Burrows Wheeler Transform

Janosch Ruff

Georg-August-Universität Göttingen

December 22, 2020

Structure

Core idea

Definition

Data compression *is the process of encoding information by using fewer bits than its original representation.*

- Idea: Transform a string to make it "more compressible".
- Question: Why is a string more compressible than another?

What is compressible?
Kolmogorov Complexity

- "Kolmogorov complexity is a notion every computer scientist should know about but not every computer scientist knows."

Definition

*Fix a programming Language. Then the **Kolmogorov Complexity** $C(\cdot)$ for a string ω is defined by*

$$C(\omega) := \textit{Bit length of the shortest program that outputs } \omega \tag{1}$$

- "Kolmogorov complexity represents the limit for optimal compressors and thus is the Gold Standard for compression."

Compression examples
Stochastic Process

Figure 1: Flipping a coin

Example 1 a simple pattern: $10101010..101010$
Example 2 the first 128 bits of π:
$1100100100001111110110101010001000100001011010001100001000110$
$1001100010011000110011000101000101110000000110111000001110011$
$010001..$

Motivation

- Observation: A string with low Kolmogorov Complexity has a "structure" but some structures are simpler to see than others.
- Idea: Construct an algorithm to transform a string that discovers the patterns given by "low" Kolmogorov Complexity.
- Aim: Exploit discovered patterns to improve compression.

Kolmogorov Complexity
Properties

- The Kolmogorov Complexity is not computable.
- The Kolmogorov Complexity of a string is at most the length of the string itself $C(\omega) \leq |\omega| + \mathcal{O}(1)$ (ω hard coded).
- No "magic universal compressor" exists and $\exists \omega$ s.t. $C(\omega) \geq |\omega|$ (There are always strings that are not compressible).
- Let ϕ be a computable bijection then $C(\phi(\omega)) = C(\omega) + C(\phi) + \mathcal{O}(1) = C(\omega) + \mathcal{O}(1)$
- An incompressible string ξ with $C(\xi) = |\xi| + \mathcal{O}(1)$ is equivalent to an algorithmic random string.

Sorting
A simple idea

- Idea: Enforce a structure by sorting the string.
- Example original string:
  "kolmogorov complexity is a notion every computer scientist should know about but not every computer scientist knows."
- Example string after sorting:
  "aabbcccccdeeeeeeeeeghiiiiiiikkklllmmmmnnnnnnn oooooooooooooopppprrrrrsssssssttttttttttuuuuuvvvwwxyyy"
- Result: Long runs of letters and thus easier to compress.
- Problem: How to recover the original string from the string after being sorted?

Sorting

A little bit more sufisticated

- Next idea: Sort the word to discover potential patterns.
- Implementation: Output the letter which occurs in the original string before the sorted letter.
- Example: "BANANA$" $\xrightarrow{BWT}$ "ANNB$AA"

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

Properties

Questions

- Can the sorting be done in linear time $\mathcal{O}(n)$?
- Is it easy to find a good compressor φ for $BWT(\omega)$?
- Is the transformed string invertible $BWT^{-1}(BWT(\omega)) = \omega$?

Sorting

In linear time

- Notice that the green colored substrings is a Suffix Tree/ Suffix Array of the string "BANANA$"
  $BANANA
  A$BANAN
  ANA$BAN
  ANANA$B
  BANANA$
  NA$BANA
  NANA$BA
- Lemma: We can construct a Suffix Array in linear time.

Transformation

Using Suffix Array

- Concatenation $\alpha = \omega \cdot \$ \cdot \omega$
  (e.g. ω = "BANANA" $\xrightarrow{\alpha}$ "BANANA$BANANA"
- Create a suffix array of $\omega \cdot \$$
  (e.g. ω = "BANANA" $\xrightarrow{SA}$ [6,5,3,1,0,4,2]
  [6]$BANANA
  [5]A$BANAN
  [3]ANA$BAN
  [1]ANANA$B
  [0]BANANA$
  [4]NA$BANA
  [2]NANA$BA
- $\eta[i] = \alpha[SA[i] + n]$ where $|\omega| = n$ and $0 \leq i \leq n$.

