Succinct Data Structures for NLP-at-Scale

Matthias Petri Trevor Cohn

Computing and Information Systems
The University of Melbourne, Australia
first.last@unimelb.edu.au

November 20, 2016

Who are we?

Trevor Cohn, University of Melbourne

- Probabilistic machine learning for structured problems in language: NP Bayes, Deep learning, etc.
- Applications to machine translation, social media, parsing, summarisation, multilingual transfer.

Matthias Petri, University of Melbourne

- Data Compression, Succinct Data Structures, Text Indexing, Compressed Text Indexes, Algorithmic Engineering, Terabyte scale text processing
- Machine Translation, Information Retrieval, Bioinformatics

Who are we?

Tutorial based partly on research [Shareghi et al., 2015, Shareghi et al., 2016b] with collaborators at Monash University:

Ehsan Shareghi

Gholamreza Haffari

Outline

1. Introduction and Motivation (15 Minutes)
2. Basic Technologies and Notation (20 Minutes)
3. Index based Pattern Matching (20 Minutes)

   Break (20 Minutes)

4. Pattern Matching using Compressed Indexes (40 Minutes)
5. Applications to NLP (30 Minutes)

Introduction and Motivation (15 Mins)

What is it?

- Data structures and algorithms for working with large data sets
- Desiderata
  - miminise space requirement
  - maintaining efficient searchability
- Classes of compression do just this! Near-optimal compression, with minor effect on runtime
- E.g., bitvector and integer compression, wavelet trees, compressed suffix array, compressed suffix trees

Why do we need it?

- Era of 'big data': text corpora are often 100s of gigabytes or terabytes in size (e.g., CommonCrawl, Twitter)
- Even simple algorithms like counting n-grams become difficult
- One solution is to use distributed computing, however can be very inefficient
- Succinct data structures provide a compelling alternative, providing compression and efficient access
- Complex algorithms become possible in memory, rather than requiring cluster and disk access

Why do we need it?

- Era of 'big data': text corpora are often 100s of gigabytes or terabytes in size (e.g., CommonCrawl, Twitter)
- Even simple algorithms like counting n-grams become difficult
- One solution is to use distributed computing, however can be very inefficient
- Succinct data structures provide a compelling alternative, providing compression and efficient access
- Complex algorithms become possible in memory, rather than requiring cluster and disk access

E.g., Infinite order language model possible, with runtime similar to current fixed order models, and lower space requirement.

Who uses it and where is it used?

Surprisingly few applications in NLP

- Bioinformatics, Genome assembly
- Information Retrieval, Graph Search (Facebook)
- Search Engine Auto-complete
- Trajectory compression and retrieval
- XML storage and retrieval (xpath queries)
- Geo-spartial databases
- ...

Basic Technologies and Notation (20 Mins)

1. Bitvectors
2. Rank and Select
3. Succinct Tree Representations
4. Variable Size Integers

Basic Building blocks: the bitvector

Definition

A bitvector (or bit array) B of length n compactly stores n binary numbers using n bits.

Example

	0	1	2	3	4	5	6	7	8	9	10	11
B	1	1	0	0	1	1	0	1	0	1	1	0

$B[0] = 1$, $B[1] = 1$, $B[2] = 0$, $B[n-1] = B[11] = 0$ etc.

Bitvector operations

Access and Set

$B[0] = 1$, $B[0] = B[1]$

Logical Operations

A OR B, A AND B, A XOR B

Advanced Operations

POPCOUNT(B): Number of one bits set
MSB_SET(B): Most significant bit set
LSB_SET(B): Least significant bit set

Operation RANK

Definitions

$\textsc{Rank}_1(B, j)$: How many 1's are in $B[0, j]$

$\textsc{Rank}_0(B, j)$: How many 0's are in $B[0, j]$

Example

	0	1	2	3	4	5	6	7	8	9	10	11
B	1	1	0	0	1	1	0	1	0	1	1	0

$\textsc{Rank}_1(B, 7) = 5$

$\textsc{Rank}_0(B, 7) = 8 - \textsc{Rank}_1(B, 7) = 3$

Operation $\textsc{Select}$

Definitions

$\textsc{Select}_1(B, j)$: Where is the j-th (start count at 0) 1 in B

$\textsc{Select}_0(B, j)$: Where is the j-th (start count at 0) 0 in B

Inverse of $\textsc{Rank}$: $\textsc{Rank}_1(B, \textsc{Select}_1(B, j)) = j$

Example

	0	1	2	3	4	5	6	7	8	9	10	11
B	1	1	0	0	1	1	0	1	0	1	1	0

$\textsc{Select}_1(B, 4) = 7$
$\textsc{Select}_0(B, 3) = 8$

Complexity of Operations Rank and Select

Simple and Slow

Scan the whole bitvector using $O(1)$ extra space and $O(n)$ time to answer both Rank and Select

Constant time Rank

Periodically store the absolute count up till that position explicitly. Only scan a small part of the bitvector to get the right answer. Space usage: $n + o(n)$ bits. Runtime: $O(1)$. In practice: 25% extra space.

Constant time Select

Similar to Rank but more complex as blocks are based on the number of $1/0$ observed

Compressed Bitvectors

Idea

If only few 1's or clustering present in the bitvector, we can use compression techniques to substantially reduce space usage while efficiently supporting operations Rank and Select

In Practice

Bitvector of size 1 GiB with 10% of all bits randomly set to 1:
Encodings:

- Elias-Fano ['73]: x MiB
- RRR ['02]: y MiB

Bitvectors - Practical Performance

How fast are RANK and SELECT in practice? Experiment: Cost per operation averaged over 1M executions: (code)

Uncompressed:

BV Size	Access	Rank	Select	Space
1MB	3ns	4ns	47ns	127%
10MB	10ns	14ns	85ns	126%
1GB	26ns	36ns	303ns	126%
10GB	78ns	98ns	372ns	126%

Compressed:

BV Size	Access	Rank	Select	Space
1MB	68ns	65ns	49ns	33%
10MB	99ns	88ns	58ns	30%
1GB	292ns	275ns	219ns	32%
10GB	466ns	424ns	336ns	30%

Using Rank and Select

- Basic building block of many compressed / succinct data structures
- Different implementations provide a variety of time and space trade-offs
- Implemented an ready to use in SDSL and many others:
  - http://github.com/simongog/sdsl-lite
  - http://github.com/facebook/folly
  - http://sux.di.unimi.it
  - http://github.com/ot/succinct
- Used in practice! For example: Facebook Graph search (Unicorn)

Succinct Tree Representations

Idea

Instead of storing pointers and objects, flatten the tree structure into a bitvector and use RANK and SELECT to navigate

From

```
typedef struct {
    void* data;       // 64 bits
    node_t* left;     // 64 bits
    node_t* right;    // 64 bits
    node_t* parent;   // 64 bits
} node_t;
```

To

Bitvector + RANK + SELECT + Data (≈ 2 bits per node)

Succinct Tree Representations

Definition: Succinct Data Structure

A succinct data structure uses space "close" to the information theoretical lower bound, but still supports operations time-efficiently.

Succinct Tree Representations:

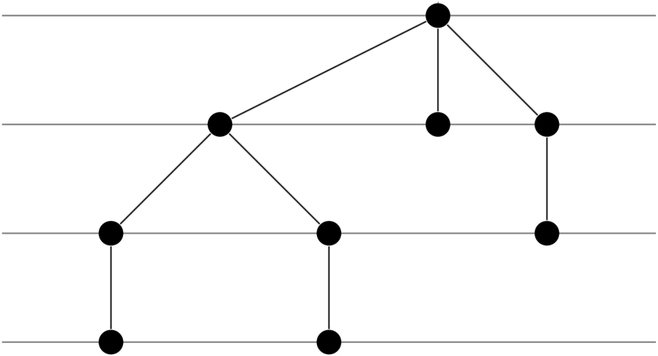
There number of unique binary trees containing n nodes is (roughly) 4^n. To differentiate between them we need at least $log_2(4^n) = 2n$ bits. Thus, a succinct tree representations should require $2n$ bits (plus a bit more).

LOUDS level order unary degree sequence

LOUDS

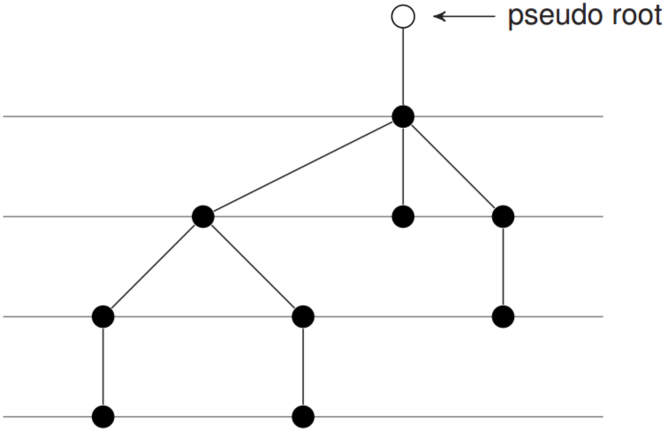
A succinct representation of a rooted, ordered tree containing nodes with arbitrary degree [Jacobson'89]

Example:



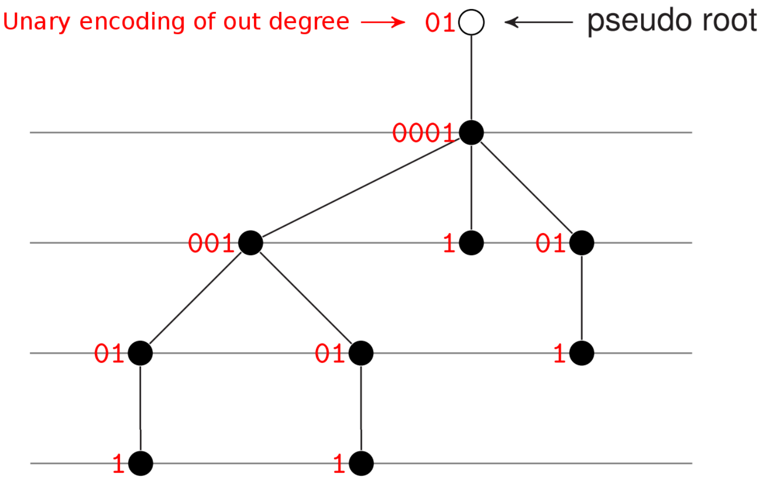
LOUDS Step 1

Add Pseudo Root:



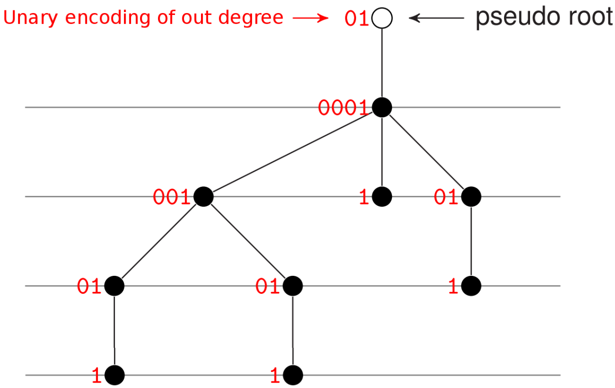
LOUDS Step 2

For each node unary encode the number of children:



LOUDS Step 3

Write out unary encodings in level order:



LOUDS sequence $L = 010001001101010111$

LOUDS Nodes

- Each node (except the pseudo root) is represented twice
  - Once as "0" in the child list of its parent
  - Once as the terminal ("1") in its child list
- Represent node v by the index of its corresponding "0"
- I.e. root corresponds to "0"
- A total of $2n$ bits are used to represent the tree shape!

