Prefix-free parsing with applications

Giovanni Manzini
University of Eastern Piedmont, Alessandria, Italy

Pisa, February 6h, 2020

Motivations

- We need to handle huge collections of data with lot of redundancy
- The compressed data fit in RAM and could be handled efficiently
- The bottleneck is the compression operation itself: we want to exploit repetitions even appearing very far in the input

Prefix Free Parsing

- Fix a window size w and a modulus p
- For each window W compute a KR fingerprint KR(W)
- If KR(W) mod p = 0, W is a trigger string
- Once a trigger string, always a trigger string

Example

T = catgattatacgattacattacatcatgattacatacg

T = catgattatacgattacattacatcatgattacatacg

T = catgattatacgattacattacatcatgattacatacg

T has many repetitions!

Example

Assume trigger strings: at & cg

T = `catgattatacgattacattacatcatgattacatacg`

- Distinct words: cat, gat, tat, acg, at, tacat
- Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
- Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1

Word IDs follow the lexicographic order

Idea

- Dictionary and Parse provide a representation of T that we can use instead of the original
- By changing the modulus p we can have shorter/longer phrases
- No dictionary word is the prefix of another one:
  - `acctaccat`
  - `acctaccatcctcg`

Not Possible!

Observations

- Terrible in the worst case ($T = a^n$) works very well in practice
- Dictionary and parsing can be computed in one scan in external memory and in parallel

Applications

- BWT computation (WABI ‘18)
- Grammar compression (SPIRE ‘19)
- Others (ongoing)

Grammar Compression

- Represent T by a context free grammar that only generates T (Straight Line Program)
- Example: grammar representing

  catgattatacgattacattacatcatgattacat

```
S → FXXcGVVFV
F → CG
V → TC
G → gA
C → cA
A → at
X → ta
```

Example:

X → ta

A → at

C → cA = cat

G → gA = gat

V → XC = tacat

F → CG = catgat

```
S → FXXcGVVFV
F → CG
V → XC
G → gA
C → cA
A → at
X → ta
```

S → F X X c G V V F V

S → catgat ta ta c gat tacat tacat catgat tacat

= T

Grammar Compression

- Very good compression for highly repetitive inputs
- Encoding represented as a binary tree
- We can often operate on the compressed representation

Application: Matrix Mult.

- We are given a possibly sparse matrix, with often same values in the same columns
- A suitable grammar compression can be used to compute matrix multiplication in time proportional to the grammar size

Example: We are working with the matrix

$$\begin{bmatrix} 2 & 4 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 7 \\ 2 & 4 & 3 & 5 & 3 \\ 4 & 0 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 0 \end{bmatrix}$$

Example: We are working with the matrix

$$\begin{bmatrix} 2 & 4 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 7 \\ 2 & 4 & 3 & 5 & 3 \\ 4 & 0 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 0 \end{bmatrix}$$

there are indeed some repeated patterns!

$$\begin{bmatrix} 2 & 4 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 7 \\ 2 & 4 & 3 & 5 & 3 \\ 4 & 0 & 6 & 0 & 3 \\ 3 & 0 & 3 & 5 & 0 \end{bmatrix} \rightarrow \begin{array}{l} a_{1,2}\, a_{2,4}\, a_{3,6}\, a_{5,3} \\ a_{1,3}\, a_{3,3}\, a_{4,5}\, a_{5,7} \\ a_{1,2}\, a_{2,4}\, a_{3,3}\, a_{4,5}\, a_{5,3} \\ a_{1,4}\, a_{3,6}\, a_{5,3} \\ a_{1,3}\, a_{3,3}\, a_{4,5} \end{array}$$

