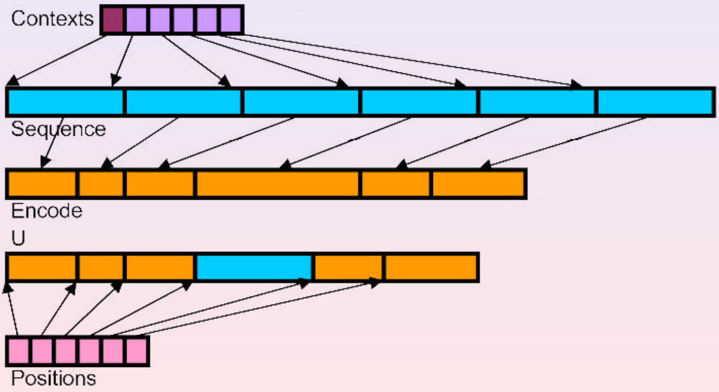
Data structures



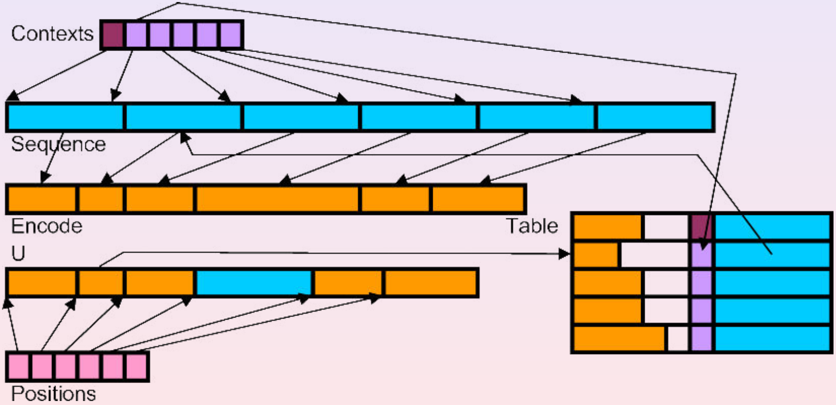
Data structures



Data structures



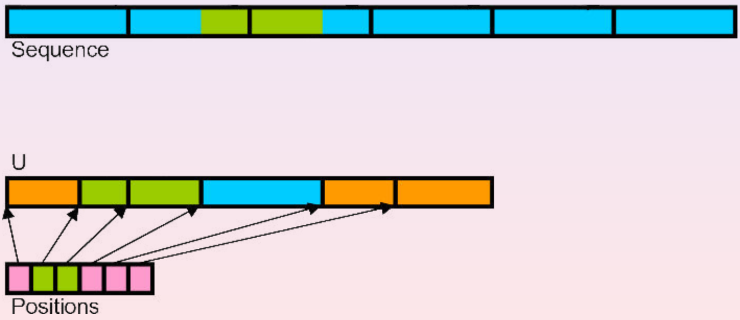
Data structures



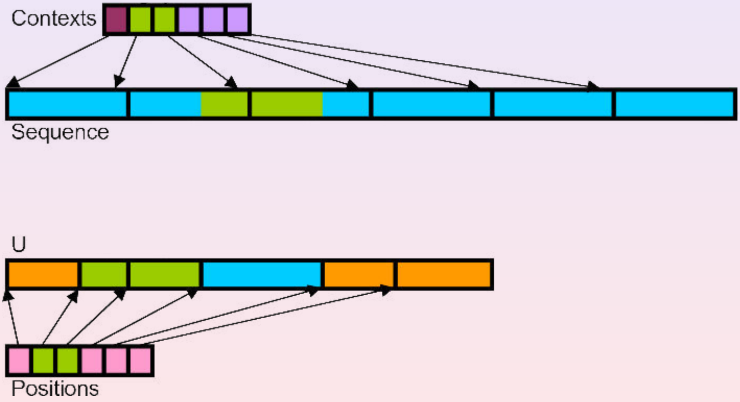
Outline

Decoding Algorithm

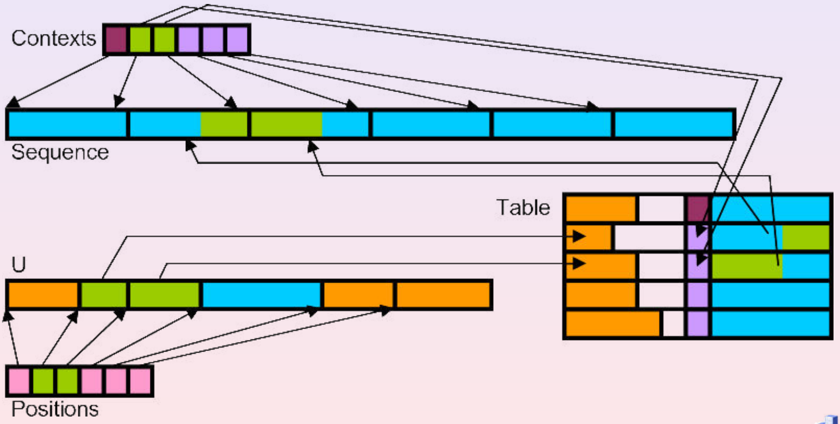
Decoding Algorithm



Decoding Algorithm



Decoding Algorithm



Outline

Space requirement

Size of U

- $|U| = \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \; n^{1/2} \log n / 2$

Space requirement

Size of U

- $|U| = \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \ n^{1/2} \log n/2$

Space requirement

Size of U

- $|U| = \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \; n^{1/2} \log n/2$

Space requirement

Size of U

- $|U| = \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \; n^{1/2} \log n/2$

Space requirement

Size of U

- $|U| =\leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \; n^{1/2} \log n/2$

Space requirement

Size of U

- $|U| = \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used.
- Huffman: $f_k(\text{Huffman}, S_i) < b$, thus we achive $nH_k(S) + O(k \log n) + n$ bits.
- Arithmetic: $f_k(\text{Arithmetic}, S_i) \leq 2$, thus we achive $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.

Other structures

- Contexts: $(n/b)k \log \sigma = O(nk \log \sigma / \log_\sigma n)$
- Positions: $O(n \log \log n / \log_\sigma n)$
- Table: $\sigma^k \ n^{1/2} \log n/2$

Space requirement

Theorem

Let $S[1, n]$ be a sequence over an alphabet A of size σ. Our data structure uses $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log\log n))$ bits of space for any $k < (1 - \epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$, and it supports access to any substring of S of size $\Theta(\log_\sigma n)$ symbols in $O(1)$ time.

Corollary

Our structure takes space $nH_k(S) + o(n \log \sigma)$ if $k = o(\log_\sigma n)$.

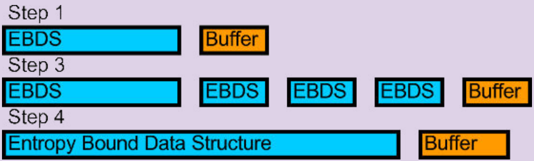
Outline

Supporting appends

Theorem

The structure supports appending symbols in constant amortized time and retains the same space and query time complexities.

Append scheme



Outline

Succinct full-text self-indexes

Definition

- A succinct full-text index is an index that uses space proportional to the compressed text. Those indexes that contain sufficient information to recreate the original text are known as self-indexes. Some examples are the FM-index family and the LZ-index.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
  - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
  - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
  - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
  - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
  - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
  - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
    - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
    - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
    - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
  - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
  - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
  - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
  - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
  - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
  - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Burrows-Wheeler Transform (BWT)

BWT

- The FM-index family is based on the Burrows-Wheeler Transform (BWT). The BWT of a text T, $T^{bwt} = bwt(T)$, is a reversible transformation from strings to strings, which is easier to compress by local optimization methods.
- An important property of the transformation is: if $T[k] = T^{bwt}[i]$, then $T[k-1] = T^{bwt}[LF(i)]$, where
  - $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.
  - $C[c]$ is the total number of text characters which are alphabetically smaller than c.
  - $Occ(c, i)$ is the number of occurrences of character c in the prefix $T^{bwt}[1, i]$.
- This property permits navigating the text T backwards.

