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

December 11, 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)

What is it main goal of this tutorial?

Understand the basic concepts and underlying techniques and data structures of a practical, **compressed** text index which can:

- Perform pattern searches efficiently
- Store and extract any part of the original text
- Extract complex statistics (Co-occurrence counts) about arbitrarily length pattern efficiently
- Space usage of the index is equivalent to the compressed size of the input text (e.g. bzip2 size)
- Practical, implemented, easy to use!

Example: Search index over 1GB English text requires 250MiB RAM

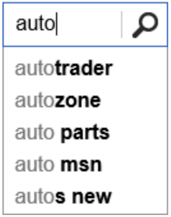
What is it?

- 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

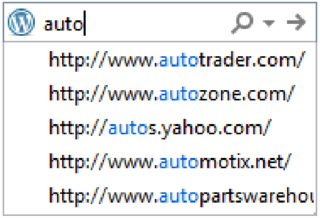
Application 1: Top-k query completion



auto

autotrader
autozone
auto parts
auto msn
autos new

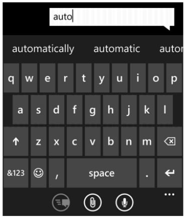
(a) Search engine



auto

http://www.autotrader.com/
http://www.autozone.com/
http://autos.yahoo.com/
http://www.automotix.net/
http://www.autopartswarehou

(b) Browser



auto

automatically automatic autor

(c) Soft keyboard [1]

Formally: Given a set S of strings with associated "scores", for a given query string q, return the k highest scoring strings in S prefixed by q.

[1] Taken from "Space-Efficient Data Structures for Top-k Completion", Hsu and Ottaviano (WWW'13)

Application 1: Top-k query completion

Issue

Indexing by prefix allows fast lookup, but hard to find max count extension efficiently.

- Use range maximum query structure
- Index much smaller than the original string set
- Can answer queries in microseconds
- Practical and a version of this index can be implemented with the structures we will discuss today!

Application 2: Concordance counts

- Trivial to index pairwise word coccurrences on large corpora
- Full concordance more difficult, especially if no limit on context around search pattern
- Concordance queries can be done efficiently over massive corpora using Compressed Suffix Tree and Compressed Suffix Array structures
- Near-optimal memory cost to store corpus

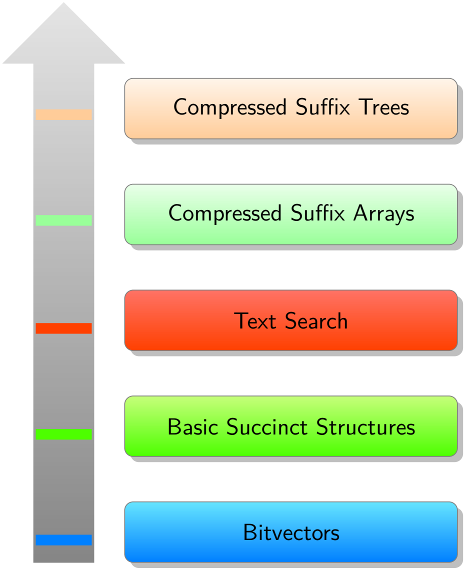
Application 3: Infinite Order Language Models

- Practical Language Model with space usage independent of n-gram size
- Can answer infinite order n-gram queries
- Practical performance similar to state-of-the-art models
- Implemented and usuable for large datasets
- Implemented using CST and CSA structures we will discuss today!

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
- ...


Compressed Suffix Trees
Compressed Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

Practicality

The SDSL library (GitHub repo: link) contains most practical compressed structures we talk about today.

It is easy to install:

```
git clone https://github.com/simongog/sdsl-lite.git
cd sdsl-lite
./install.sh
```

Throughout this tutorial we will show how to use SDSL to create and use a variety of different compressed data structures.

License: Currently GPLv3 but in 1-2 month: BSD. Can be used in a commercial setting!

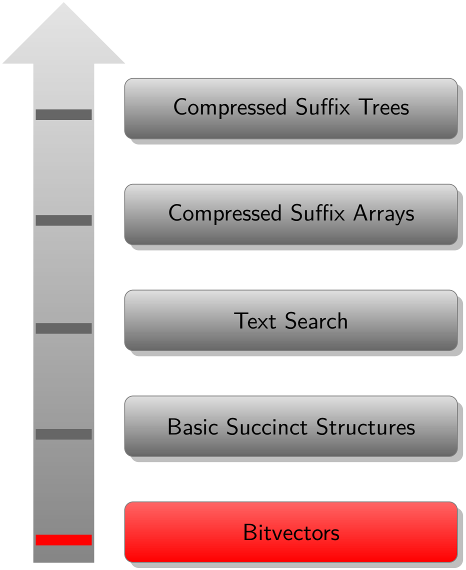
SDSL Resources

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 Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

Basic Technologies and Notation (20 Mins)

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 $\textsc{Rank}$

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

$\textsc{Rank}_1(B, 7) = 5$
$\textsc{Rank}_0(B, 7) = 8 - \textsc{Rank}_1(B, 7) = 3$

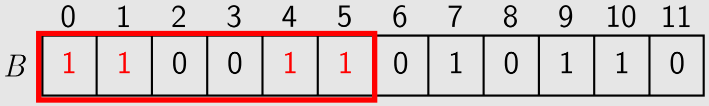
Operation SELECT

Definitions

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

$\textsc{Select}_0(B,j)$: Where is the j-th (start count at 1) 0 in B

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

$\textsc{Select}_1(B,4) = 5$
$\textsc{Select}_0(B,3) = 6$

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

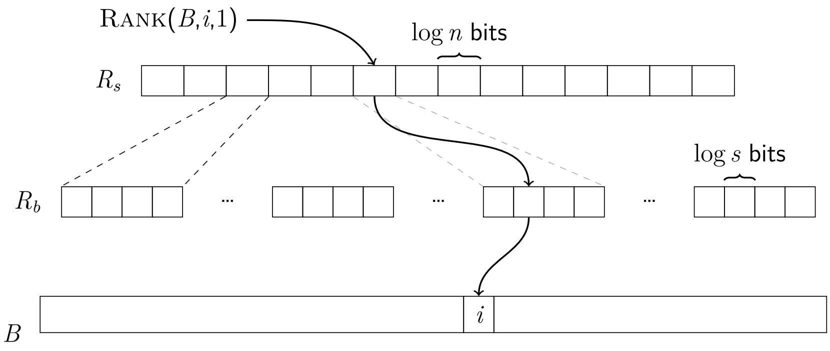
Constant time RANK

Divide bitvector into blocks. Store absolute ranks at block boundaries. Subdivide blocks into subblocks. Store ranks relative to block boundary. Subblocks are $O(\log n)$ which can be processed in constant time. 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

Rank in $O(1)$ time.



Store superblocks every $s = \log^2 n$ bits using $\log_2 n$ bits to store the absolute count.

Divide superblock into blocks of size $\log n$ bots and store relative counts in $\log_2 s$ bits

Space usage: $R_s = n\lceil \frac{\log n}{\log^2 n} \rceil \in o(n)$ bits, $R_b = n\lceil \frac{\log s}{\log n} \rceil \in o(n)$ bits.