We encode each entry with its column index and value

$$s_1 \rightarrow r_2\, r_4$$

$$s_2 \rightarrow r_3\, a_{5,7}$$

$$s_3 \rightarrow r_2\, r_1\, a_{5,3}$$

$$s_4 \rightarrow a_{1,4}\, r_4$$

$$s_5 \rightarrow r_3$$

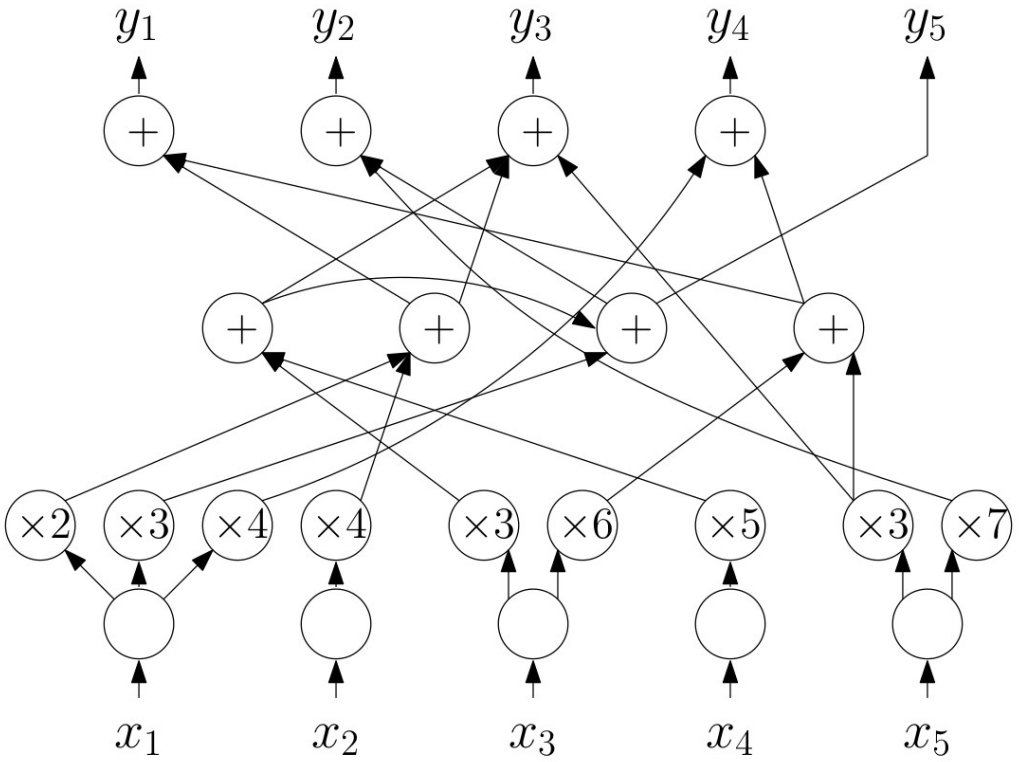
$$r_1 \rightarrow a_{3,3}\, a_{4,5}$$

$$r_2 \rightarrow a_{1,2}\, a_{2,4}$$

$$r_3 \rightarrow a_{1,3}\, r_1$$

$$r_4 \rightarrow a_{3,6}\, a_{5,3}$$

We build a grammar for each row, sharing non-terminals



The grammar provides a strategy for computing $y = Ax$ and $y^T = x^T A$ in # operations equal to the grammar size

Grammar construction

- The grammar construction tool used in practice is RePair [Larsson&Moffat ‘99]
- RePair is a greedy, linear time/space algorithm with good performance in practice
- Internal memory algorithm: becomes slow if we limit RAM

PFP+RePair = BigRePair

- We apply prefix free parsing and obtain a Dictionary and a Parsing
- Example:
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- We apply RePair to P and to D separately
- We combine the two grammars to get a grammar for T

Experimental results

File	Size	RePair			BigRePair			SOLCA			BigSOLCA		
		Ratio	Time	Spc	Ratio	Time	Spc	Ratio	Time	Spc	Ratio	Time	Spc
c50	2.75	0.80%	1832	3842	0.91%	29.30	454.7	1.35%	244.1	107.4	1.54%	66.47	183.4
c100	5.51	0.30%	7311	3155	0.48%	25.05	246.4	0.77%	236.4	53.67	0.86%	56.96	130.4
c250	13.8				0.23%	22.10	119.8	0.40%	239.0	29.78	0.44%	48.55	95.00
c500	27.5				0.14%	22.31	118.0	0.28%	237.4	17.05	0.30%	47.46	84.72
c1000	55.1				0.10%	22.61	117.3	0.22%	237.3	13.56	0.23%	47.79	78.82
s815	3.75	1.72%	8478	3726	1.93%	51.70	2254	3.01%	317.7	161.0	3.50%	104.1	291.4
s2073	9.72				2.01%	55.48	1055	3.01%	370.9	153.1	3.53%	116.9	285.9
s4570	22.0				2.61%	201.1	534.2	3.57%	480.6	154.4	4.24%	142.8	335.1
s11264	53.1				1.51%	2560	294.2	2.20%	620.2	92.60	2.61%	113.1	206.7