LOUDS Navigation

Use RANK and SELECT to navigate the tree in constant time

Examples:

Compute node degree

```
int node_degree(int v) {
  if is_leaf(v) return 0
  id = Rank_0(L, v)
  return Select_1(L, id + 2)
    −Select_1(L, id + 1) − 1
}
```

Return the i-th child of node v

```
int child(int v, i) {
  if i > node_degree(v)
    return −1
  id = Rank_0(L, v)
  return Select_1(L, id + 1) + i
}
```

Complete construction, load, storage and navigation code of LOUDS is only 200 lines of C++ code.

Variable Size Integers

- Using 32 or 64 bit integers to store mostly small numbers is wasteful
- Many efficient encoding schemes exist to reduce space usage

Variable Byte Compression

Idea

Use variable number of bytes to represent integers. Each byte contains 7 bits "payload" and one continuation bit.

Examples

Number	Encoding	
824	00000110	10111000
5	10000101	

Storage Cost

Number Range	Number of Bytes
0 – 127	1
128 – 16383	2
16384 – 2097151	3

Variable Byte Compression - Algorithm

Encoding

function ENCODE(x)
    while $x >= 128$ **do**
        WRITE($x \mod 128$)
        $x = x \div 128$
    end while
    WRITE($x + 128$)
end function

Decoding

function DECODE(bytes)
    $x = 0$
    $y =$READBYTE(bytes)
    while $y < 128$ **do**
        $x = 128 \times x + y$
        $y =$READBYTE(bytes)
    end while
    $x = 128 \times x + (y - 128)$
    return x
end function

Variable Sized Integer Sequences

Problem

Sequences of vbyte encoded numbers can not be accessed at arbitrary positions

Solution: Directly addressable variable-length codes (DAC)

Separate the indicator bits into a bitvector and use $\textsc{Rank}$ and $\textsc{Select}$ to access integers in $O(1)$ time. [Brisboa et al.'09]

DAC - Concept

Sample vbyte encoded sequence of integers:

01010101	11110111	11000111	00110110	01110110	10000100	11101011	10000110	01101011	10000001	10000000	10001000

DAC restructuring of the vbyte encoded sequence of integers:

01010101	11000111	00110110	11101011	10000110	01101011	10000000	10001000

11110111	01110110	10000001

10000100

Separate the indicator bits:

01011011	1010101	1000111	0110110	1101011	0000110	1101011	0000000	0001000
101	1110111	1110110	0000001					
1	0000100							

DAC - Access

01011011	1010101	1000111	0110110	1101011	0000110	1101011	0000000	0001000
101	1110111	1110110	0000001					
1	0000100							

Accessing element A[5]:

- Access indicator bit of the first level at position 5: $I1[5] = 0$
- 0 in the indicator bit implies the number uses at least 2 bytes
- Perform $Rank_0(I1, 5) = 3$ to determine the number of integers in $A[0, 5]$ with at least two bytes
- Access $I2[3 - 1] = 1$ to determine that number $A[5]$ has two bytes.
- Access payloads and recover number in $O(1)$ time.

Practical Exercise

Index based Pattern Matching (20 Mins)

Pattern Matching

Definition

Given a text T of size n, find all occurrences (or just count) of pattern P of length m.

Online Pattern Matching

Preprocess P, scan T. Examples: KMP, Boyer-Moore, BMH etc. $O(n+m)$ search time.

Offline Pattern Matching

Preprocess T, Build Index. Examples: Inverted Index, Suffix Tree, Suffix Array. $O(m)$ search time.

Suffix Tree (Weiner'73)

- Data structure capable of processing T in $O(n)$ time and answering search queries in $O(n)$ space and $O(m)$ time. Optimal from a theoretical perspective.
- All suffixes of T into a trie (a tree with edge labels)
- Contains n leaf nodes corresponding to the n suffixes of T
- Search for a pattern P is performed by finding the subtree corresponding to all suffixes prefixed by P

Suffix Tree - Example

$$T = \texttt{abracadabracarab\$}$$

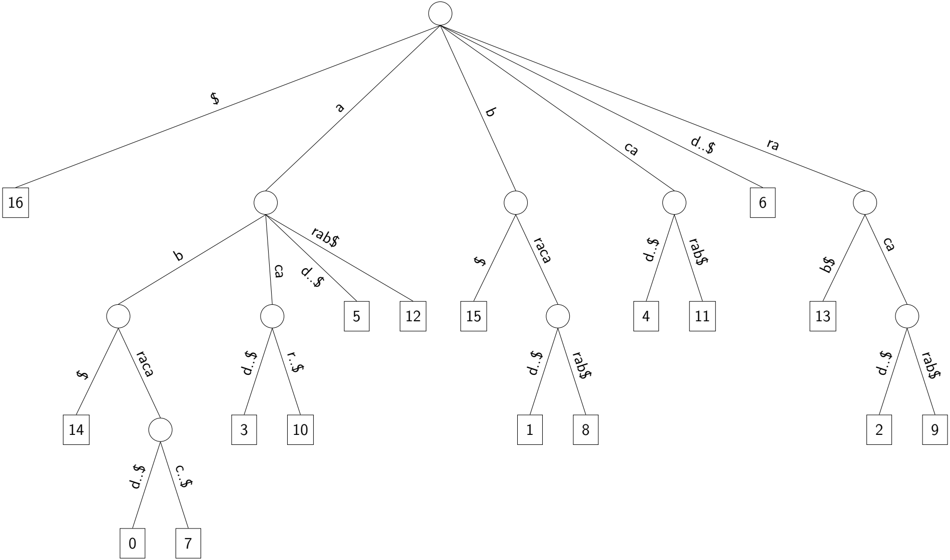
Suffix Tree - Example

$$T = \texttt{abracadabracarab\$}$$

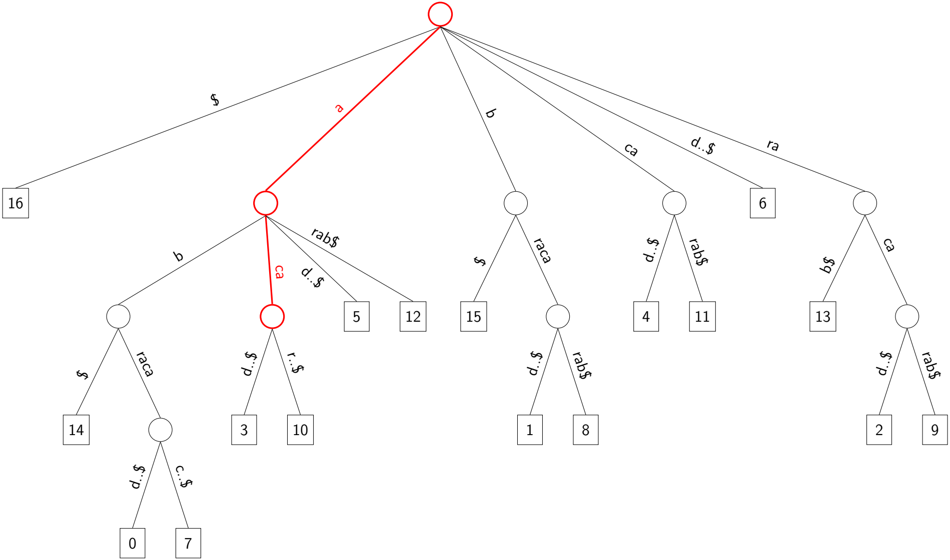
Suffixes:

0	abracadabracarab$
1	bracadabracarab$
2	racadabracarab$
3	acadabracarab$
4	cadabracarab$
5	adabracarab$
6	dabracarab$
7	abracarab$
8	bracarab$
9	racarab$
10	acarab$
11	carab$
12	arab$
13	rab$
14	ab$
15	b$
16	$

Suffix Tree - Example



Suffix Tree - Search for "aca"



Suffix Tree - Problems

- Space usage in practice is large. $20 - 40$ times n for highly optimized implementations.
- Only useable for small datasets.

Suffix Arrays (Manber'89)

- Reduce space of Suffix Tree by only storing the n leaf pointers into the text
- Requires $n \log n$ bits for the pointers plus T to perform search
- In practice $5 - 9$n bytes for character alphabets
- Search for P using binary search

Suffix Arrays - Example

$$T = \texttt{abracadabracarab\$}$$

Suffix Arrays - Example

$$T = \texttt{abracadabracarab\$}$$

Suffixes:

0	abracadabracarab$
1	bracadabracarab$
2	racadabracarab$
3	acadabracarab$
4	cadabracarab$
5	adabracarab$
6	dabracarab$
7	abracarab$
8	bracarab$
9	racarab$
10	acarab$
11	carab$
12	arab$
13	rab$
14	ab$
15	b$
16	$

Suffix Arrays - Example

$$T = \texttt{abracadabracarab\$}$$

Sorted Suffixes:

16	\$
14	ab\$
0	abracadabracarab\$
7	abracarab\$
3	acadabracarab\$
10	acarab\$
5	adabracarab\$
12	arab\$
15	b\$
1	bracadabracarab\$
8	bracarab\$
4	cadabracarab\$
11	carab\$
6	dabracarab\$
13	rab\$
2	racadabracarab\$
9	racarab\$

Suffix Arrays - Example

$$T = \texttt{abracadabracarab\$}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays - Search

$T =$abracadabracarab\$, $P =$abr

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays - Search

$T=$abracadabracarab\$, $P=$abr

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	\$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays - Search

$T =$abracadabracarab\$, $P =$abr

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays - Search

$T =$ `abracadabracarab$`, $P =$ `abr`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	b

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays - Search

$$T = \texttt{abracadabracarab\$},$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	b

		lb	rb													
		↓	↓													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9

Suffix Arrays / Trees - Resource Consumption

In practice:

- Suffix Trees requires $\approx 20n$ bytes of space (for efficient implementations)
- Suffix Arrays require $5 - 9n$ bytes of space
- Comparable search performance

Example: 5GB English text requires 45GB for a character level suffix array index and up to 200GB for suffix trees

Suffix Arrays / Trees - Construction

In theory: Both can be constructed in optimal $O(n)$ time

In practice:

- Suffix Trees and Suffix Arrays construction can be parallelized
- Most efficient suffix array construction algorithm in practice are note $O(n)$
- Efficient semi-external memory construction algorithms exist
- Parallel suffix array construction algorithms can index 20MiB/s (24 threads) in-memory and 4MiB/s in external memory
- Suffix Arrays of terabyte scale text collection can be constructed. Practical!
- Word-level Suffix Array construction also possible.

Dilemma

- There is lots of work out there which proposes solutions for different problems based on suffix trees
- Suffix trees (and to a certain extend suffix arrays) are not really applicable for large scale problems
- However, large scale suffix arrays can be constructed efficiently without requiring large amounts of memory

Solutions:

- External or Semi-External memory representation of suffix trees / arrays

Dilemma

- There is lots of work out there which proposes solutions for different problems based on suffix trees
- Suffix trees (and to a certain extend suffix arrays) are not really applicable for large scale problems
- However, large scale suffix arrays can be constructed efficiently without requiring large amounts of memory

Solutions:

- External or Semi-External memory representation of suffix trees / arrays
- Compression?