Rank in Practice

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

int main() {
  // use a regular bitvector
  using bv_type = sdsl::bit_vector;
  // 5% overhead rank structure to rank 1s
  using rank_type = sdsl::rank_support_v5<1>;
  bv_type bv(1000000);
  // set 10% to 1
  for(auto i=0;i<bv.size();i++) bv[i] = rand()%10==0;
  // build rank structure. BV now immutable
  rank_type rank1(&bv);
  // perform a ranks
  auto num_ones = rank1(bv.size()-1);
  auto ones_before_1k == rank1(1000);
  auto bv_size = sdsl::size_in_bytes(bv);
  auto rank_size = sdsl::size_in_bytes(rank1);
}
```

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 $\textsc{Rank}$ and $\textsc{Select}$

In Practice

Bitvector of size 1 GiB marking all uppercase letters in 8 GiB wikiepdia text:
Encodings:

- Elias-Fano ['73]: 343 MiB
- RRR ['02]: 335 MiB

Elias-Fano Coding

Elias-Fano Coding

Given a non-decreasing sequence X of length m over alphabet $[0..n]$. X can be represented using $2m + m \log \frac{n}{m} + o(m)$ bits while each element can still be accessed in constant time.

This representation can also be used to represent a bitvector (e.g. n is bitvector length, m the number of set bits, and X the position of the set bits)

How does Elias-Fano coding work?

$X =$ 4 13 15 24 26 27 29

How does Elias-Fano coding work?

$X =$	4	13	15	24	26	27	29
	00100	01101	01111	11000	11010	11011	11101

How does Elias-Fano coding work?

$$X = \quad 4 \quad 13 \quad 15 \quad 24 \quad 26 \quad 27 \quad 29$$

$$\underbrace{00100}\ \underbrace{01101}\ \underbrace{01111}\ \underbrace{11000}\ \underbrace{11010}\ \underbrace{11011}\ \underbrace{11101}$$

How does Elias-Fano coding work?

$$X = \quad 4 \quad 13 \quad 15 \quad 24 \quad 26 \quad 27 \quad 29$$

$$00\underbrace{100}_{4}\; 01\underbrace{101}_{5}\; 01\underbrace{111}_{7}\; 11\underbrace{000}_{0}\; 11\underbrace{010}_{2}\; 11\underbrace{011}_{3}\; 11\underbrace{101}_{5}$$

$$L = \quad 4 \quad 5 \quad 7 \quad 0 \quad 2 \quad 3 \quad 5$$

How does Elias-Fano coding work?

$$X = \quad 4 \quad 13 \quad 15 \quad 24 \quad 26 \quad 27 \quad 29$$

$$\underbrace{00}_{0}\underbrace{100}_{4}\ \underbrace{01}_{1}\underbrace{101}_{5}\ \underbrace{01}_{1}\underbrace{111}_{7}\ \underbrace{11}_{3}\underbrace{000}_{0}\ \underbrace{11}_{3}\underbrace{010}_{2}\ \underbrace{11}_{3}\underbrace{011}_{3}\ \underbrace{11}_{3}\underbrace{101}_{5}$$

$$L = \quad 4 \quad 5 \quad 7 \quad 0 \quad 2 \quad 3 \quad 5$$

How does Elias-Fano coding work?

$$X = \quad 4 \quad 13 \quad 15 \quad 24 \quad 26 \quad 27 \quad 29$$

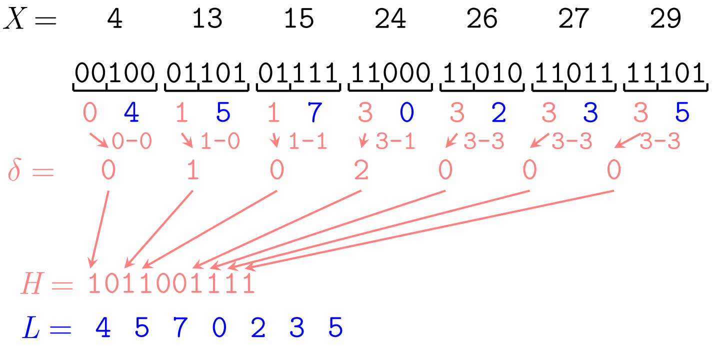
$$\underbrace{00\,100}_{0\quad 4}\;\underbrace{01\,101}_{1\quad 5}\;\underbrace{01\,111}_{1\quad 7}\;\underbrace{11\,000}_{3\quad 0}\;\underbrace{11\,010}_{3\quad 2}\;\underbrace{11\,011}_{3\quad 3}\;\underbrace{11\,101}_{3\quad 5}$$

$$0-0 \quad 1-0 \quad 1-1 \quad 3-1 \quad 3-3 \quad 3-3 \quad 3-3$$

$$\delta = \quad 0 \quad 1 \quad 0 \quad 2 \quad 0 \quad 0 \quad 0$$

$$L = \quad 4 \quad 5 \quad 7 \quad 0 \quad 2 \quad 3 \quad 5$$

How does Elias-Fano coding work?



How does Elias-Fano coding work?

- Divide each element into two parts: high-part and low-part.
- $\lfloor \log m \rfloor$ high-bits and $\lceil \log n \rceil - \lfloor \log m \rfloor$ low bits
- Sequence of high-parts of X is also non-decreasing.
- Gap encode the high-parts and use unary encoding to represent gaps. Call result H.
- I.e. for a gap of size g_i we use $g_i + 1$ bits (g_i zeros, 1 one).
- Sum of gaps $(= \#zeros)$ is at most $2^{\lfloor \log m \rfloor} \le 2^{\log m} = m$
- I.e. H has size at most $2m$ $(\#zeros + \#ones)$
- Low-parts are represented explicitly.

[2]

[2]Coming slides taken from Simon Gog: Advanced Data Structures (KIT)

How does Elias-Fano coding work?

Constant time access

- Add a select structure to H (Okanohara & Sadakane '07).

```
ACCESS($i$)
    $p \leftarrow \textsc{Select}_1(H, i+1)$
    $x \leftarrow p - i$
    **return** $x \cdot 2^{\lceil \log n \rceil - \lfloor \log m \rfloor} + L[i]$
```

Elias-Fano in Practice

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

int main() {
  // use a regular bitvector
  using bv_type = sdsl::bit_vector;
  bv_type bv(1000000);
  for(auto i=0;i<bv.size();i++) bv[i] = rand()%10==0;
  // create EF encoding. again immutable
  sd_vector<> sdv(bv);
  sd_vector<>::rank_1_type rank1(&sbv);
  // perform a ranks
  auto num_ones = rank1(bv.size()-1);
  auto ones_before_1k == rank1(1000);
  auto bv_size = sdsl::size_in_bytes(bv);
  auto ef_size = sdsl::size_in_bytes(sbv);
  auto rank_size = sdsl::size_in_bytes(rank1);
}
```

Bitvectors - Practical Performance

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

Uncompressed:

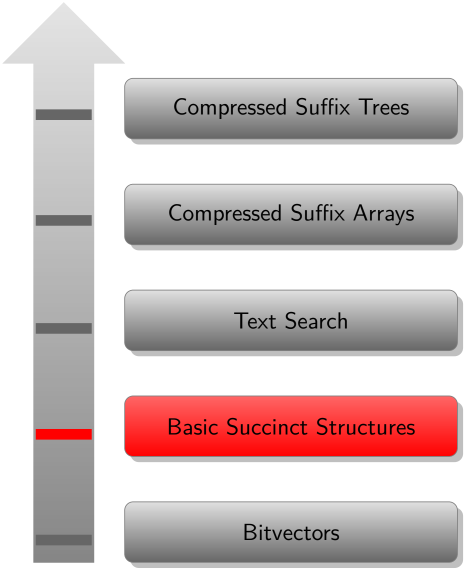
BV Size	Access	Rank	Select	Space
1MB	3ns	4ns	47ns	127%
10MB	10ns	14ns	85ns	126%
1GB	26ns	36ns	303ns	126%
10GB	78ns	98ns	372ns	126%

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)


Compressed Suffix Trees
Compressed Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

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.

Example: Succinct Tree Representations:

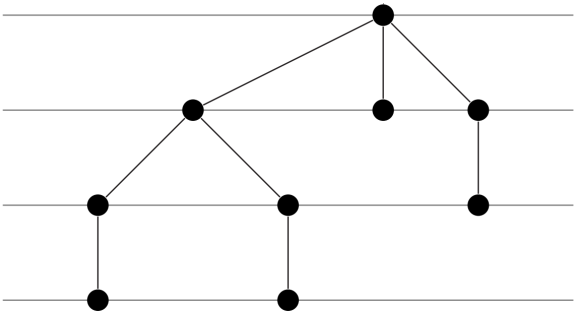
The number of unique binary trees containing n nodes is (roughly) 4^n. To differentiate between them we need at least $log_2(4^n) = 2n$ bits. Thus, a succinct tree representations should require $2n + o(n)$ bits.