Time: sec/GB, Space: MB/GB, RAM: 10GB

xz: good compression, but 50 times slower than BigRePair

BWT construction

- The Burrows-Wheeler Transform is a data transformation useful for Data Compression and Indexing
- Unfortunately it is a global transformation, we have to see all the input before producing a single output

The BWT

`swiss·miss·missing`

The BWT

swiss·miss·missing

Consider all rotations of the input text

```
s wiss·miss·missin g
w iss·miss·missing s
i ss·miss·missings w
s s·miss·missingsw i
s ·miss·missingswi s
· miss·missingswis s
m iss·missingswiss ·
i ss·missingswiss· m
s s·missingswiss·m i
s ·missingswiss·mi s
· missingswiss·mis s
m issingswiss·miss ·
i ssingswiss·miss· m
s singswiss·miss·m i
s ingswiss·miss·mi s
i ngswiss·miss·mis s
n gswiss·miss·miss i
g swiss·miss·missi n
```

The BWT

`swiss·miss·missing`

Consider all rotations of the input text

Sort them in lexicographic order

```
· miss·missingswis s
· missingswiss·mis s
g swiss·miss·missi n
i ngswiss·miss·mis s
i ss·miss·missings w
i ss·missingswiss· m
i ssingswiss·miss· m
m iss·missingswiss ·
m issingswiss·miss ·
n gswiss·miss·miss i
s ·miss·missingswi s
s ·missingswiss·mi s
s ingswiss·miss·mi s
s s·miss·missingsw i
s s·missingswiss·m i
s singswiss·miss·m i
s wiss·miss·missin g
w iss·miss·missing s
```

The BWT

swiss·miss·missing

Consider all rotations of the input text

Sort them in lexicographic order

Take the last character of each rotation

ssnswmm··isssiiigs

		L
·	miss·missingswis	s
·	missingswiss·mis	s
g	swiss·miss·missi	n
i	ngswiss·miss·mis	s
i	ss·miss·missings	w
i	ss·missingswiss·	m
i	ssingswiss·miss·	m
m	iss·missingswiss	·
m	issingswiss·miss	·
n	gswiss·miss·miss	i
s	·miss·missingswi	s
s	·missingswiss·mi	s
s	ingswiss·miss·mi	s
s	s·miss·missingsw	i
s	s·missingswiss·m	i
s	singswiss·miss·m	i
s	wiss·miss·missin	g
w	iss·miss·missing	s

Algorithm Idea

- Instead of the original text
  - T = `catgattatacgattacattacatcatgattacatacg`
- We work with
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1

Algorithm Idea

- Instead of the original text
  - T = `catgattatacgattacattacatcatgat`tacatacg
- We work with
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Observe that each suffix of T consists of a suffix of a dictionary word followed by full dictionary words
  - `tacatacg` = `tacat 1`

Algorithm Idea

- Instead of the original text
  - T = `catgattatacgattacatt`**`acatcatgattacatacg`**
- We work with
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Observe that each suffix of T consists of a suffix of a dictionary word followed by full dictionary words
  - **`acatcatgattacatacg`** `= acat 3,4,5,1`

First attempt

- We first compute the lexicographic order of all the dictionary word suffixes. Is this enough?
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Example of lexicographic comparisons:
  - `tacatacg = tacat 1`
  - `ttacatacg = t 5,1`
  - `acatcatgattacatacg = acat 3,4,5,1`

First attempt

- We first compute the lexicographic order of all the dictionary word suffixes. Is this enough?
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Example of lexicographic comparisons:
  - `tacatacg = tacat 1`
  - `catgattacatacg = cat 4,5,1`

tacat > cat
the first suffix is larger!

First attempt

- We first compute the lexicographic order of all the dictionary word suffixes. Is this enough?
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Example of lexicographic comparisons:
  - `tacatacg = tacat 1`
  - `tacatcatgattacatacg = tacat 3,4,5,1`

tacat = tacat, 1<3
the first suffix is smaller!