Succinct full-text self-indexes

Wavelet tree

- The original FM-index solves *Occ* by storing some directories over S and compressing S. To give constant-time access to S they require exponential space in σ.
- The wavelet tree $wt(S)$ built on S is a binary tree, built on the alphabet symbols, such that the root represents the whole alphabet and each node has the information telling which of its characters belongs to the left/right child.

Succinct full-text self-indexes

Wavelet tree

- The original FM-index solves *Occ* by storing some directories over S and compressing S. To give constant-time access to S they require exponential space in σ.
- The wavelet tree $wt(S)$ built on S is a binary tree, built on the alphabet symbols, such that the root represents the whole alphabet and each node has the information telling which of its characters belongs to the left/right child.

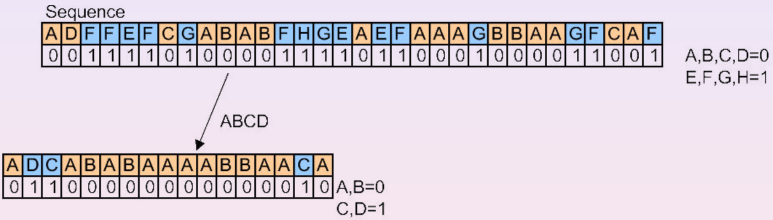
Wavelet tree

Sequence

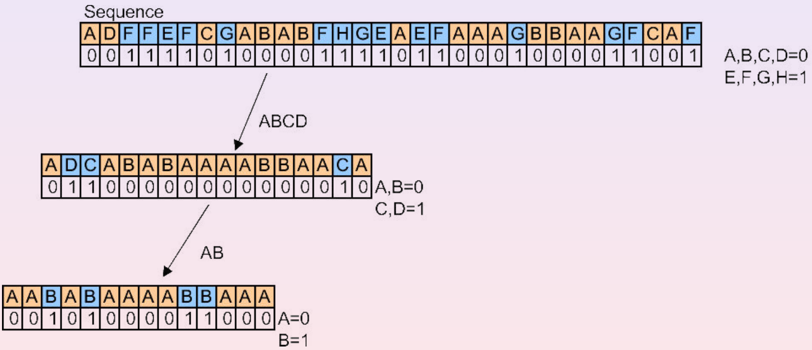
A	D	F	F	E	F	C	G	A	B	A	B	F	H	G	E	A	E	F	A	A	A	G	B	B	A	A	G	F	C	A	F
0	0	1	1	1	1	0	1	0	0	0	0	1	1	1	1	0	1	1	0	0	0	1	0	0	0	0	1	1	0	0	1

A,B,C,D=0
E,F,G,H=1

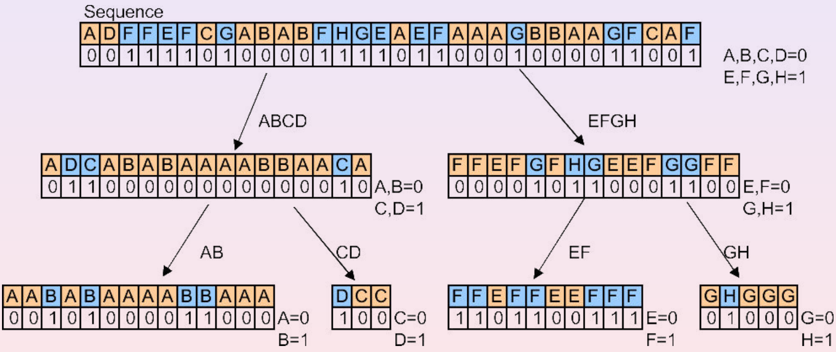
Wavelet tree



Wavelet tree



Wavelet tree



Outline

Relationship between T^{bwt} and T

We could encode $S = bwt(T)$ within $nH_k(S) + o(n \log \sigma)$ bits, but how this relates to $nH_k(T)$?