LOUDS –level order unary degree sequence

LOUDS

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

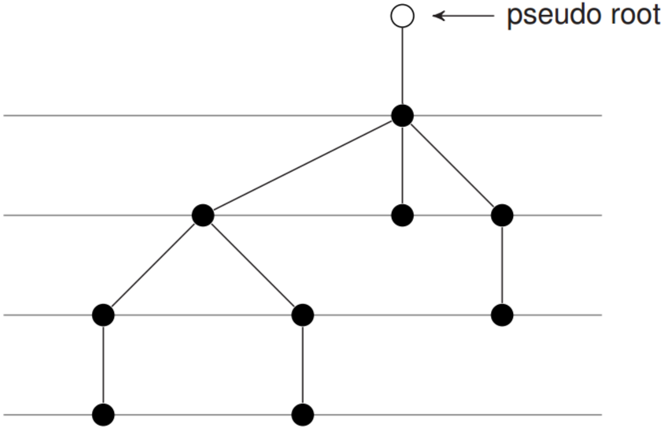
Example:[3]



[3]Taken from Simon Gog: Advanced Data Structures (KIT)

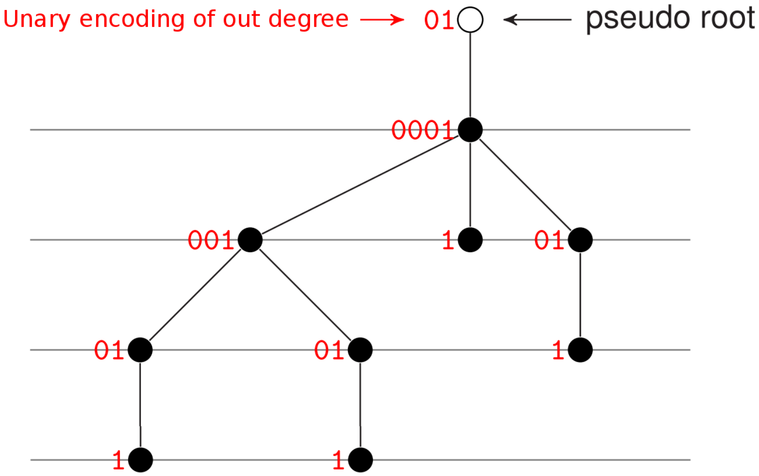
LOUDS –Step 1

Add Pseudo Root:



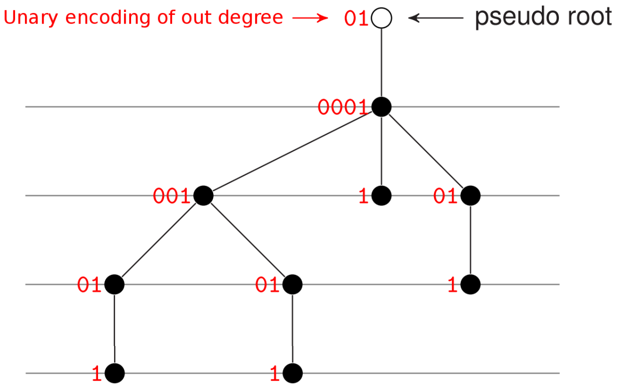
LOUDS –Step 2

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 $\textsc{Rank}$ and $\textsc{Select}$ to navigate the tree in constant time

Examples:

Compute node degree

```
int node_degree(int v) {
  if is_leaf(v) return 0
  id = $\textsc{Rank}_0(L, v)$
  return $\textsc{Select}_1(L, id + 2)$
     $-\textsc{Select}_1(L, id + 1) - 1$
}
```

Return the i-th child of node v

```
int child(int v,i) {
  if i > node_degree(v)
    return -1
  id = $\textsc{Rank}_0(L, v)$
  return $\textsc{Select}_1(L, id + 1) + i$
}
```

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 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

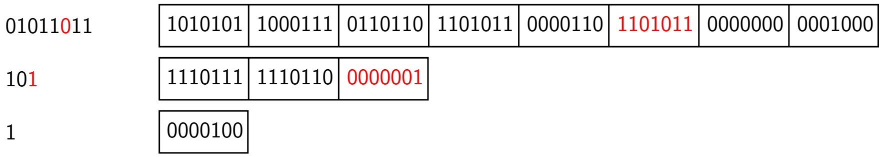
DAC restructuring of the vbyte encoded sequence of integers:

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



Accessing element A[5]:

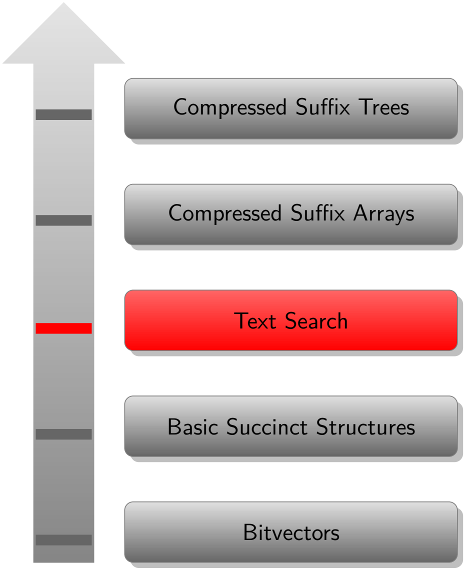
- Access indicator bit of the first level at position 5: $I1[5] = 0$
- 0 in the indicator bit implies the number uses at least 2 bytes
- Perform $Rank_0(I1, 5) = 3$ to determine the number of integers in $A[0, 5]$ with at least two bytes
- Access $I2[3-1] = 1$ to determine that number $A[5]$ has two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

Practical Exercise

```
#include <vector>
#include "sdsl/dac_vector.hpp"

int main(int , char const *argv[])
{ using u32 = uint32_t; sdsl::int_vector<8> T;
  sdsl::load_vector_from_file(T,argv[1],1);
  std::vector<u32> counts(256*256*256,0);
  u32 cur3gram = (u32(T[0]) << 16) | (u32(T[1]) << 8);
  for(size_t i=2;i<T.size();i++) {
    cur3gram = ((cur3gram&0x0000FFFF)<<8) | u32(T[i]);
    counts[cur3gram]++;
  }
  std::cout << "u32 = " << sdsl::size_in_mega_bytes(counts);
  sdsl::dac_vector<3> dace(counts);
  std::cout << "dac = " << sdsl::size_in_mega_bytes(dace);
}
```

Code: here.


Compressed Suffix Trees
Compressed Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

Index based Pattern Matching (20 Mins)

Problem Definition

Given a string T and a pattern P over an alphabet Σ of constant size σ. Let $n = |T|$ be the length of T, and $m = |P|$ be the length of P and $n \gg m$.

Example

$T = abracadabrabarbara\$$
$P = bar$
$\Sigma = \{\$, a, b, c, d, r\}, \sigma = 6, n = 18, m = 3$

Problem: String search

- Does P occur in T? (Existence query)
- How often does P occur in T? (Count query)
- Where does P occur in T? (Locate query)

Problem Solutions

Scanning the text:

- Knuth, Morris, and Pratt precomputed a table of size m which allows to shift the pattern by possibly more than one position in case of a mismatch and get complexity: $\mathcal{O}(n+m)$
- This solution is optimal in the online scenario, in which we are not allowed to pre-process T (online scenario), but not in ...

Our scenario

We are allowed to pre-compute an index structure I for T and use I for the string search.