External / Semi-External Suffix Indexes

String-B Tree

- Cache-Oblivious
- Complicated
- Not implemented anywhere (not practical?)

Compressed Suffix Arrays and Trees

Idea

Utilize data compression techniques to substantially reduce the space of suffix arrays/trees while retaining their functionality

Compressed Suffix Arrays (CSA):

- Use space equivalent to the compressed size of the input text. Not 4-8 times more! Example: 1GB English text compressed to roughly 300MB using gzip. CSA uses roughly 300MB (sometimes less)!
- Provide more functionality than regular suffix arrays
- Implicitly contain the original text, no need to retain it. Not needed for query processing
- Similar search efficiency than regular suffix arrays.
- Used to index terabytes of data on a reasonably powerful machine!

CSA and CST in practice using SDSL

```
#include "sdsl/suffix_arrays.hpp"
#include <iostream>

int main(int argc, char** argv) {
    std::string input_file = argv[1];
    std::string out_file = argv[2];
    sdsl::csa_wt<> csa;
    sdsl::construct(csa, input_file, 1);
    std::cout << "CSA size = "
        << sdsl::size_in_megabytes(csa) << std::endl;
    sdsl::store_to_file(csa, out_file);
}
```

How does it work? Find out after the break!

Break Time

See you back here in 20 minutes!

Compressed Indexes (40 Mins)

Compressed Suffix Arrays - Overview

Two practical approaches developed independently:

- CSA-SADA: Proposed by Grossi and Vitter in 2000. Practical refinements by Sadakane also in 2000.
- CSA-WT: Also referred to as the FM-Index. Proposed by Ferragina and Manzini in 2000.

Many practical (and theoretical) improvements to compression, query speed since then. Efficient implementations available in SDSL: `csa_sada<>` and `csa_wt<>`.

For now, we focus on CSA-WT.

CSA-WT or the FM-Index

- Utilizes the Burrows-Wheeler Transform (BWT) used in compression tools such as `bzip2`
- Requires RANK and SELECT on non-binary alphabets
- Heavily utilize compressed bitvector representations
- Theoretical bound on space usage related to compressibility (entropy) of the input text

The Burrows-Wheeler Transform (BWT)

- Reversible Text Permutation
- Initially proposed by Burrows and Wheeler as a compression tool. The BWT is more compressible than the original text!
- Defined as $BWT[i] = T[SA[i] - 1 \mod n]$
- In words: $BWT[i]$ is the symbol preceding suffix $SA[i]$ in T

Why does it work? How is it related to searching?

BWT - Example

$$T = \texttt{abracadabracarab\$}$$

BWT - Example

$$T = \texttt{abracadabracarab\$}$$

0	abracadabracarab$
1	bracadabracarab$
2	racadabracarab$
3	acadabracarab$
4	cadabracarab$
5	adabracarab$
6	dabracarab$
7	abracarab$
8	bracarab$
9	racarab$
10	acarab$
11	carab$
12	arab$
13	rab$
14	ab$
15	b$
16	$

BWT - Example

$T =$abracadabracarab$

Suffix Array

16	$
14	ab$
0	abracadabracarab$
7	abracarab$
3	acadabracarab$
10	acarab$
5	adabracarab$
12	arab$
15	b$
1	bracadabracarab$
8	bracarab$
4	cadabracarab$
11	carab$
6	dabracarab$
13	rab$
2	racadabracarab$
9	racarab$

BWT - Example

$$T = \texttt{abracadabracarab\$}$$

Suffix Array

BWT

16	\$	b
14	ab\$	r
0	abracadabracarab	\$
7	abracarab\$	d
3	acadabracarab\$	r
10	acarab\$	r
5	adabracarab\$	c
12	arab\$	c
15	b\$	a
1	bracadabracarab\$	a
8	bracarab\$	a
4	cadabracarab\$	a
11	carab\$	a
6	dabracarab\$	a
13	rab\$	a
2	racadabracarab\$	b
9	racarab\$	b

BWT - Example

$$T = \texttt{abracadabracarab\$}$$

$	b	
a	r	
a	$	
a	d	BWT
a	r	
a	r	
a	c	
a	c	
b	a	
b	a	
b	a	
c	a	
c	a	
d	a	
r	a	
r	b	
r	b	

BWT - Reconstructing T from BWT

$$T =$$

b
r
$
d
r
r
c
c
a
a
a
a
a
a
a
b
b

BWT - Reconstructing T from BWT

$T =$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

1. Sort BWT to retrieve first column F

BWT - Reconstructing T from BWT

$T =$ $

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

2. Find last symbol $\$$ in F at position 0 and write to output

BWT - Reconstructing T from BWT

$T =$ b$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

2. Symbol preceding $ in T is $BWT[0] = b$. Write to output

BWT - Reconstructing T from BWT

$T =$ b$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

3. As there are no b before $BWT[0]$, we know that this b corresponds to the first b in F at pos $F[8]$.

BWT - Reconstructing T from BWT

$T =$ ab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

4. The symbol preceding $F[8]$ is $BWT[8] = a$. Output!

BWT - Reconstructing T from BWT

$T =$ ab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

5. Map that a back to F at position $F[1]$

BWT - Reconstructing T from BWT

$T =$ rab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

6. Output $BWT[1] = r$ and map r to $F[14]$

BWT - Reconstructing T from BWT

$T =$ arab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

7. Output $BWT[14] = a$ and map a to $F[7]$

BWT - Reconstructing T from BWT

$T =$ arab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

Why does $BWT[14] = a$ map to $F[7]$?

BWT - Reconstructing T from BWT

All a preceding $BWT[14] = a$ preceed suffixes smaller than $SA[14]$.

$T =$ arab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

BWT - Reconstructing T from BWT

Thus, among the suffixes starting with a, the one preceding $SA[14]$ must be the last one.

$T =$ arab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

BWT - Reconstructing T from BWT

$T =$abracadabracarab$

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

Searching using the BWT

$T =$abracadabracarab\$, $P =$abr

0	\$	b
1	a	r
2	a	\$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

Searching using the BWT

$T =$abracadabracarab\$, $P =$abr

0	\$	b
1	a	r
2	a	\$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

Search backwards, start by finding the r interval in F

Searching using the BWT

T =abracadabracarab\$, P =abr

0	\$	b
1	a	r
2	a	\$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
→ 14	r	a
15	r	b
→ 16	r	b

Search backwards, start by finding the r interval in F

Searching using the BWT

T =abracadabracarab\$, P =abr

0	\$	b
1	a	r
2	a	\$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
→ 14	r	a
15	r	b
→ 16	r	b

How many b's are the r interval in $BWT[14, 16]$? 2

Searching using the BWT

T =abracadabracarab$, P =abr

0	$	b
1	a	r
2	a	$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
→ 14	r	a
15	r	b
→ 16	r	b

How many suffixes starting with b are smaller than those 2?

1 at $BWT[0]$

Searching using the BWT

$T =$abracadabracarab\$, $P =$abr

Thus, all suffixes starting with br are in $SA[9, 10]$.

	0	$	b
	1	a	r
	2	a	$
	3	a	d
	4	a	r
	5	a	r
	6	a	c
	7	a	c
	8	b	a
→	9	b	a
→	10	b	a
	11	c	a
	12	c	a
	13	d	a
	14	r	a
	15	r	b
	16	r	b

Searching using the BWT

$T =$abracadabracarab\$, $P =$abr

	0	\$	b
	1	a	r
	2	a	\$
	3	a	d
	4	a	r
	5	a	r
	6	a	c
	7	a	c
	8	b	a
→	9	b	a
→	10	b	a
	11	c	a
	12	c	a
	13	d	a
	14	r	a
	15	r	b
	16	r	b

How many of the suffixes starting with br are preceded by a? 2

Searching using the BWT

$T =$abracadabracarab\$, $P =$abr

0	\$	b
1	a	r
2	a	\$
3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
→ 9	b	a
→ 10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

How many of the suffixes smaller than br are preceded by a? 1

Searching using the BWT

T =abracadabracarab$, P =abr

0	$	b
1	a	r
→ 2	a	$
→ 3	a	d
4	a	r
5	a	r
6	a	c
7	a	c
8	b	a
9	b	a
10	b	a
11	c	a
12	c	a
13	d	a
14	r	a
15	r	b
16	r	b

There are 2 occurrences of abr in T corresponding to suffixes $SA[2, 3]$

Searching using the BWT

- We only require F and BWT to search and recover T
- We only had to count the number of times a symbol s occurs within an interval, and before that interval $BWT[i, j]$
- Equivalent to $Rank_s(BWT, i)$ and $Rank_s(BWT, j)$
- Need to perform $Rank$ on non-binary alphabets efficiently

Wavelet Trees - Overview

- Data structure to perform $Rank$ and $Select$ on non-binary alphabets of size σ in $O(\log_2 \sigma)$ time
- Decompose non-binary $Rank$ operations into binary $Rank$'s via tree decomposition
- Space usage $n \log \sigma + o(n \log \sigma)$ bits. Same as original sequence + Rank + Select overhead

Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b

Symbol	Codeword
$	00
a	010
b	011
c	10
d	110
r	111

Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9	10
b	$	a	a	a	a	a	a	a	b	b
1	0	1	1	1	1	1	1	1	1	1

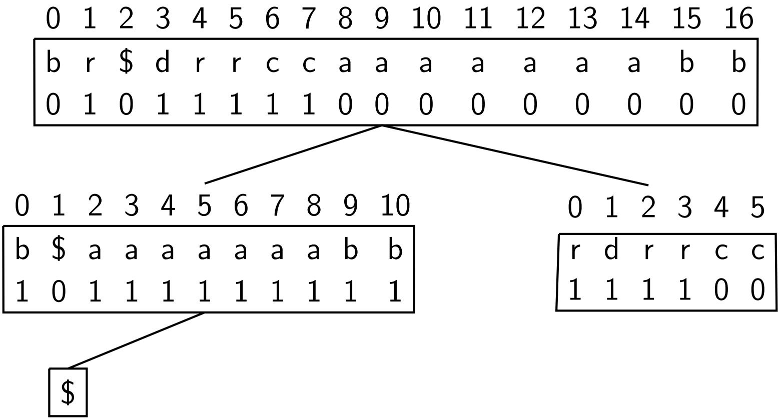
Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

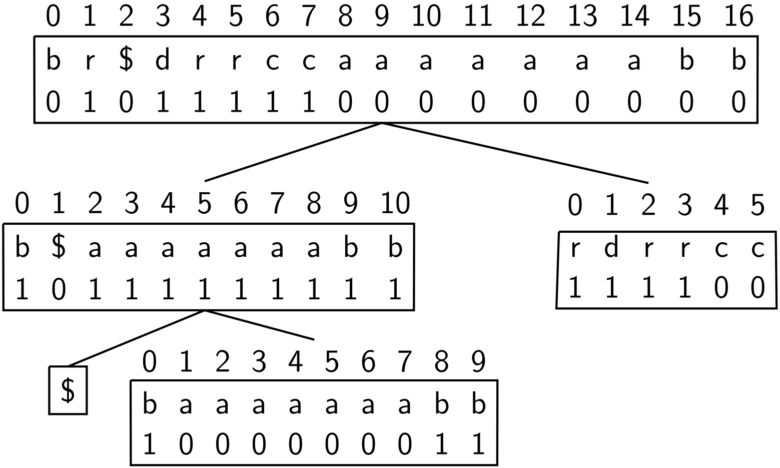
0	1	2	3	4	5	6	7	8	9	10
b	$	a	a	a	a	a	a	a	b	b
1	0	1	1	1	1	1	1	1	1	1