Transformation

Algorithm

Algorithm 1 BWT(ω)

$\alpha \leftarrow \omega\$\omega$
$n \leftarrow |\omega|$
$\Gamma \leftarrow$ SuffixArray($\omega\$$)
$\eta \leftarrow \epsilon$
$i \leftarrow 0$
while $i \leq n$ **do**
  $\eta[i] \leftarrow \alpha[\Gamma[i] + n]$
  $i \leftarrow i + 1$
end while
return η

Properties
Answers

- Can the sorting be done in linear time $\mathcal{O}(n)$? ✓
- Is it easy to find a good compressor φ for $BWT(\omega)$?
- Is the transformed string invertible $BWT^{-1}(BWT(\omega)) = \omega$?

Compressibilty

Why $BWT(\omega)$ is easier to compress?

- Observation: $\eta[i]$ is the letter which occurs before $\omega_{sort}[i]$.
- This shows that η depends on a sorted list (e.g. consider the word "banana", then "n" occurs always before "a" and thus we can expect *runs* of "n".)
- There exists no "magic universal compressor" but we can apply a custom-made compression scheme φ exploiting the properties of a transformed string with predictable properties.
- Kolmogorov Complexity tells us $C(BWT(\omega)) = C(\omega) + \mathcal{O}(1)$ but BWT clusters the letters based on the structure of ω and reveals hidden patterns what makes it easier to compress for φ.

Properties
Answers

- Can the sorting be done in linear time $\mathcal{O}(n)$? ✓
- Is it easy to find a good compressor φ for $BWT(\omega)$? ✓
- Is the transformed string invertible $BWT^{-1}(BWT(\omega)) = \omega$?

Inverse Functions

Basics

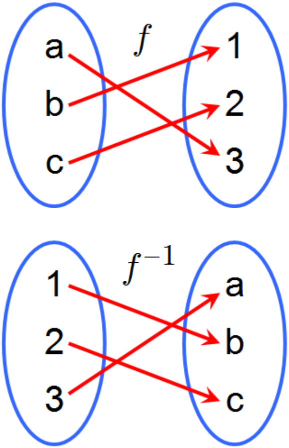

Figure 2: Example of an inverse function f

Unsort the sorted string

Availabe information

- Observation: $\eta[i]$ is the letter which occurs before $\omega_{sort}[i]$.
- This entails that $\eta[0]$ is the letter which occurs before $\omega_{sort}[0] = \$$ hence $\eta[0] = \omega[n-1]$ the last letter of ω
  $BANANA
  A$BANAN
  ANA$BAN
  ANANA$B
  BANANA$
  NA$BANA
  NANA$BA
- Idea: Reconstructing ω from η by finding the letter occuring before the letter $\eta[i]$ in ω.

First Last Column

Indexing

- Observation: We know in which row of the sorted list (first column) the element $\eta[k]$ can be found (because it is sorted).
- The last element in this row indicates the letter which occurs before $\omega_{sort}[k]$

  F..L

  $\$_1$..$A_1$

  A_1..N_1

  A_2..N_2

  A_3..B_1

  B_1..$\$_1$

  N_1..A_2

  N_2..A_3
- Observation: The structure of the first row is completely predictable since it is sorted and thus only the last column is needed to recreate ω from η hence, $\exists BWT^{-1}(\eta) = \omega$.