- I should be small
- Time complexity of matching independent of n

First Index: 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

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

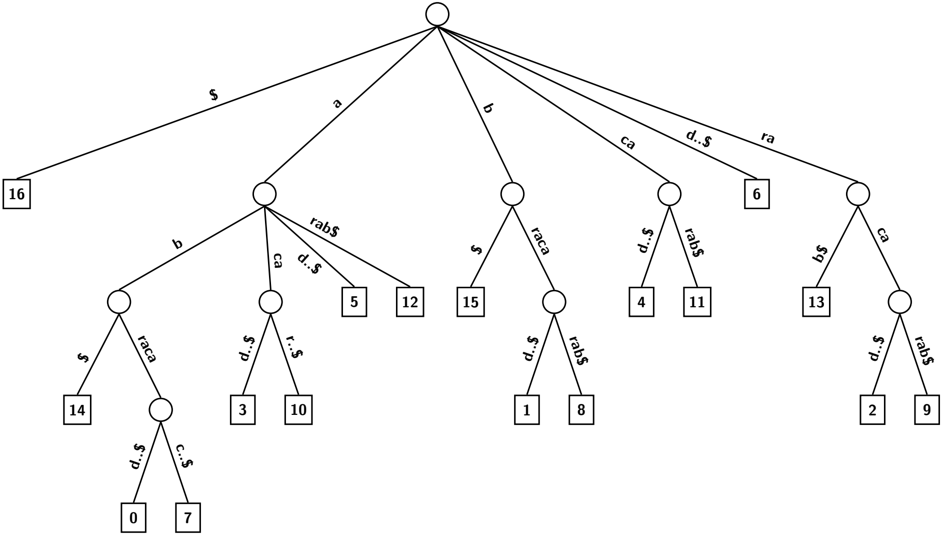
Suffix Tree - Example

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

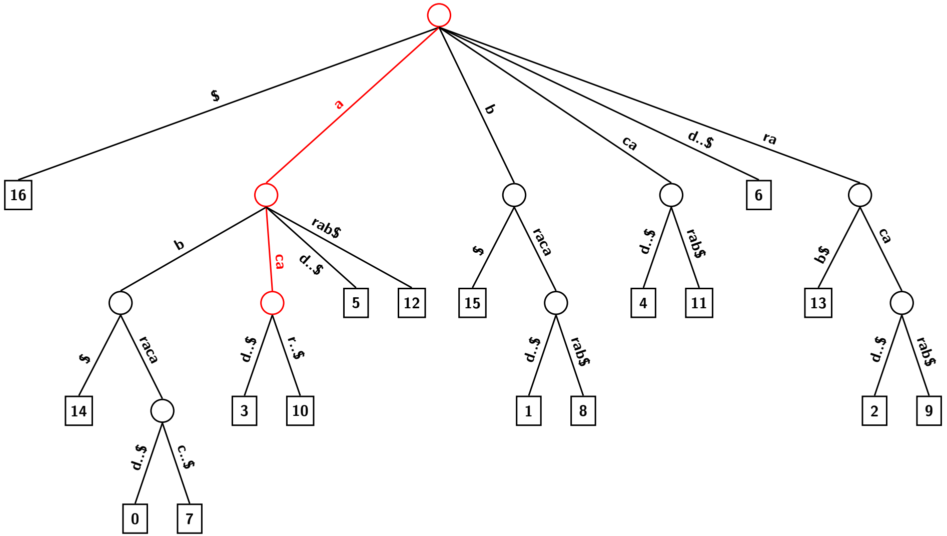
Suffixes:

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 and Myers'92)

- 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

$T =$`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$

First attempt: Suffix Arrays (1)

i	$SA[i]$	$T[SA[i]..n-1]\,T[0..SA[i]-1]$
18	18	$abracadabrabarbara
17	17	a$abracadabrabarbar
10	10	abarbara$abracadabr
7	7	abrabarbara$abracad
0	0	abracadabrabarbara$
3	3	acadabrabarbara$abr
5	5	adabrabarbara$abrac
15	15	ara$abracadabrabarb
12	12	arbara$abracadabrab
14	14	bara$abracadabrabar
11	11	barbara$abracadabra
8	8	brabarbara$abracada
1	1	bracadabrabarbara$a
4	4	cadabrabarbara$abra
6	6	dabrabarbara$abraca
16	16	ra$abracadabrabarba
9	9	rabarbara$abracadab
2	2	racadabrabarbara$ab
13	13	rbara$abracadabraba

- First sort suffixes of T. (quicksort: $\mathcal{O}(n^2 \log n)$, best algorithms: $\mathcal{O}(n)$)
- Storing all suffixes takes $n^2 \log \sigma$ bits space. Only store starting positions of suffixes in SA ($n \log n$ bits).
- Question: How fast can we search using T and SA?

First attempt: Suffix Arrays (2)

- The suffixes are *ordered* in SA. We can use *binary search*!
- Start with the empty string ϵ which matches all prefixes (i.e. the interval $[sp_0..ep_0] = [0..n-1]$) of suffixes in SA.
- Then use binary search to determine the interval $SA[sp_j..ep_j]$ in $SA[sp_{j-1}..ep_{j-1}]$ so that all suffixes start with $P[0..j-1]$ for all $j \in [1..m]$.
- P occurs in T if $[sp_m..ep_m]$ is not empty.
- If P occurs the count query can be answered by $ep_m - sp_m + 1$.
- Time complexity: $\mathcal{O}(m \cdot \log n)$, space $\mathcal{O}(n \log n + n \log \sigma)$

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]\,T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for bar.

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]\,T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for bar.
- Step 1: b interval [9..12]

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]\, T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for bar.
- Step 1: b interval [9..12]
- Step 2: ba interval [9..10]

First attempt: Suffix Arrays, Example

i	$SA[i]$	$T[SA[i]..n-1]\,T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for bar.
- Step 1: b interval [9..12]
- Step 2: ba interval [9..10]
- Step 2: bar interval [9..10]

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

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

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

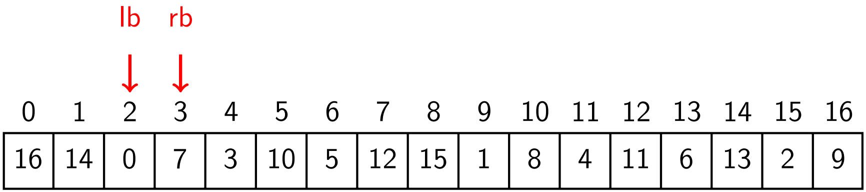
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	r	a	c	a	d	a	b	r	a	c	a	r	a	b	b

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



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 not $O(n)$
- 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

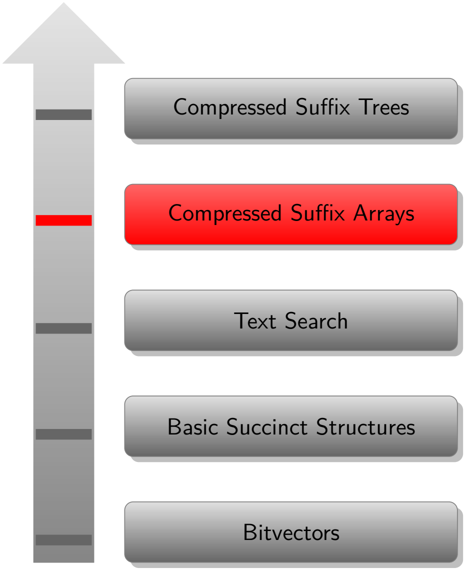
Solutions?

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?

- Compression?