0	1	2	3	4	5
r	d	r	r	c	c
1	1	1	1	0	0

Wavelet Trees - Example



Wavelet Trees - Example



Wavelet Trees - Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

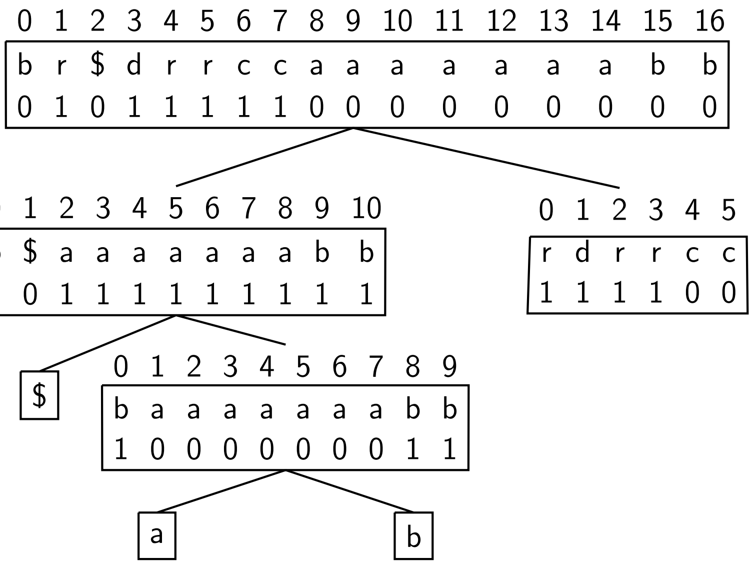
0	1	2	3	4	5	6	7	8	9	10
b	$	a	a	a	a	a	a	a	b	b
1	0	1	1	1	1	1	1	1	1	1

0	1	2	3	4	5
r	d	r	r	c	c
1	1	1	1	0	0

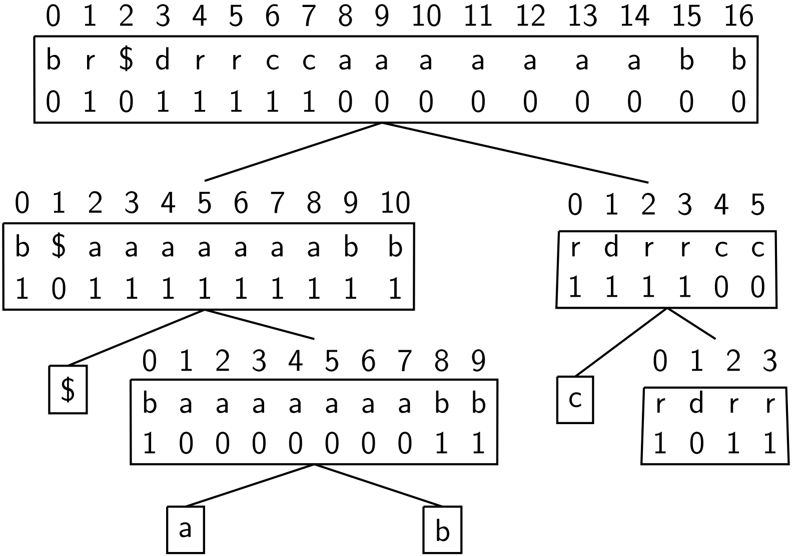
$

0	1	2	3	4	5	6	7	8	9
b	a	a	a	a	a	a	a	b	b
1	0	0	0	0	0	0	0	1	1

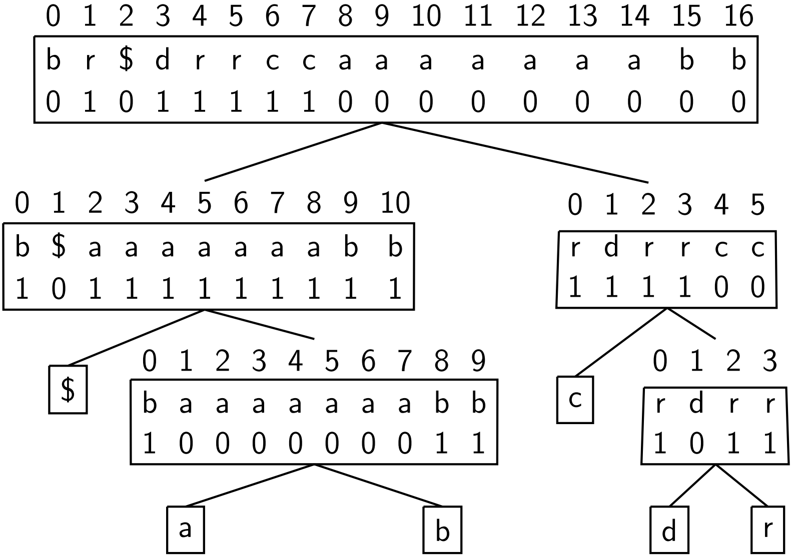
Wavelet Trees - Example



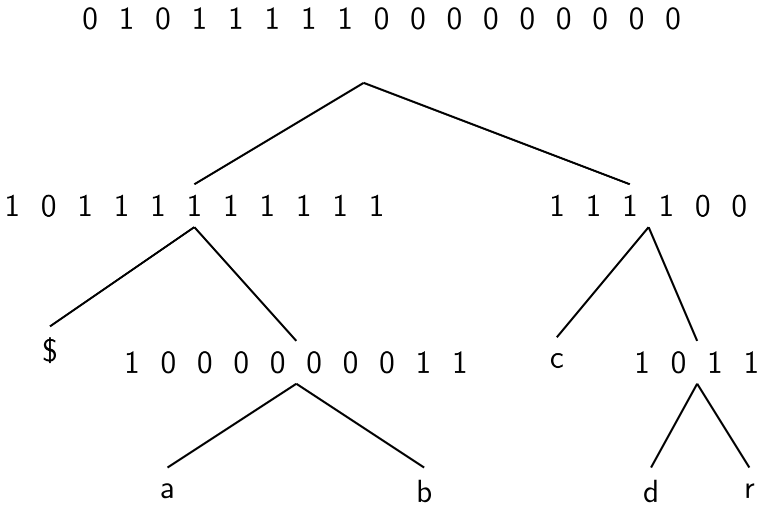
Wavelet Trees - Example



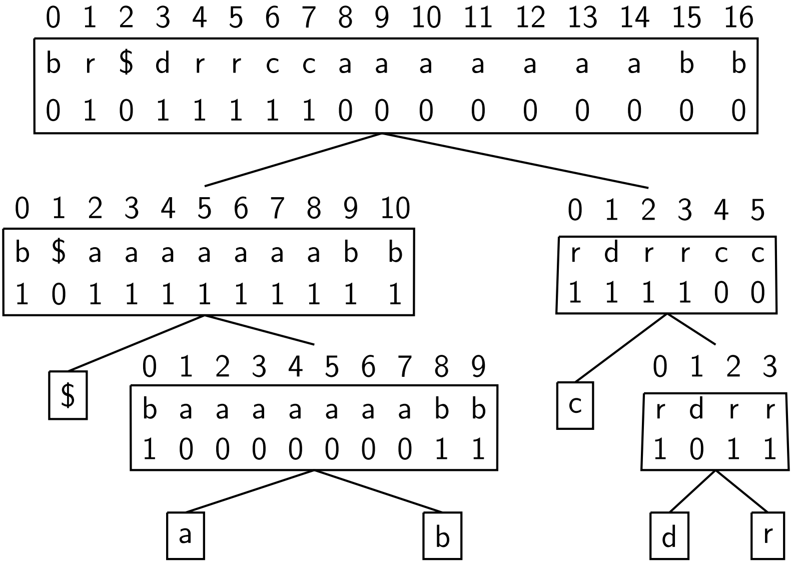
Wavelet Trees - Example



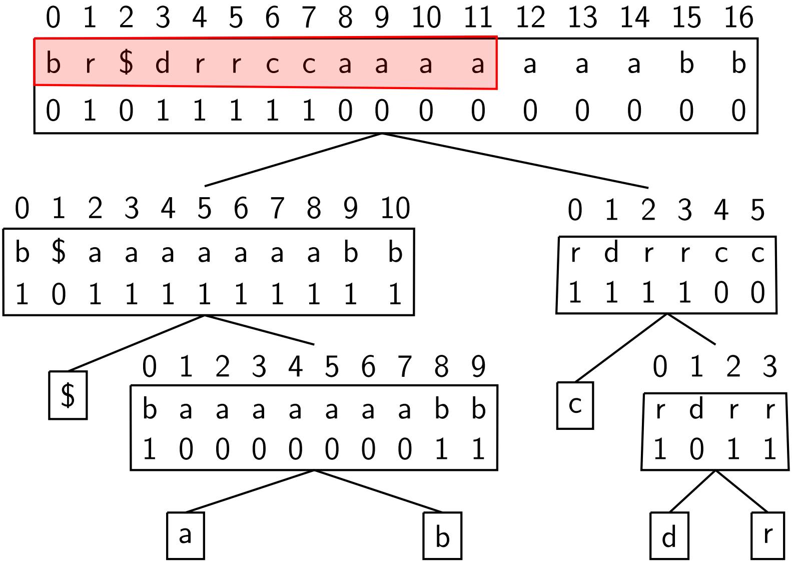
Wavelet Trees - What is actually stored



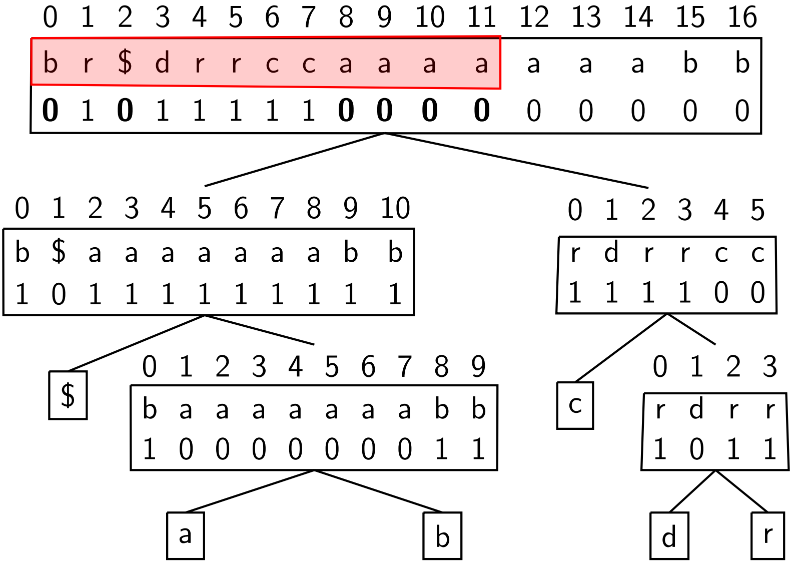
Wavelet Trees - Performing $Rank_a(BWT, 11)$