BWT^{-1}

Algorithm

Algorithm 2 $\mathsf{BWT}^{-1}(\eta)$

```
π ← permutation(η)
ω ← ϵ
n ← |η|
i ← n − 1
k ← 0
while i ≥ 0 do
    ω[i] ← η[k]
    i ← i − 1
    k ← π[k]
end while
return ω
```

Calculating the permutation
Algorithm

Algorithm 3 permutation(η)

$\pi \leftarrow [0] \cdot |\eta|$
$\Lambda, \vartheta \leftarrow [0] \cdot |\Sigma|$
for $i = 0, 1, .., |\eta| - 1$ **do**
    $\Lambda[\eta[i]] \leftarrow \Lambda[\eta[i]] + 1$
end for
for $i = 1, 2, .., |\eta| - 1$ **do**
    $\vartheta[i] \leftarrow \vartheta[i-1] + \Lambda[i-1]$
end for
for $i = 0, 1, 2, .., |\eta| - 1$ **do**
    $\pi[i] \leftarrow \vartheta[\eta[i]]$
    $\vartheta[\eta[i]] \leftarrow \vartheta[\eta[i]] + 1$
end for
return π

Properties

Answers

- Can the sorting be done in linear time $\mathcal{O}(n)$? $\checkmark$
- Is it easy to find a good compressor φ for $BWT(\omega)$? $\checkmark$
- Is the transformed string invertible $BWT^{-1}(BWT(\omega)) = \omega$? $\checkmark$

Further upgrades

What can be improved?

- Question: Is it possible to do the BWT without an additional special character $?

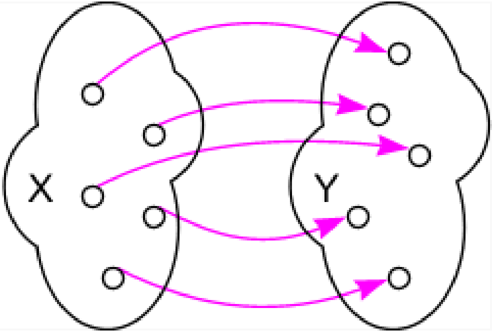

Figure 3: Example sketch of a bijective function

Bijection

Lyndon Words

Definition

*A **Lyndon Word** is strictly smaller than any of its proper suffixes and strictly smaller in lexicographic order than all of its rotations.*

- Lyndon Words of the binary alphabet:
  $0, 1, \cancel{00}, 01, \cancel{10}, \cancel{11}, \cancel{000}, 001, \cancel{010}, 011, \cancel{100}, \cancel{101}, \cancel{110}, \cancel{111}, \cancel{0000}, 0001, ..$
- Chen–Fox–Lyndon Theorem: Every word can be uniquely factored into $\omega_1\omega_2..\omega_m$ Lyndon words where $\omega_i \geq_{\text{lex}} \omega_{i+1}$.
- $"BANANA" \xrightarrow{\text{Factors}} "B" >_{\text{lex}} "AN" \geq_{\text{lex}} "AN" >_{\text{lex}} "A"$
- Compute a list of all Lyndon factors of ω and their respective rotations and sort this list by $u <_\omega v$ such that $uuu.. <_{\text{lex}} vvv..$
- $B|AN|AN|A \xrightarrow{\text{Rotations}} B|AN, NA|AN, NA|A$
  $\xrightarrow{\text{Sort}} A <_\omega AN \leq_\omega AN <_\omega B <_\omega NA \leq_\omega NA$
  $\xrightarrow{\text{Chars}} ANNBAA$

BBWT
Algorithm

Algorithm 4 BBWT(ω)

1: $\omega_1\omega_2..\omega_m \leftarrow$ LyndonFactors(ω)
2: $\chi \leftarrow \varnothing$
3: **for** $i = 1, 2, .., m$ **do**
4: $\quad\quad \chi \cup$ CyclicRotations(ω_i)
5: **end for**
6: $\tau \leftarrow$ OmegaSort(χ)
7: $\eta \leftarrow$ LastChars(τ)
8: **return** η

- Open Problem: Can line 6 the sorting based on comparing infinite periodic repetitions be done in linear time?

Inverting BBWT

Key Observations

- Observation: If $\eta = BBWT(\omega)$, then the last character of ω_m (and hence of ω) is $\eta[0]$.
- We can reconstruct ω by reconstructing the i-th Lyndon Factor ω_i by using the computation of the Algorithm permutation.
- Question: How do we notice that we completed the i-th Lyndon Factor and where do we proceed?
- Idea: Keep track of visited letters in η telling us if we would "recycle" this letter and we have to skip it in which case we notice that the last letter of ω_{i-1} is next to the last letter of ω_i.