Compressed Suffix Trees
Compressed Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

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);
}
```

Code: here.
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

$T =$ 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

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

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
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

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
$\longrightarrow$ 14	r	a
15	r	b
$\longrightarrow$ 16	r	b

Search backwards, start by finding the r interval in F

Searching using the BWT

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

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

	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
$\longrightarrow$	14	r	a
	15	r	b
$\longrightarrow$	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
$\longrightarrow$ 14	r	a
15	r	b
$\longrightarrow$ 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
$\longrightarrow$	9	b	a
$\longrightarrow$	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

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

	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
$\longrightarrow$	9	b	a
$\longrightarrow$	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

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

	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
$\rightarrow$	2	a	\$
$\rightarrow$	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

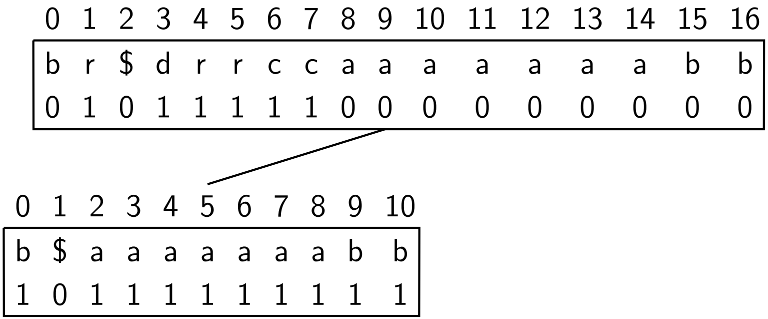
Wavelet Trees - Example

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

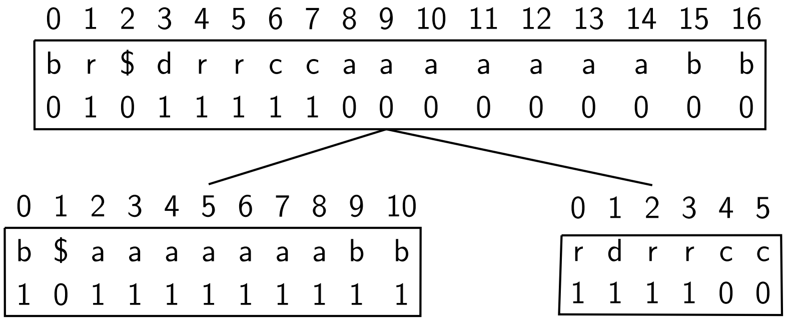
Wavelet Trees - Example



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

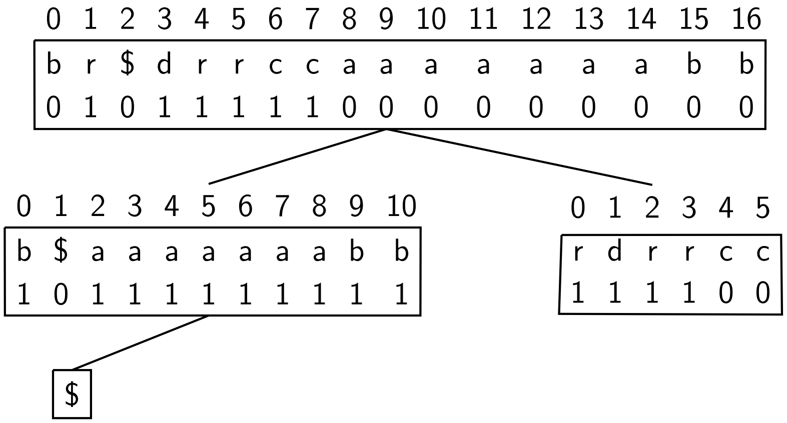


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	r	$	d	r	r	c	c	a	a	a	a	a	a	a	b	b
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

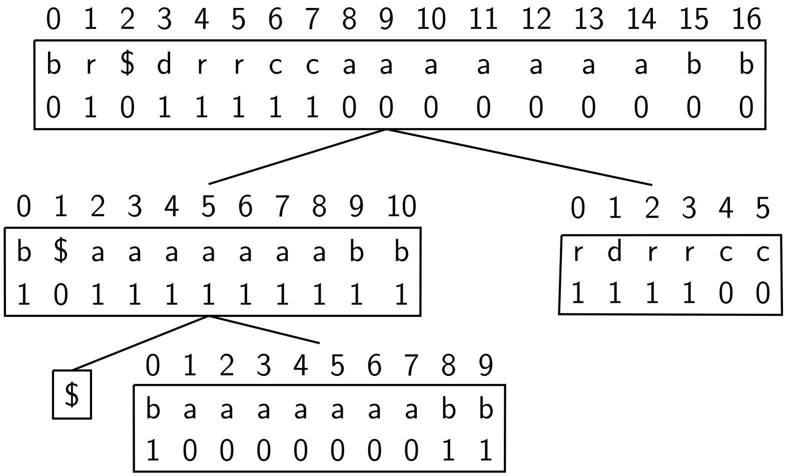
0	1	2	3	4	5	6	7	8	9	10
b	$	a	a	a	a	a	a	a	b	b
1	0	1	1	1	1	1	1	1	1	1

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

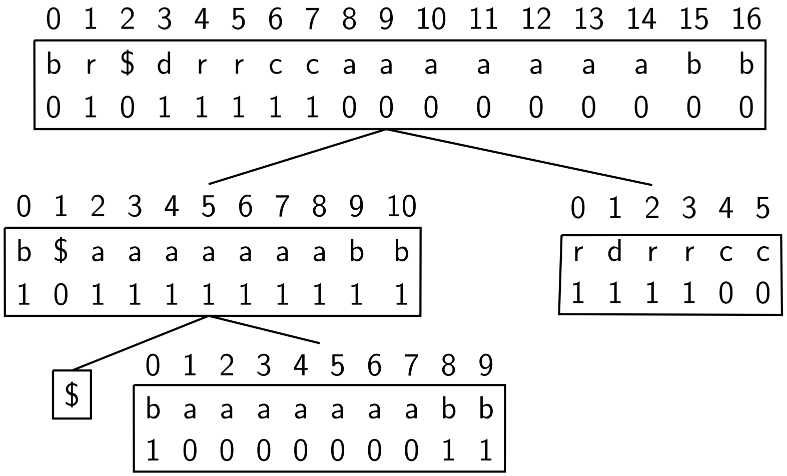
Wavelet Trees - Example



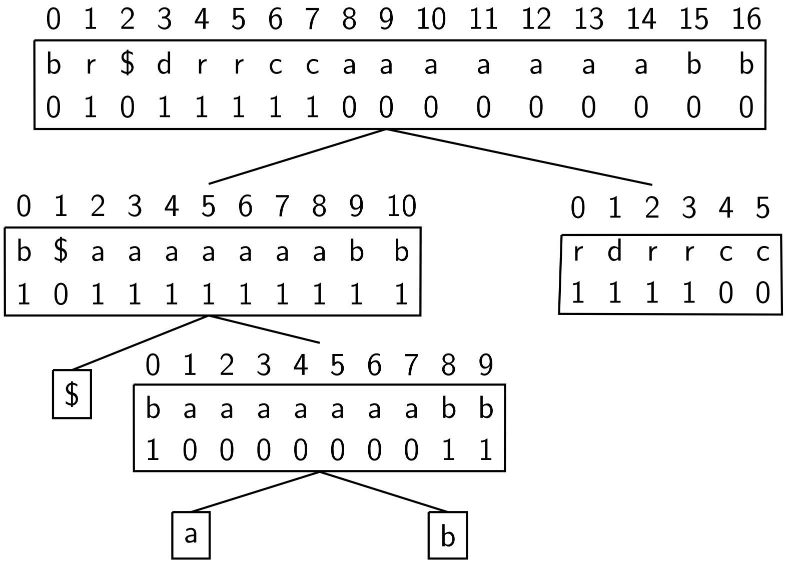
Wavelet Trees - Example



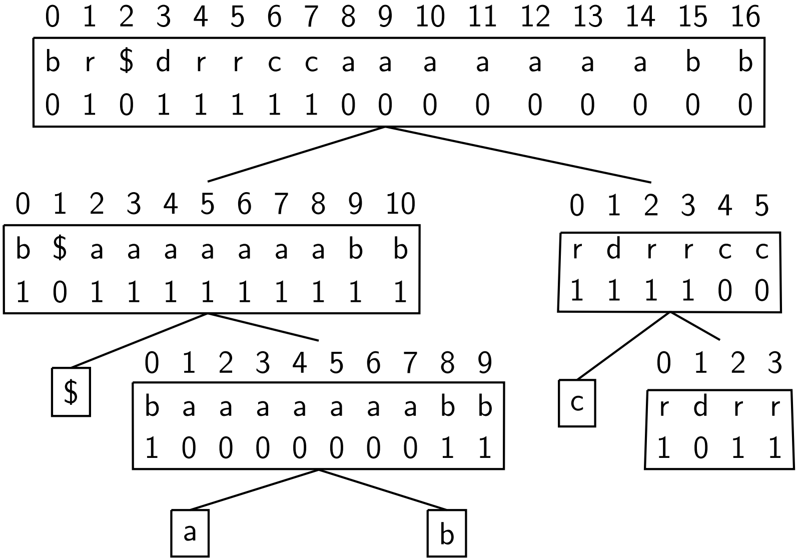
Wavelet Trees - Example



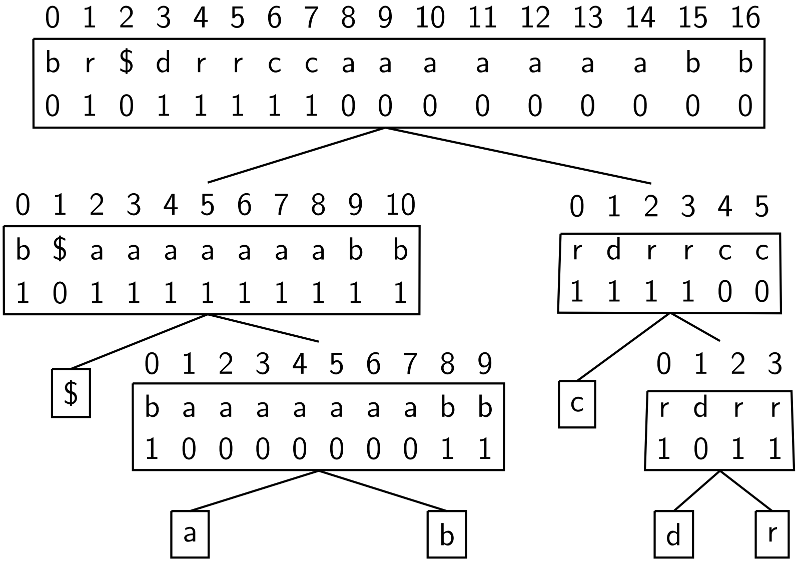
Wavelet Trees - Example



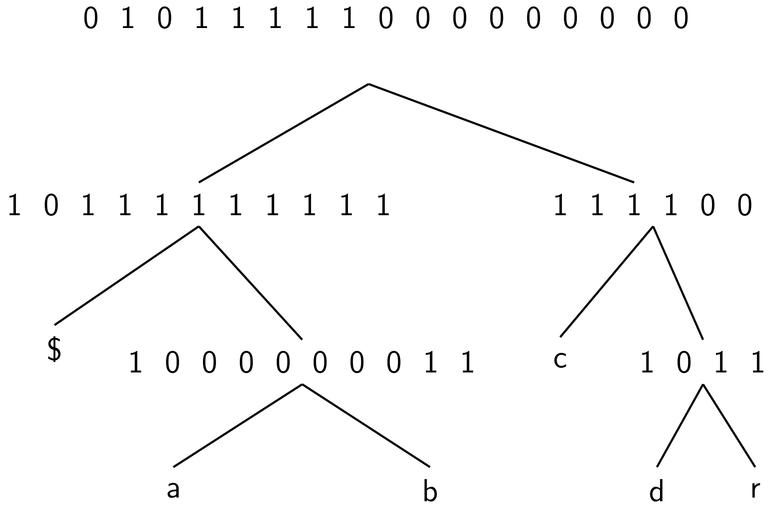
Wavelet Trees - Example



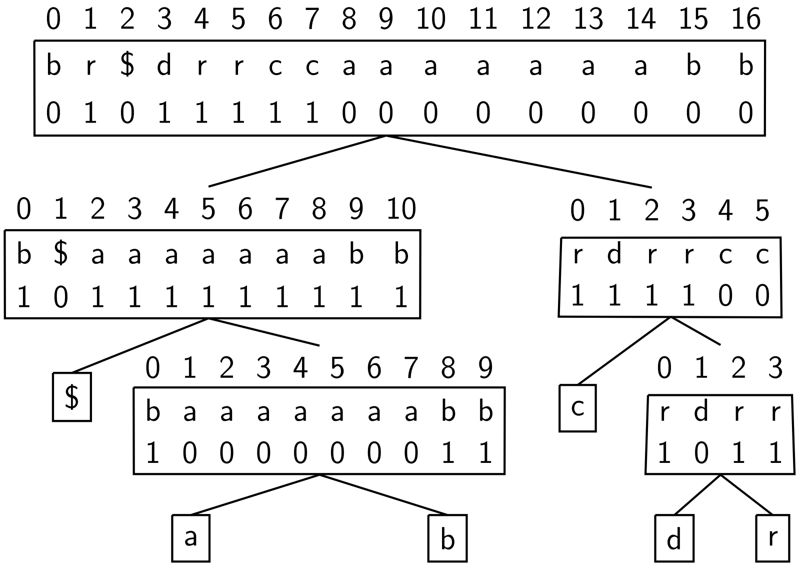
Wavelet Trees - Example



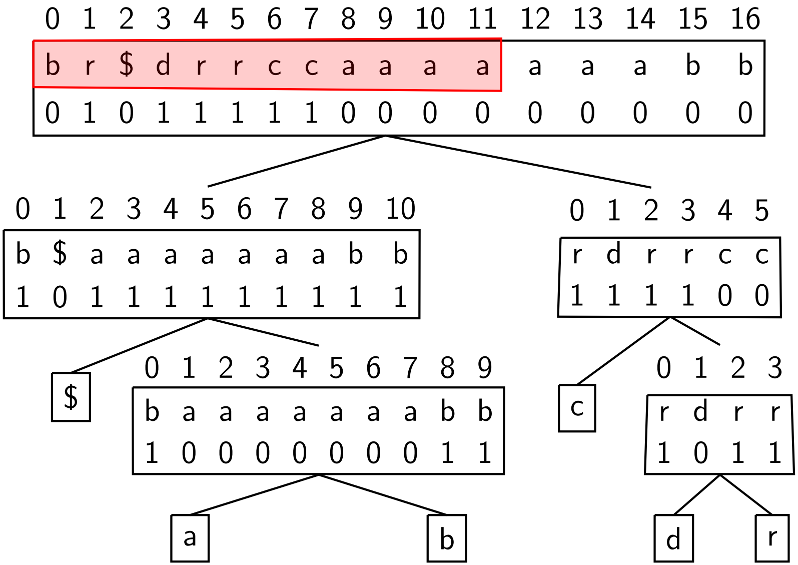
Wavelet Trees - What is actually stored



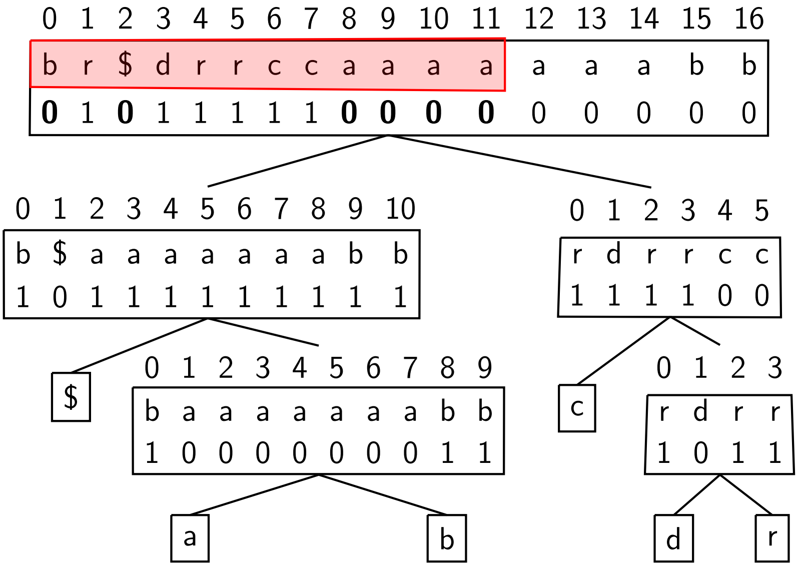
Wavelet Trees - Performing $Rank_a(BWT, 11)$



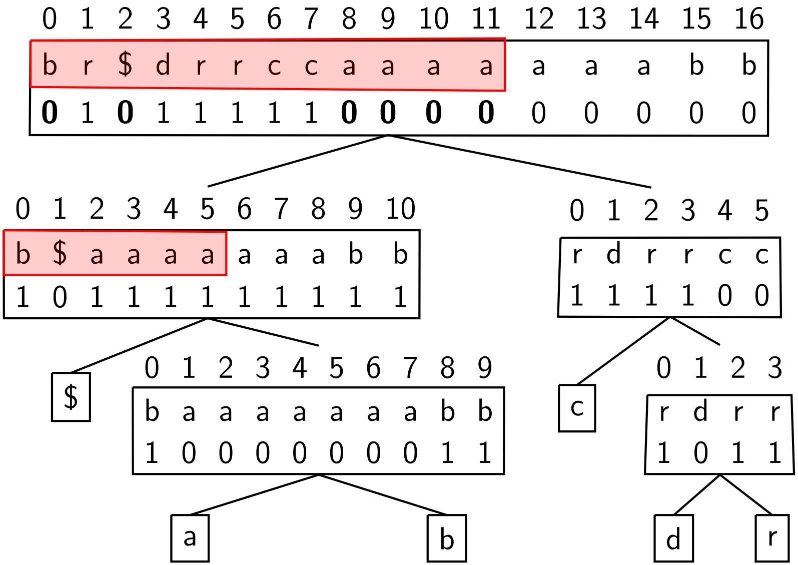
Wavelet Trees - Performing $Rank_a(BWT, 11)$



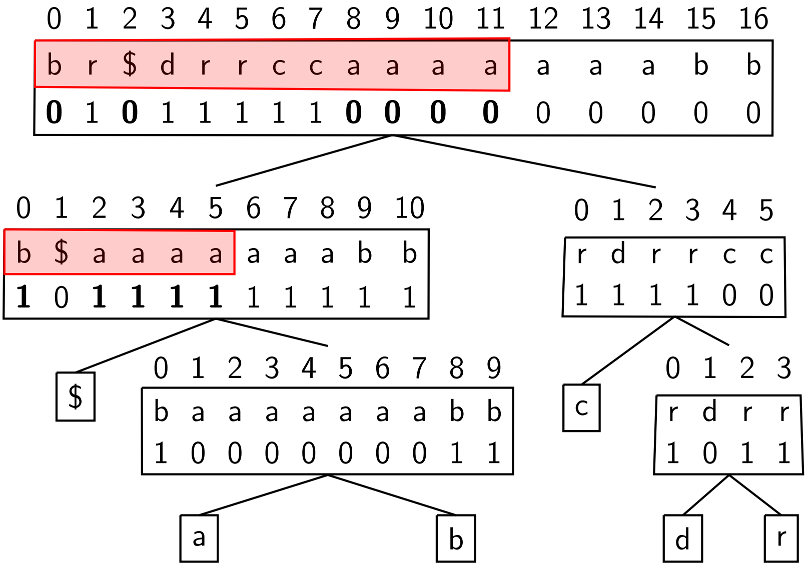