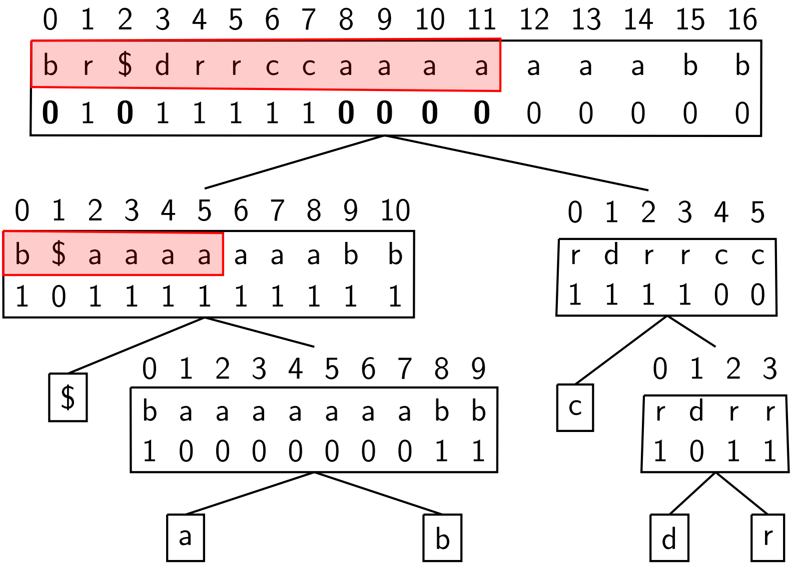
Wavelet Trees - Performing $Rank_a(BWT, 11)$



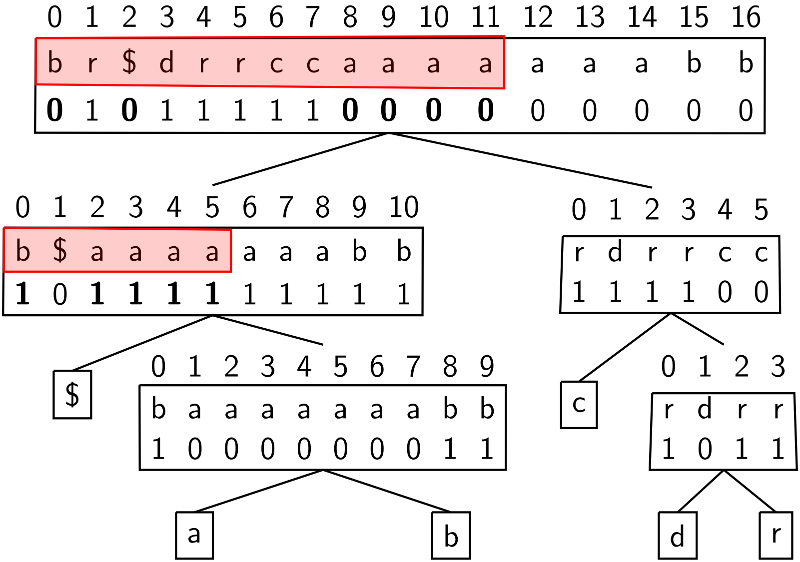
Wavelet Trees - Performing $Rank_a(BWT, 11)$



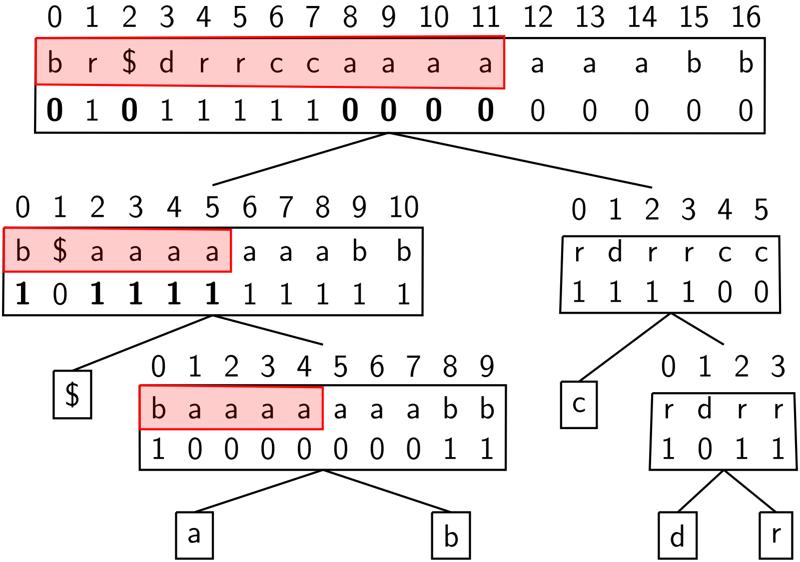
Wavelet Trees - Performing $Rank_a(BWT, 11)$



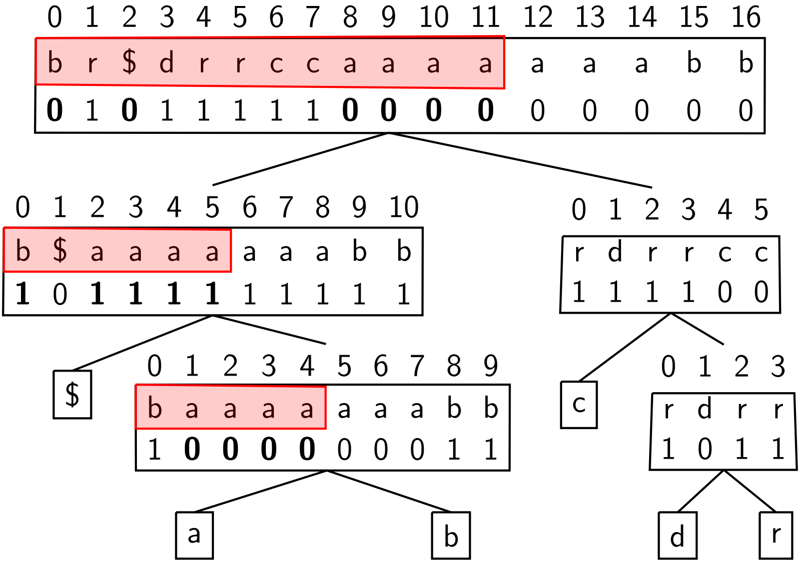
Wavelet Trees - Performing $Rank_a(BWT, 11)$



Wavelet Trees - Performing $Rank_a(BWT, 11)$



Wavelet Trees - Performing $Rank_a(BWT, 11)$



Wavelet Trees - Space Usage

Currently: $n \log \sigma + o(n \log \sigma)$ bits. Still larger than the original text!

How can we do better?

- Compressed bitvectors

Wavelet Trees - Space Usage

Currently: $n \log \sigma + o(n \log \sigma)$ bits. Still larger than the original text!

How can we do better?

- Picking the codewords for each symbol smarter!

Wavelet Trees - Space Usage

Currently

Symbol	Freq	Codeword
\$	1	00
a	7	010
b	3	011
c	2	10
d	1	110
r	3	111

Bits per symbol: 2.82

Huffman Shape:

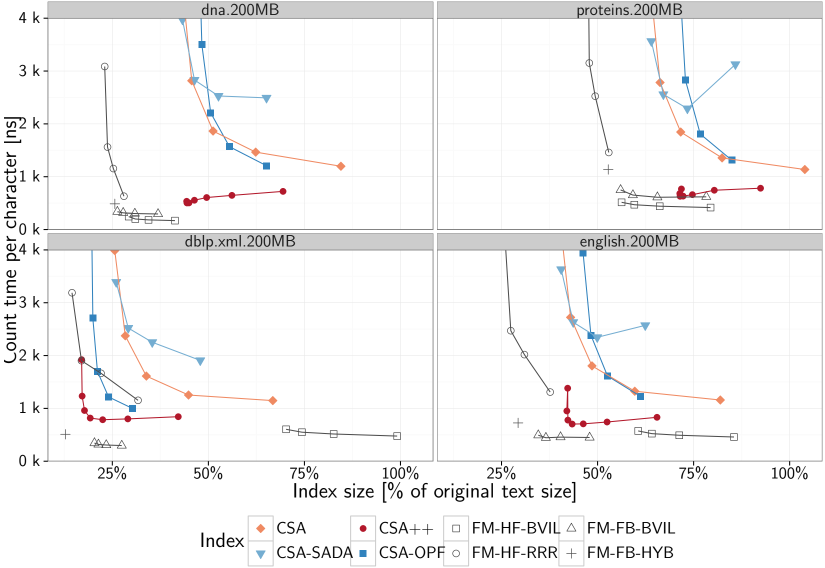
Symbol	Freq	Codeword
\$	1	1100
a	7	0
b	3	101
c	2	111
d	1	1101
r	3	100

Bits per symbol: 2.29

Space usage of Huffman shaped wavelet tree:
$H_0(T)n + o(H_0(T)n)$ bits.

Even better: Huffman shape + compressed bitvectors

CSA-WT - Space Usage in practice



CSA-WT - Trade-offs in SDSL

```
#include "sdsl/suffix_arrays.hpp"
#include "sdsl/bit_vectors.hpp"
#include "sdsl/wavelet_trees.hpp"

int main(int argc, char** argv) {
    std::string input_file = argv[1];
    // use a compressed bitvector
    using bv_type = sdsl::hyb_vector<>;
    // use a huffman shaped wavelet tree
    using wt_type = sdsl::wt_huff<bv_type>;
    // use a wt based CSA
    using csa_type = sdsl::csa_wt<wt_type>;
    csa_type csa;
    sdsl::construct(csa, input_file, 1);
    sdsl::store_to_file(csa, out_file);
}
```

CSA-WT - Trade-offs in SDSL

```
// use a regular bitvector
using bv_type = sdsl::bit_vector;
// 5% overhead rank structure
using rank_type = sdsl::rank_support_v5<1>;
// don't need select so we just use
// scanning which is O(n)
using select1_type = sdsl::select_support_scan<1>;
using select0_type = sdsl::select_support_scan<0>;
// use a huffman shaped wavelet tree
using wt_type  = sdsl::wt_huff<bv_type,
                               rank_type,
                               select1_type,
                               select0_type>;
using csa_type = sdsl::csa_wt<wt_type>;
csa_type csa;
sdsl::construct(csa,input_file,1);
sdsl::store_to_file(csa,out_file);
```

CSA-WT - Searching

```cpp
int main(int argc,char** argv) {
    std::string input_file = argv[1];
    sdsl::csa_wt<> csa;
    sdsl::construct(csa,input_file,1);

    std::string pattern = "abr";
    auto nocc = sdsl::count(csa,pattern);
    auto occs = sdsl::locate(csa,pattern);
    for(auto& occ : occs) {
        std::cout << "found at pos "
                  << occ << std::endl;
    }
    auto snippet = sdsl::extract(csa,5,12);
    std::cout << "snippet = '"
              << snippet << "'" << std::endl;
}
```

CSA-WT - Searching - UTF-8

```
sdsl::csa_wt<> csa; //
sdsl::construct(csa, "this-file.cpp", 1);
std::cout << "count(\"\") : "
     << sdsl::count(csa, "") << endl;
auto occs = sdsl::locate(csa, "\n");
sort(occs.begin(), occs.end());
auto max_line_length = occs[0];
for (size_t i=1; i < occs.size(); ++i)
    max_line_length = std::max(max_line_length,
                                   occs[i]-occs[i-1]+1);
std::cout << "max line length : "
          << max_line_length << endl;
```

CSA-WT - Searching - Words

32 bit integer words:

```
sdsl::csa_wt_int<> csa;
// file containing uint32_t ints
sdsl::construct(csa, "words.u32", 5);
std::vector<uint32_t> pattern = {532432,43433};
std::cout << "count() : "
          << sdsl::count(csa,pattern) << endl;
```

$\log_2 \sigma$ bit words in SDSL format:

```
sdsl::csa_wt_int<> csa;
// file containing a serialized sdsl::int_vector ints
sdsl::construct(csa, "words.sdsl", 0);
std::vector<uint32_t> pattern = {532432,43433};
std::cout << "count() : "
          << sdsl::count(csa,pattern) << endl;
```

CSA - Usage Resources

Tutorial:
http://simongog.github.io/assets/data/sdsl-slides/tutorial

Cheatsheet:
http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf

Examples: https://github.com/simongog/sdsl-lite/examples

Tests: https://github.com/simongog/sdsl-lite/test

Compressed Suffix Trees

- Compressed representation of a Suffix Tree
- Internally uses a CSA
- Store extra information to represent tree shape and node depth information
- Three different CST types available in SDSL

Compressed Suffix Trees - CST

- Use a succinct tree representation to store suffix tree shape
- Compress the LCP array to store node depth information

Operations:
`root`, `parent`, `first_child`, `iterators`, `sibling`, `depth`, `node_depth`, `edge`, `children`... many more!