First attempt

- We first compute the lexicographic order of all the dictionary word suffixes. Is this enough?
  - Dictionary D = {1:acg, 2:at, 3:cat, 4:gat, 5:tacat, 6:tat}
  - Parsing P = 3,4,6,1,2,5,5,3,3,4,5,1
- Example of lexicographic comparisons:
  - `tacatacg = tacat 1`
  - `ttacatacg = t 5,1`

t = t, acat$\leqslant$5 ?
no idea! ☹

Solution

- We modify the dictionary: splitting strings assigned to both current & next word
  - T= `$catgattatacgattacattacatcatgattacatacg`
  - Dictionary D = {1:$cat, 2:atacg, 3:atcat, 4:atgat, 5:attacat, 6: attat, 7:cgat}
  - Parsing P = 1,4,6,2,7,5,5,3,4,5,2
- This is a prefix & suffix free parsing!

Experimental results

Salmonella genomes

Number of genomes	Size	$w = 6, p = 20$			$w = 8, p = 50$			$w = 10, p = 100$		
		Dict.	Parse	%	Dict.	Parse	%	Dict.	Parse	%
50	249	68	43	44	*77*	*20*	*39*	91	10	40
100	485	83	85	35	*99*	*39*	*28*	122	19	29
500	2436	273	424	29	314	194	21	*377*	*96*	*19*
1000	4861	475	847	27	541	388	19	*643*	*192*	*17*
5000	24936	2663	4334	28	2915	1987	20	*3196*	*985*	*17*
10,000	49420	4190	8611	26	4652	3939	17	*5176*	*1955*	*14*

Dictionary and Parse size as a function of w,p

- Size in MBs
- Percentages: (Dict+Parse)/Text

Experimental results

Salmonella genomes

Number of genomes	$w = 6, p = 20$		$w = 8, p = 50$		$w = 10, p = 100$		`simplebwt`	
	Time	Peak	Time	Peak	Time	Peak	Time	Peak
50	71	*545*	63	642	65	782	*53*	2247
100	118	*709*	*100*	837	102	1059	103	4368
500	570	*2519*	443	2742	*402*	3304	565	21,923
1000	1155	*4517*	876	4789	*776*	5659	1377	43,751
5000	7412	*42,067*	5436	46,040	*4808*	51,848	11,600	224,423
10,000	19,152	*68,434*	12,298	74,500	*10,218*	84,467	43,657	444,780

Time and Peak space usage for BWT construction

- Time: seconds
- Peak space in MBs
- **`simplebwt`**: linear time suffix array construction algorithm

Experimental results

Pizza&Chili corpus

File	$w = 6, p = 20$		$w = 8, p = 50$		$w = 10, p = 100$		`simplebwt`	
	Time	Peak	Time	Peak	Time	Peak	Time	Peak
`cere`	90	603	79	*559*	*74*	801	90	3962
`einstein.en.txt`	53	196	40	88	*35*	*53*	97	4016
`influenza`	*27*	*166*	27	284	33	435	30	1331
`kernel`	43	170	29	*143*	*25*	144	50	2216
`world_leaders`	7	*50*	7	74	*7*	98	7	405

Time and Peak space usage for BWT construction

- Time: seconds
- Peak space in MBs
- **`simplebwt`**: linear time suffix array construction algorithm

Related technique

- String Synchronizing Sets were introduced by Kempa and Kociumaka in STOC 19 for sublinear time BWT construction
- Given $|T|=n$, $\tau < n/2$, a set $S \subset [1 .. n-2\tau+1]$, is a τ-synchronizing set if
  - $T[i,i+2\tau) = T[j,j+2\tau)$ then $i \in S \Leftrightarrow j \in S$
  - $S \cap [i,i+\tau) = \phi \Leftrightarrow T[i,i+3\tau-2)$ has period $< \tau/3$
- Good worst case, not practical

Further work

- We already know how to compute (a sampling of) the Suffix Array
- We are implementing the computation of (a sampling of) the Longest Common Prefix array
- We plan to experiment with multiplication algorithms involving a grammar compressed matrix