Wavelet Trees - Performing $Rank_a(BWT, 11)$



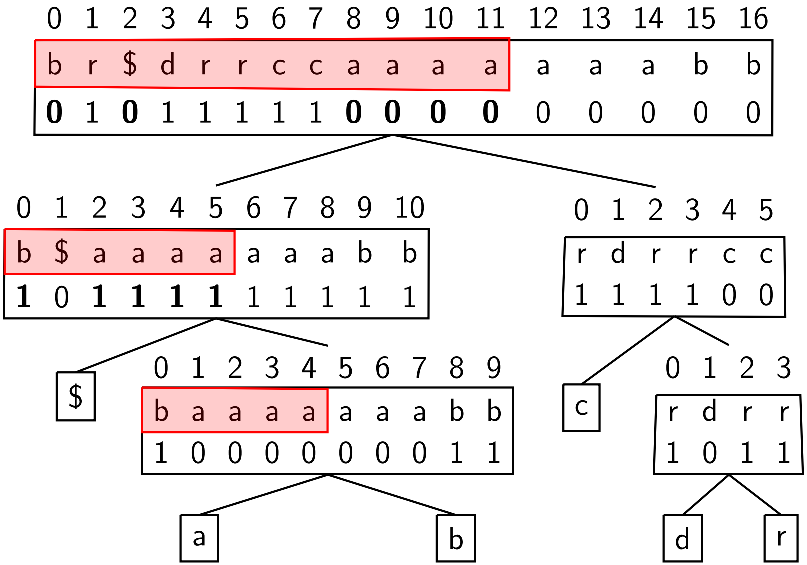
Wavelet Trees - Performing $Rank_a(BWT, 11)$



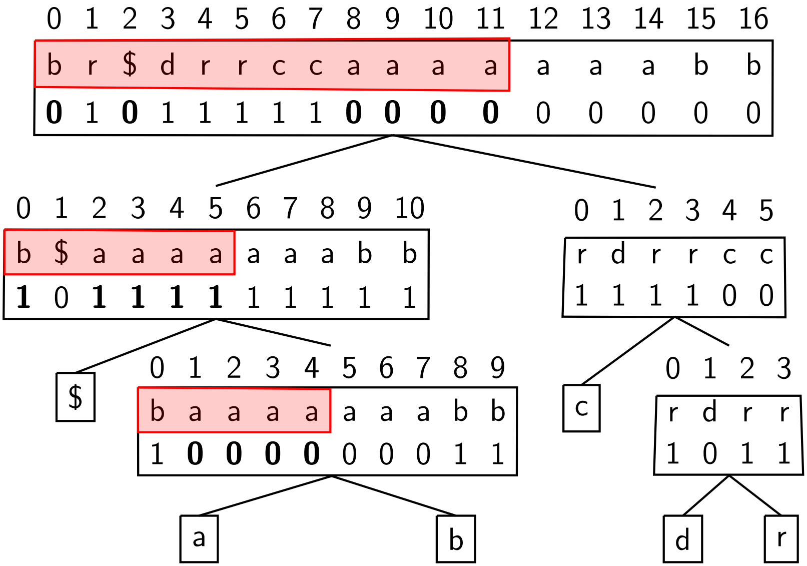
Wavelet Trees - Performing $Rank_a(BWT, 11)$



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

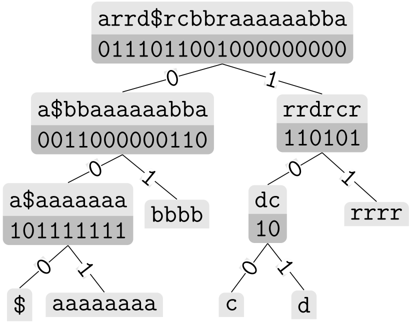
Bits per symbol: 2.29

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

Even better: Huffman shape + compressed bitvectors

Simple solution for rank (second attempt)

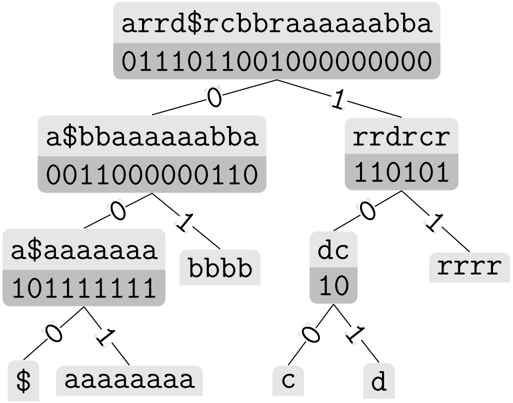
Use a wavelet tree to handle general alphabets:



Char c	$codeword(c)$	freq
$	000	1
a	001	8
b	01	4
c	100	1
d	101	1
r	11	4

Depth: $\log \sigma$. Only bitvectors and pointers to bitvectors are stored. Total space: $\approx n \log \sigma + 2\sigma \log n$

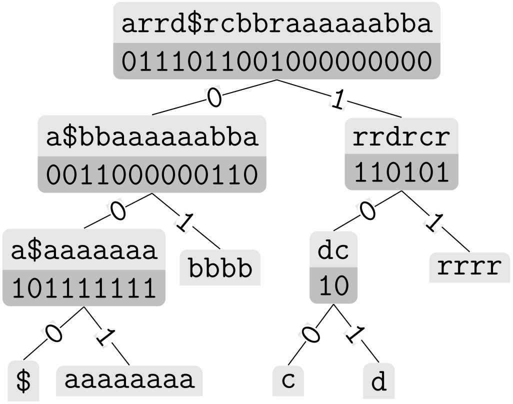
Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) =$

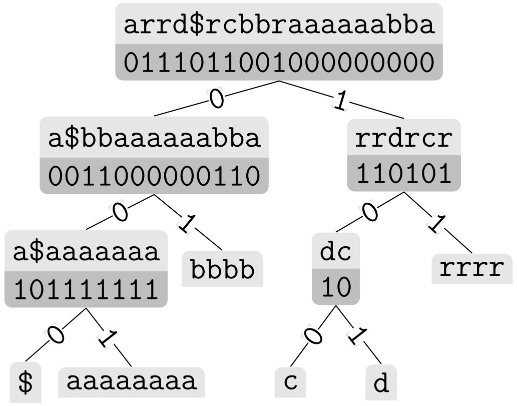
Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) =$ $rank(11, 0, b_\epsilon) = 5$

Wavelet Tree Example: Calculate Rank

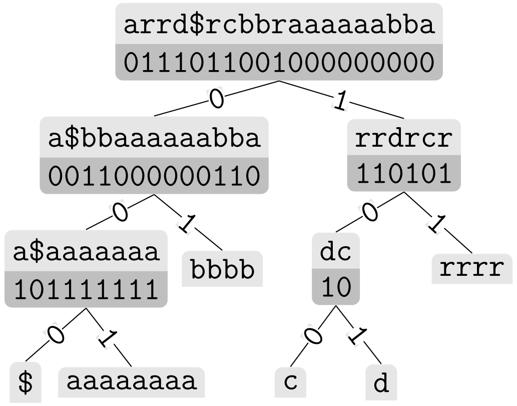


a = 001

$rank(11, a, WT) =$ $rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) =$

3

Wavelet Tree Example: Calculate Rank



a = 001

$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$

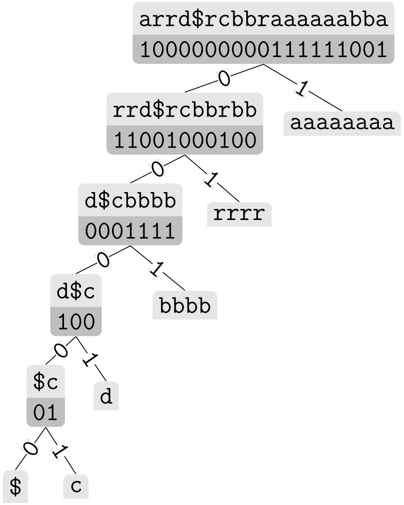
Pseudocode for rank on WT

$rank(i, c, WT)$