CST - Example

```
using csa_type = sdsl::csa_wt<>;
sdsl::cst_sct3<csa_type> cst;
sdsl::construct_im(cst, "ananas", 1);
for (auto v : cst) {
    cout << cst.depth(v) << "-[" << cst.lb(v) << ","
         << cst.rb(v) << "]" << endl;
}
auto v = cst.select_leaf(2);
for (auto it = cst.begin(v); it != cst.end(v); ++it) {
    auto node = *it;
    cout << cst.depth(v) << "-[" << cst.lb(v) << ","
         << cst.rb(v) << "]" << endl;
}
v = cst.parent(cst.select_leaf(4));
for (auto it = cst.begin(v); it != cst.end(v); ++it) {
    cout << cst.depth(v) << "-[" << cst.lb(v) << ","
         << cst.rb(v) << "]" << endl;
}
```

CST - Space Usage Visualization

http://simongog.github.io/assets/data/space-vis.html

Applications to NLP (30 Mins)

Application to NLP: language modelling

Language models & succinct data structures

Count-based language models:

$$P(w_i|w_1, \ldots, w_{i-1}) \approx P^{(k)}(w_i|w_{i-k}, \ldots, w_{i-1})$$

Estimation from k-gram corpus statistics using ST/SA

- based arounds suffix arrays [Zhang and Vogel, 2006]
- and suffix trees [Kennington et al., 2012]
- practical using CSA/CST [Shareghi et al., 2016b]

In all cases, on-the-fly calculation and no cap on k required.[1]

Related, machine translation

Lookup of (dis)contiguous ‘phrases’, as part of dynamic phrase-table [Callison-Burch et al., 2005, Lopez, 2008].

[1] Caps needed on smoothing parameters [Shareghi et al., 2016a].

Faster & cheaper language model research

Commonly, store probabilities for k-grams explicitly.

Efficient storage

- tries and hash tables for fast lookup [Heafield, 2011]
- lossy data structures [Talbot and Osborne, 2007]
- storage of approximate probabilities using quantisation and pruning [Pauls and Klein, 2011]
- parallel 'distributed' algorithms [Brants et al., 2007]

Overall: fast, but limited to fixed m-gram, and intensive hardware requirements.

Language models

Definition

A language model defines probability $P(w_i|w_1,\dots,w_{i-1})$, often with a Markov assumption, i.e., $P \approx P^{(k)}(w_i|w_{i-k},\dots,w_{i-1})$.

Example: MLE for k-gram LM

$$P^{(k)}(w_i|w_{i-k}^{i-1}) = \frac{c(w_{i-k}^{i})}{c(w_{i-k}^{i-1})}$$

- using count of context, $c(w_{i-k}^{i-1})$; and
- count of full k-gram, $c(w_{i-k}^{i})$

Notation: $w_i^j \triangleq (w_i, w_{i+1}, \dots, w_j)$

Smoothed count-based language models

Interpolate or backoff from higher to lower order models

$$P^{(k)}(w_i|w_{i-k}^{i-1}) = f(w_{i-k}^{i}) + g(w_{i-k}^{i-1})P^{(k-1)}(w_i|w_{i-k+1}^{i-1})$$

terminating at unigram MLE, $P^{(1)}$.

Selecting f and g functions

interpolation f is a *discounted* function of the context and k-gram counts, reserving some mass for g

backoff only one of f or g term is non-zero, based on whether full pattern is found

Involved computation of either the discount or normalisation.

Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998)

Intuition

Not all k-grams should be treated equally $\Rightarrow$ k-grams occurring in fewer contexts should carry lower weight.

Example

Fransisco is a common unigram, but only occurs in one context, *San Franscisco*
Treat unigram *Fransisco* as having count 1.

Enacted through formulation based occurrence counts for scoring component $k < m$ grams and discount smoothing.

Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998)

$$P^{(k)}(w_i|w_{i-k}^{i-1}) = f(w_{i-k}^{i}) + g(w_{i-k}^{i-1})P^{(k-1)}(w_i|w_{i-k+1}^{i-1})$$

Highest order $k = m$

$$f(w_{i-k}^{i}) = \frac{[c(w_{i-k+1}^{i}) - D_k]^+}{c(w_{i-k+1}^{i-1})}$$

$$g(w_{i-k}^{i-1}) = \frac{D_k N_{1+}(w_{i-k-1}^{i-1}\ \bullet)}{c(w_{i-k+1}^{i-1})}$$

$0 \leq D_k < 1$ are discount constants.

Lower orders $k < m$

$$f(w_{i-k}^{i}) = \frac{[N_{1+}(\bullet\ w_{i-k+1}^{i}) - D_k]^+}{N_{1+}(\bullet\ w_{i-k+1}^{i-1}\ \bullet)}$$

$$g(w_{i-k}^{i-1}) = \frac{D_k N_{1+}(w_{i-k+1}^{i-1}\ \bullet)}{N_{1+}(\bullet\ w_{i-k+1}^{i-1}\ \bullet)}$$

Uses unique context counts, rather than counts directly.

Modified Kneser Ney

Discount component now a function of the k-gram count / occurrence count

$$D_k : [0, 1, 2, 3+] \rightarrow \mathcal{R}$$

Consequence: complication to g term!

Now must incorporate the number of k-grams with given prefix

- with count 1, $N_1(w_{i-k+1}^{i-1} \bullet)$;
- with count 2, $N_2(w_{i-k+1}^{i-1} \bullet)$; and
- with count 3 or greater, $N_{1+} - N_1 - N_2$.

Sufficient Statistics

Kneser Ney probability compution requires the following:

$$c(w_i^j) \qquad \text{basic counts}$$

$$\left.\begin{array}{c} N_{1+}(w_i^j \bullet) \\ N_{1+}(\bullet\, w_i^j) \\ N_{1+}(\bullet\, w_i^j \bullet) \\ N_1(w_i^j \bullet) \\ N_2(w_i^j \bullet) \end{array}\right\} \text{occurrence counts}$$

Other smoothing methods also require forms of occurrence counts, e.g., Good-Turing, Witten-Bell.

Construction and querying

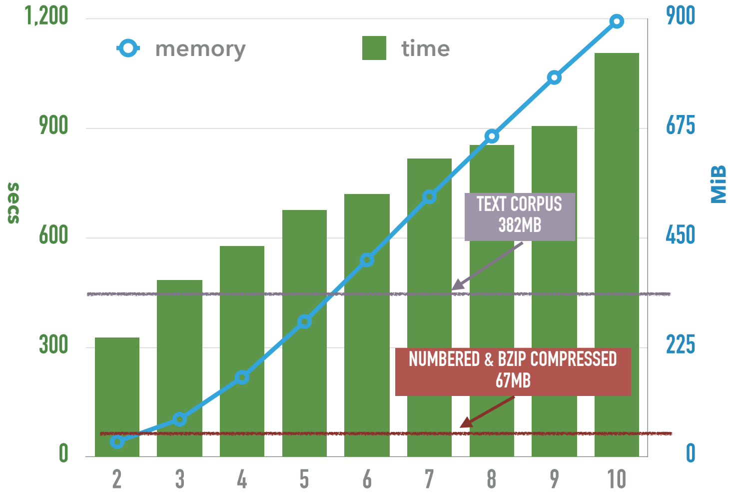
Probabilities computed ahead of time

- Calculate a static hashtable or trie mapping k-grams to their probability and backoff values.
- Big: number of possible & observed k-grams grows with k

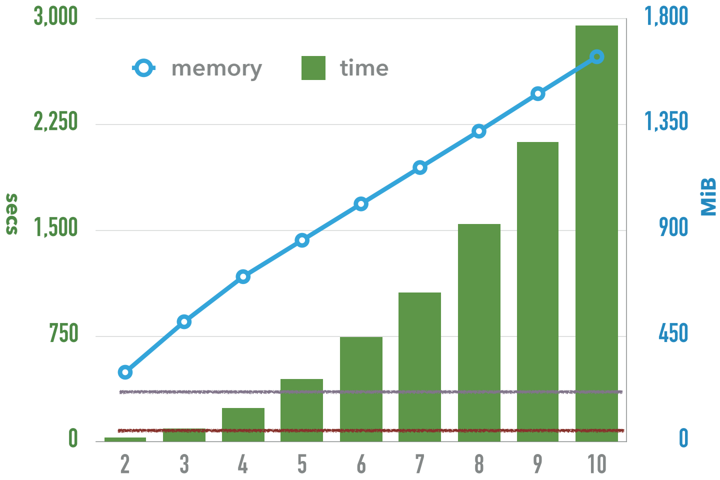
Querying

Lookup the longest matching span including the current token, and without the token. Probability computed from the full score and context backoff.