BBWT^{-1}

Algorithm

Algorithm 5 BWT$^{-1}(\eta)$

```
π ← permutation(η)
ν ← [False] · |η|
j, k ← 0
for i = |η| − 1, |η| − 2, .., 0 do
    while ν[k] ≠ False do
        j ← j + 1
        k ← j
    end while
    ω[i] ← η[k]
    ν[k] ← True
    k ← π[k]
end for
return ω
```

Conclusion

Remarks

File	Size	BWT-compression		$\mathcal{S}$-compression		Gain	
		bytes	ratio	bytes	ratio	absolute	relative
BIB	111,261	32,022	28.78%	31,197	28.04%	0.74%	2.58%
BOOK1	768,771	242,857	31.59%	235,913	30.69%	0.90%	2.86%
BOOK2	610,856	170,783	27.96%	166,881	27.32%	0.64%	2.28%
GEO	102,400	66,370	64.81%	66,932	65.36%	-0.55%	-0.85%
NEWS	377,109	135,444	35.92%	131,944	34.99%	0.93%	2.58%
OBJ1	21,504	12,727	59.18%	12,640	58.78%	0.40%	0.68%
OBJ2	246,814	98,395	39.87%	94,565	38.31%	1.55%	3.89%
PAPER1	53,161	19,816	37.28%	18,931	35.61%	1.66%	4.47%
PAPER2	82,199	28,084	34.17%	27,242	33.14%	1.02%	3.00%
PAPER3	46,526	18,124	38.95%	17,511	37.64%	1.32%	3.38%
PAPER4	13,286	6,047	45.51%	5,920	44.56%	0.96%	2.10%
PAPER5	11,954	5,815	48.64%	5,670	47.43%	1.21%	2.49%
PAPER6	38,105	14,786	38.80%	14,282	37.48%	1.32%	3.41%
PIC	513,216	59,131	11.52%	52,406	10.21%	1.31%	11.37%
PROGC	39,611	15,320	38.68%	14,774	37.30%	1.38%	3.56%
PROGL	71,646	18,101	25.26%	17,916	25.01%	0.26%	1.02%
PROGP	49,379	13,336	27.01%	13,010	26.35%	0.66%	2.44%
TRANS	93,695	22,864	24.40%	22,356	23.86%	0.54%	2.22%
Total	3,251,493	980,022	30.14%	950,090	29.22%	0.92%	3.05%
Median	76,923	21,340	36.60%	20,644	35.30%	0.94%	2.58%

Figure 4: Comparing performance between BWT-based compression and S-based (BBWT) compression of the Calgary corpus.

Sources
Further material

- The original paper on Burrows-Wheeler Transform "A Block-sorting Lossless Data Compression Algorithm" by Burrows & Wheeler was published in 1994.
- The paper "A Bijective String Sorting Transform" by Gil & Scott has been used as the main source for the presentation and might serve for further studies in bijective BWT.
- The data compressor bzip2 uses BWT in combination with Move-to-front transform and Huffman coding to compress files.
- Another application of BWT has been found in the field of Bioinformatics in the form of FM-index here smoothly introduced by Ben Langmead to approximate string alignment.
- As a light introduction into the world of Kolmogorov Complexity or rather Algorithmic Information Theory the lecture given by Shai Ben-David is a nice starting point.

Appendix

Duval's Algorithm

- The algorithm originally published in the paper "Factorizing words over an ordered alphabet" by Duval in 1983 computes the Lyndon Factors of the word ω in linear time $\mathcal{O}(|\omega|)$.

```python
#Python implementation of Duval's Algorithm for Lyndon Factors
def duval(w):
  i = 0
  n = len(w)
  factors = []
  while(i < n):
    j = i + 1
    k = i
    while(j < n and w[k] <= w[j]):
      if(w[k] < w[j]):
        k = i
      else:
        k += 1
      j += 1
    while(i <= k):
      factors.append(w[i:i+j-k])
      i += j-k
  return(factors)
```