Lemma

Let $S = bwt(T)$, where $T[1, n]$ is a text over an alphabet of size σ. Then $H_1(S) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k < (1 - \epsilon) \log_\sigma n$ and any constant $0 < \epsilon < 1$.

Relationship between T^{bwt} and T

Application

- We can get at least the same results of the Run-Length FM-Index by compressing $bwt(T)$ using our structure.
- We can implement the original FM-index ($5nHk(T) + O(n\sigma \log\log n / \log_\sigma n + (\sigma/e)^{\sigma+3/2} n^\gamma \log_\sigma n \log\log n)$ bits) using $nH_k(T)\log\sigma + n + o(n)$ bits.

Outline

Relationship between $wt(S)$ and S

- $wt(S)$ takes $nH_0 + o(n \log \sigma)$ bits of space and permits answering *Occ* queries in time $O(\log \sigma)$
- Many FM-index variants build on the wavelet tree:
  - SSA takes $nH_0 + o(n \log \sigma)$ bits of space
  - RLFM-index takes $nH_k \log \sigma + o(n \log \sigma)$
  - AF-FM-index takes $nH_k + o(n \log \sigma)$
- In all cases the bitmaps of the $wt(S)$ are compressed to their H_0, but we can now compress them to H_k.

- Is k-th order entropy preserved across a wavelet tree? (it is for $k = 0$)

Lemma

The ratio between $H_k(wt(S))$ and $H_k(S)$, can be at least $\Omega(\log k)$. More precisely, $H_k(wt(S))/H_k(S)$ can be $\Omega(\log k)$ and $H_k(S)/H_k(wt(S))$ can be $\Omega(n/(k \log n))$.

Consequence

Applying our structure over the bitmaps of the wavelet tree does not perfectly translate into $nH_k(S)$ overall space, as there is a penalty factor of at least k in the worst case. But in the best, it can be much better than $nH_k(S)$.

Summary

- We presented a scheme based on k-th order modeling plus statistical encoding to convert any succinct data structure on sequences into a compressed data structure.
- This simplifies and slightly improves previous work.
- We presented a scheme to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol.
- We found relationships between the entropies of two fundamental structures used for compressed text indexing.

Summary

- We presented a scheme based on k-th order modeling plus statistical encoding to convert any succinct data structure on sequences into a compressed data structure.
- This simplifies and slightly improves previous work.
- We presented a scheme to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol.
- We found relationships between the entropies of two fundamental structures used for compressed text indexing.

Summary

- We presented a scheme based on k-th order modeling plus statistical encoding to convert any succinct data structure on sequences into a compressed data structure.
- This simplifies and slightly improves previous work.
- We presented a scheme to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol.
- We found relationships between the entropies of two fundamental structures used for compressed text indexing.

Summary

- We presented a scheme based on k-th order modeling plus statistical encoding to convert any succinct data structure on sequences into a compressed data structure.
- This simplifies and slightly improves previous work.
- We presented a scheme to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol.
- We found relationships between the entropies of two fundamental structures used for compressed text indexing.

Future work

- Making our structure fully dynamic
- Better understanding how the entropies evolve upon transformations such *bwt* or *wt*.
- Testing our structure in practice.
- Currently working on another way to solve the same problem. That would permit full dynamism using recent work (see next talk).

Future work

- Making our structure fully dynamic
- Better understanding how the entropies evolve upon transformations such *bwt* or *wt*.
- Testing our structure in practice.
- Currently working on another way to solve the same problem. That would permit full dynamism using recent work (see next talk).

Future work

- Making our structure fully dynamic
- Better understanding how the entropies evolve upon transformations such *bwt* or *wt*.
- Testing our structure in practice.
- Currently working on another way to solve the same problem. That would permit full dynamism using recent work (see next talk).

Future work

- Making our structure fully dynamic
- Better understanding how the entropies evolve upon transformations such *bwt* or *wt*.
- Testing our structure in practice.
- Currently working on another way to solve the same problem. That would permit full dynamism using recent work (see next talk).

Thank you!!