Query cost German Europarl, KenLM trie



Cost of construction German Europarl, KenLM trie



Precomputing versus on-the-fly

Precomputing approach

- Does not scale gracefully to high order m;
- Large training corpora also problematic

Can be computed directly from a CST

- CST captures unlimited order k-grams (no limit on m);
- Many (but not all) statistics cheap to retrieve
- LM probabilities computed on-the-fly

Sufficient statistics captured in suffix structures

$$T = \texttt{abracadabracarab\$}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SA_i	16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9
T_{SA_i}	$	a	a	a	a	a	a	a	b	b	b	c	c	d	r	r	r
$T_{SA_{i-1}}$	b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b

Sufficient statistics captured in suffix structures

$$T = \texttt{abracadabracarab\$}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SA_i	16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9
T_{SA_i}	$	a	a	a	a	a	a	a	b	b	b	c	c	d	r	r	r
$T_{SA_{i-1}}$	b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b

Sufficient statistics captured in suffix structures

$$T = \texttt{abracadabracarab\$}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SA_i	16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9
T_{SA_i}	\$	a	a	a	a	a	a	a	b	b	b	c	c	d	r	r	r
$T_{SA_{i-1}}$	b	r	\$	d	r	r	c	c	a	a	a	a	a	a	a	b	b

- $c(\texttt{abra}) = 2$ from CSA
  range between $lb = 3$ and $rb = 4$, inclusive

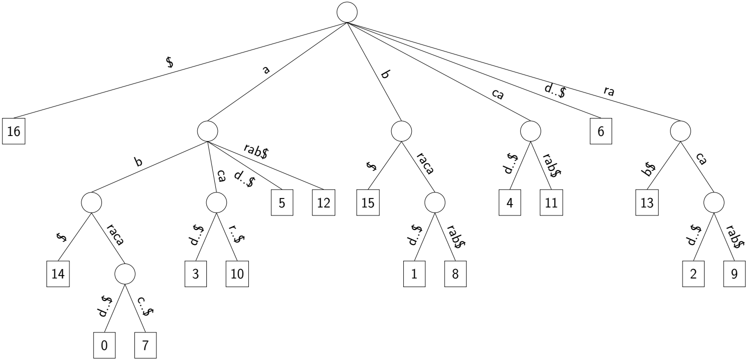
Sufficient statistics captured in suffix structures

$$T = \texttt{abracadabracarab\$}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SA_i	16	14	0	7	3	10	5	12	15	1	8	4	11	6	13	2	9
T_{SA_i}	$	a	a	a	a	a	a	a	b	b	b	c	c	d	r	r	r
$T_{SA_{i-1}}$	b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b

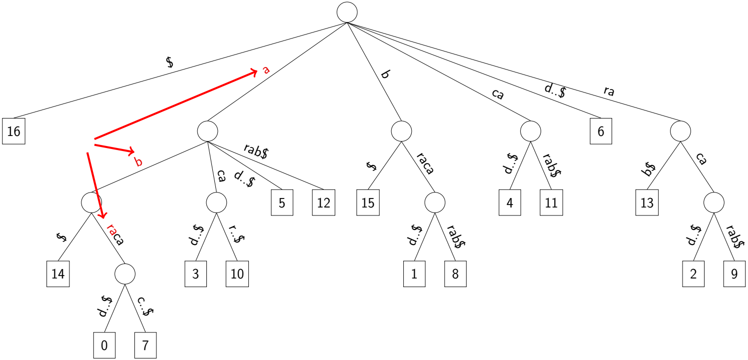
- $c(\texttt{abra}) = 2$ from CSA
  range between $lb = 3$ and $rb = 4$, inclusive
- $N_{1+}(\bullet\ \texttt{abra}) = 2$ from BWT (wavelet tree)
  size of set of preceeding symbols $\{\$, \mathsf{d}\}$

Occurrence counts from the suffix tree



Number of proceeding symbols, $N_{1+}(\alpha\,\bullet)$, is either

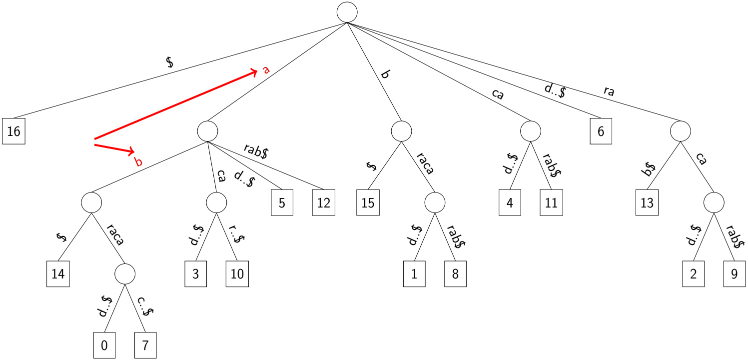
Occurrence counts from the suffix tree



Number of proceeding symbols, $N_{1+}(\alpha\,\bullet)$, is either

- 1 if internal to an edge (e.g., $\alpha =$abra)

Occurrence counts from the suffix tree



Number of proceeding symbols, $N_{1+}(\alpha\, \bullet)$, is either

- 1 if internal to an edge (e.g., $\alpha =$abra)
- $\mathrm{degree}(v)$ otherwise (e.g., $\alpha =$ab with degree 2)

More difficult occurrence counts

How to handle occurrence counts to both sides,

$$N_{1+}(\bullet\ \alpha\ \bullet) = |\{w\alpha v,\ \text{s.t. } c(w\alpha v) \geq 1\}|$$

and specific value i occurrence counts,

$$N_i(\alpha\ \bullet) = |\{\alpha v,\ \text{s.t. } c(\alpha v) = i\}|$$

No simple mapping to CSA/CST algorithm

Iterative (costly!) solution used instead:

- enumerate extensions to one side
- accumulate counts (to the other side, or query if $c = i$)

Algorithm outline

Step 1: search for pattern

Backward search for each symbol, in right-to-left order.
Results in bounds $[lb, rb]$ of matching patterns.

Step 2: find statistics

count $c(\text{a b r a}) = rb - lb - 1$ (or 0 on failure.)

left occ. $N_{1+}(\bullet\, w_i^j)$ can be computed from BWT (over preceeding symbols.)

right occ. $N_{1+}(w_i^j\, \bullet)$ based on shape of the *suffix tree*.

twin occ. etc . . . increasingly complex . . .

Nb. illustrating ideas with basic SA/STs; in practice CSA/CSTs.

Step 2: Compute statistics

Given range $[lb, rb]$ for matching pattern, α, can compute:

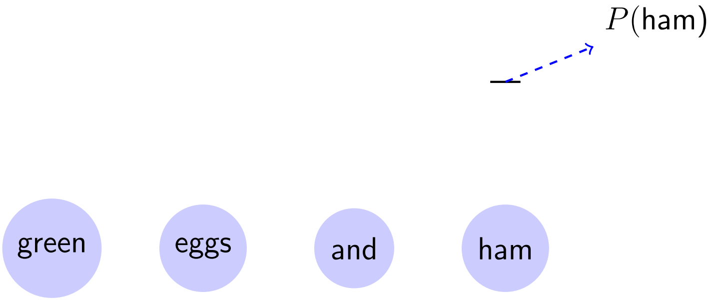
- count, $c(\alpha) = (rb - lb + 1)$
- occurrence count, $N_{1+}(\bullet\, \alpha) = \text{interval-symbols}(lb, rb)$

with time complexity

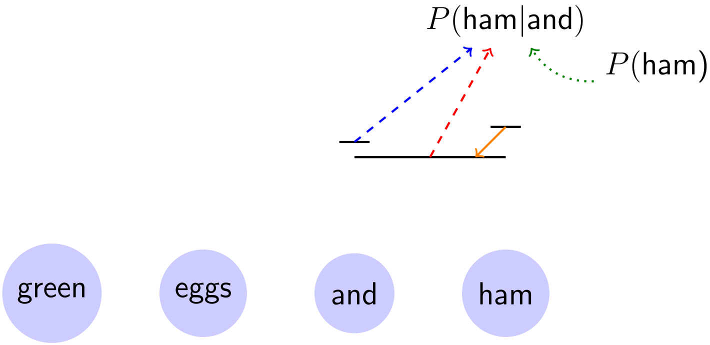
- $o(1)$; and
- $O(N_{1+}(\bullet\, \alpha) \cdot \log \sigma)$ where σ is the size of the vocabulary

What about the other required occurrence counts?

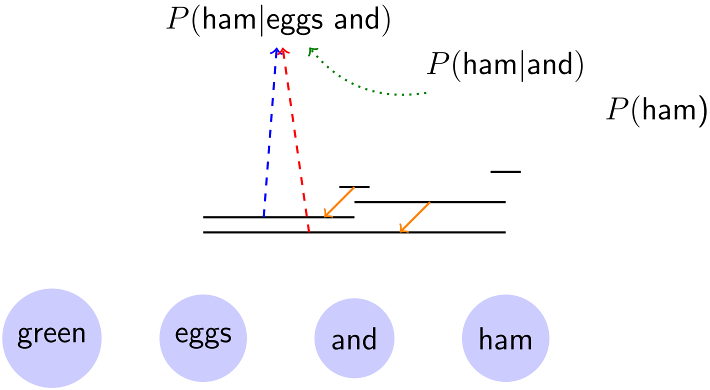
Querying algorithm: one-shot



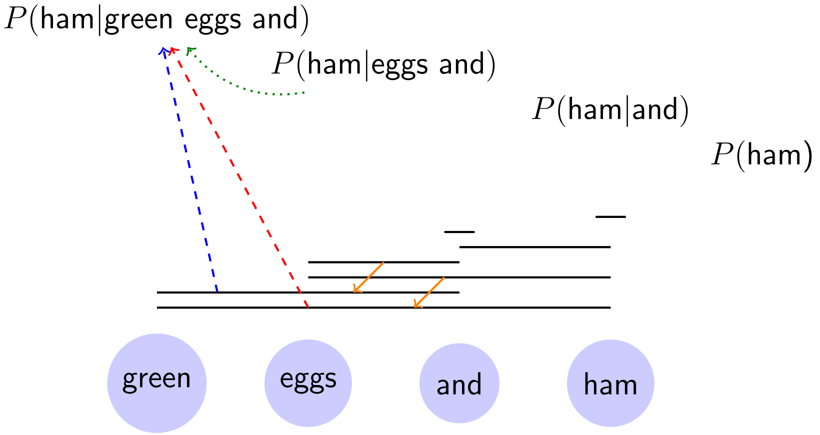
Querying algorithm: one-shot



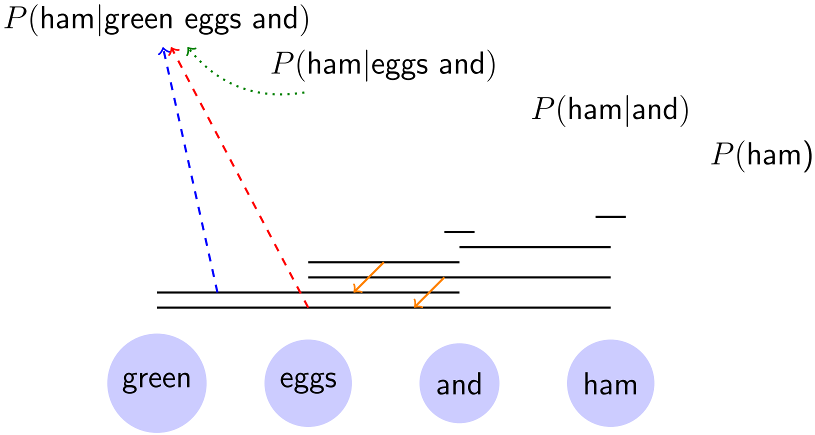
Querying algorithm: one-shot



Querying algorithm: one-shot



Querying algorithm: one-shot



At each step: 1) extend search for context and full pattern; 2) compute c and/or N^{1+} counts.

Querying algorithm: full sentence

Reuse matches

Full matches in one step become context matches for next step. E.g., *green eggs and ham* $\Leftarrow$ *green eggs and*

- recycle the CSA matches from previous query, halving search cost
- N.b., can't recycle counts, as mostly use different types of occurrence counts on numerator cf denominator

Unlimited application

No bound on size of match, can continue until pattern unseen in training corpus.

Construction algorithm

1. Sort suffixes (on disk)
2. Construct CSA
3. Construct CST
4. Compute discounts
   - efficient using traversal of k-grams in the CST (up to a given depth)
5. Precompute some expensive values
   - again use traversal of k-grams in the CST

Accelerating expensive counts

Iterative calls, e.g., $N_{1+}(\bullet\, \alpha\, \bullet)$ account for majority of runtime.

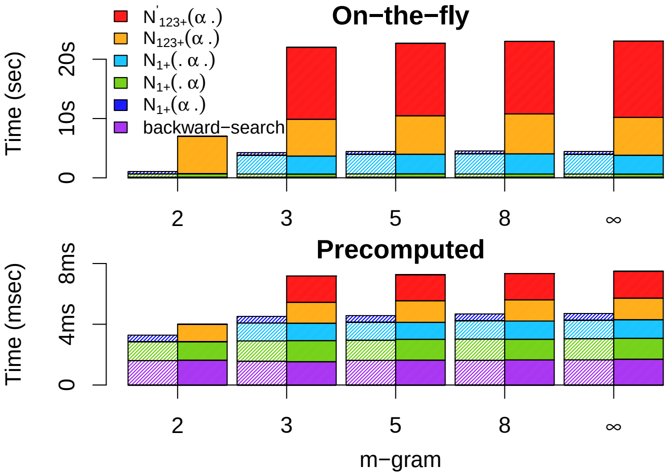
Solution: cache common values

- store values for common entries, i.e., highest nodes in CST
- values are integers, mostly with low values $\rightarrow$ very compressable!

Technique

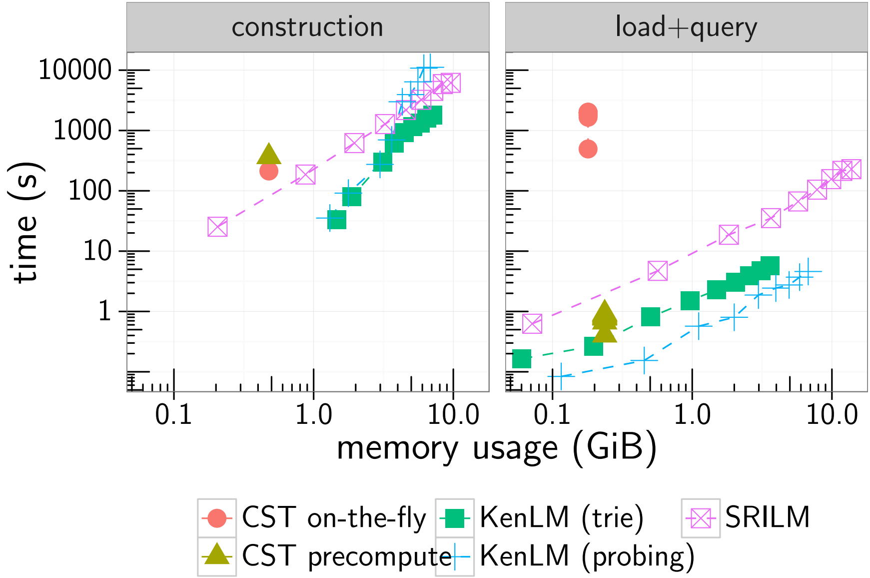
- store bit vector, bv, of length n, where $bv[i]$ records whether value for i is cached
- store cached values in an integer vector, v, in linear order
- retrieve i^{th} value using $v[\mathrm{rank}_1(bv, i)]$

Effect of caching

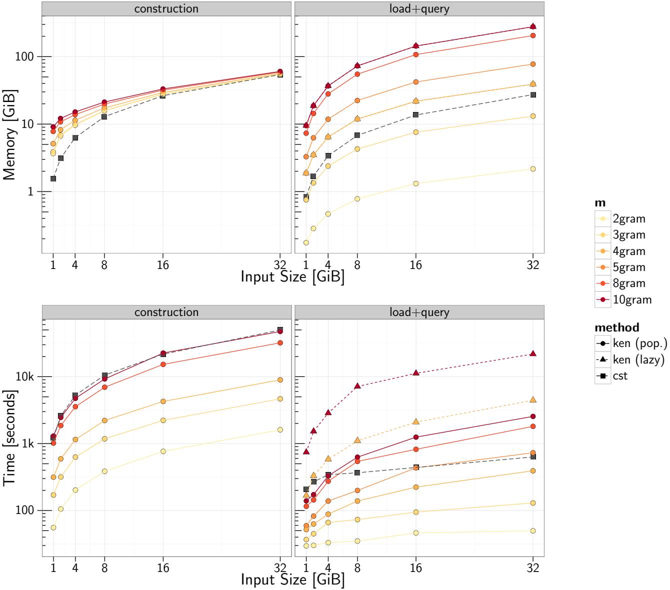


$+$15-20% space requirement (≤ 10-gram)

Timing versus other LMs: Small DE Europarl



Timing versus other LMs: Large DE Commoncrawl



Perplexity: usefulness of large or infinite context

newstest de corpus

Training	size (M) tokens	size (M) sents	perplexity $m=3$	perplexity $m=5$	perplexity $m=10$
Europarl	55	2.2	1004.8	973.3	971.4
NCrawl2007	37	2.0	514.8	493.5	488.9
NCrawl2008	126	6.8	427.7	404.8	400.0
NCrawl2013	641	35.1	268.9	229.8	225.6
NCrawl2014	845	46.3	247.6	195.2	189.3
All combined	2560	139.3	211.8	158.9	151.5
CCrawl32G	5540	426.6	336.6	292.8	287.8

1b word en

unit	time (s)	mem (GiB)	$m=5$	$m=10$	$m=20$	$m=\infty$
word	8164	6.29	73.45	68.66	68.76	68.80
byte	17 935	18.58	3.93	2.69	2.37	2.33

Practical exercise

Finding concordances for an arbitrary k-gram pattern:

Outline

- find count of k-gram in large corpus
- show tokens to left or right, sorted by count
- find pairs of tokens occurring to left and right

Tools

- building a CSA and CST
- searching for pattern
- querying CST path label & children (to right)
- querying WT for symbols to left

Semi-External Indexes

Semi-External Suffix Array (RoSA)

- Store the "top" part of a suffix tree in memory (using a compressed structure)
- If pattern short and frequent. Answer from in-memory structure (fast!)
- If pattern long or infrequent perform disk access
- Implemented, complicated, currently not used in practice

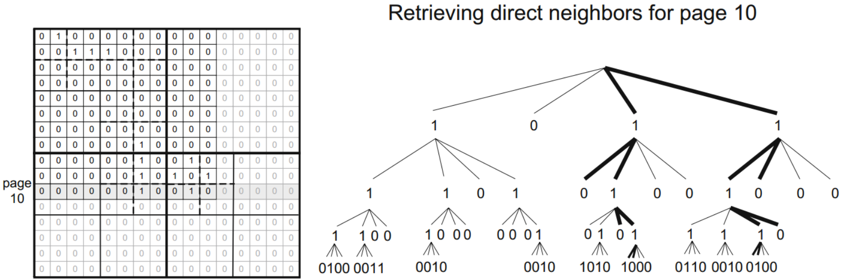
Range Minimum/Maximum Queries

- Given an array A of n items
- For any range $A[i, j]$ answer in constant time, what is the largest / smallest item in the range
- Space usage: $2n + o(n)$ bits. A not required!

Compressed Tries / Dictionaries

- Support $\textsc{lookup}(s)$ which returns unique id if string s is in dict or -1 otherwise
- Support $\textsc{retrieve}(i)$ return string with id i
- Very compact. $10\% - 20\%$ of original data
- Very fast lookup times
- Efficient construction

Graph Compression



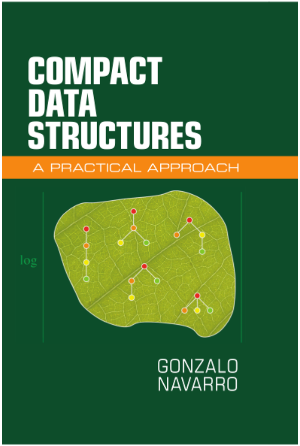
Other applications

- Store the "top" part of a suffix tree in memory (using a compressed structure)
- If pattern short and frequent. Answer from in-memory structure (fast!)
- If pattern long or infrequent perform disk access
- Implemented, complicated, currently not used in practice

Conclusions / take-home message

- Basic succinct structures rely on bitvectors and operations RANK and SELECT
- More complex structures are composed of these basic building blocks
- Many trade-offs exist
- Practical, highly engineered open source implementations exist and can be used within minutes in industry and academia
- Other fields such as Information Retrieval, Bioinformatics have seen many papers using these succinct structures in recent years

Resources



Compact Data Structures,
A practical approach
Gonzalo Navarro
ISBN 978-1-107-15238-0. 570 pages.
Cambridge University Press, 2016

Resources II

- Overview of compressed text indexes: [Ferragina et al., 2008, Navarro and Mäkinen, 2007]
- Bitvectors: [Gog and Petri, 2014]
- Document Retrieval: [Navarro, 2014a]
- Compressed Suffix Trees: [Sadakane, 2007, Ohlebusch et al., 2010]
- Wavelet Trees: [Navarro, 2014b]
- Compressed Tree Representations: [Navarro and Sadakane, 2016]

References I

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007).
Large language models in machine translation.
In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic. Association for Computational Linguistics.

Callison-Burch, C., Bannard, C. J., and Schroeder, J. (2005).
Scaling phrase-based statistical machine translation to larger corpora and longer phrases.
In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Ferragina, P., González, R., Navarro, G., and Venturini, R. (2008).
Compressed text indexes: From theory to practice.
ACM J. of Exp. Algorithmics, 13.

References II

Gog, S. and Petri, M. (2014).
Optimized succinct data structures for massive data.
Softw., Pract. Exper., 44(11):1287–1314.

Heafield, K. (2011).
KenLM: Faster and smaller language model queries.
In *Proceedings of the Workshop on Statistical Machine Translation*.

Kennington, C. R., Kay, M., and Friedrich, A. (2012).
Suffix trees as language models.
In *Proceedings of the Conference on Language Resources and Evaluation*.

Lopez, A. (2008).
Machine Translation by Pattern Matching.
PhD thesis, University of Maryland.

References III

Navarro, G. (2014a).
Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences.
ACM Comp. Surv., 46(4.52).

Navarro, G. (2014b).
Wavelet trees for all.
Journal of Discrete Algorithms, 25:2–20.

Navarro, G. and Mäkinen, V. (2007).
Compressed full-text indexes.
ACM Comp. Surv., 39(1):2.

Navarro, G. and Sadakane, K. (2016).
Compressed tree representations.
In *Encyclopedia of Algorithms*, pages 397–401.

References IV

Ohlebusch, E., Fischer, J., and Gog, S. (2010).

CST++.

In *Proceedings of the International Symposium on String Processing and Information Retrieval*.

Pauls, A. and Klein, D. (2011).

Faster and smaller n-gram language models.

In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*.

Sadakane, K. (2007).

Compressed suffix trees with full functionality.

Theory of Computing Systems, 41(4):589–607.

References V

Shareghi, E., Cohn, T., and Haffari, G. (2016a).

Richer interpolative smoothing based on modified kneser-ney language modeling.

In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 944–949, Austin, Texas. Association for Computational Linguistics.

Shareghi, E., Petri, M., Haffari, G., and Cohn, T. (2015).

Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees.

In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

References VI

Shareghi, E., Petri, M., Haffari, G., and Cohn, T. (2016b).
Fast, small and exact: Infinite-order language modelling with compressed suffix trees.
Transactions of the Association for Computational Linguistics, 4:477–490.

Talbot, D. and Osborne, M. (2007).
Randomised language modelling for statistical machine translation.
In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Zhang, Y. and Vogel, S. (2006).
Suffix array and its applications in empirical natural language processing.
Technical report, CMU, Pittsburgh PA.