```
$p \leftarrow b_\epsilon$
$j \leftarrow 0$
**while not** $p! = codeword(c)$ **do**
    **if** $codeword(c)[j] = 0$ **then**
        $i \leftarrow i - rank(i, 1, b_p)$
        $p \leftarrow p0$
    **else**
        $i \leftarrow rank(i, 1, b_p)$
        $p \leftarrow p1$
**return** $i$
```

This code can also be used in a more space-efficient WT variant.

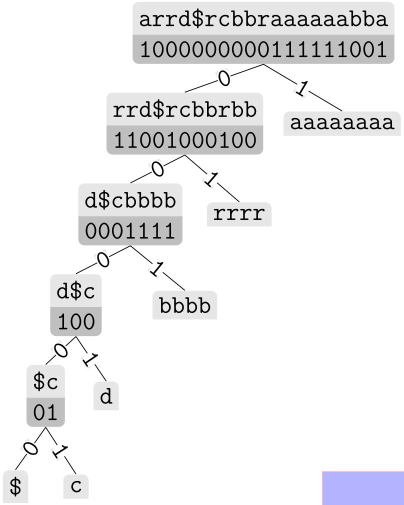
Huffman shaped wavelet tree



Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

Avg. depth: $H_0(BWT)$. Total space: $\approx nH_0 + 2\sigma \log n$

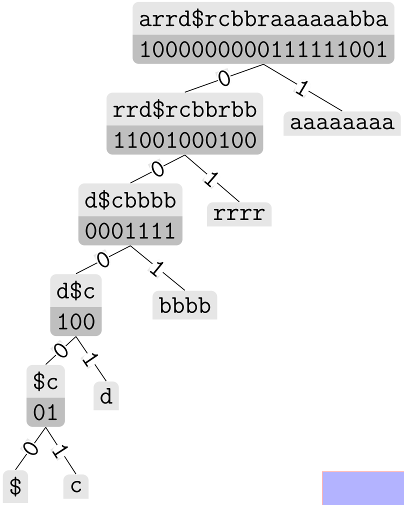
Huffman shaped wavelet tree



Char	c	$codeword(c)$
	$	00000
	a	1
	b	001
	c	00001
	d	0001
	r	01

$$rank(11, a, WT) = rank(11, 1, b_\epsilon)$$

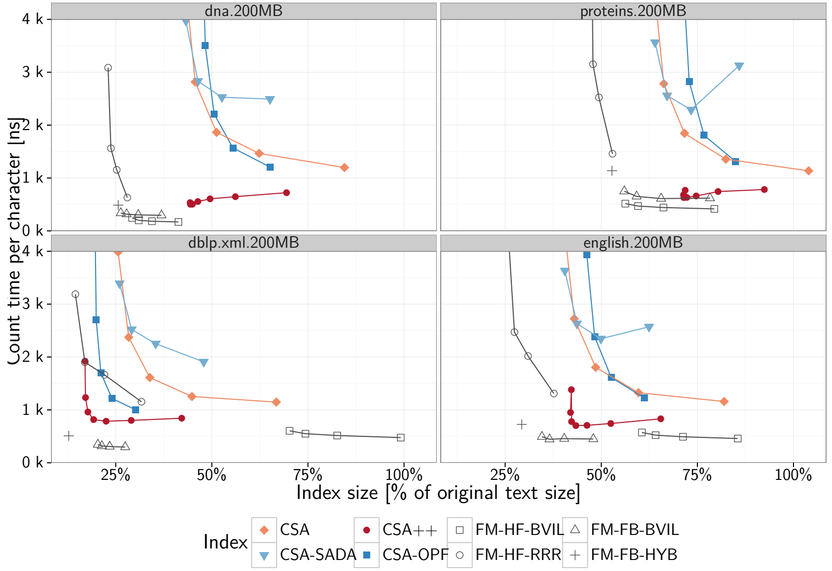
Huffman shaped wavelet tree



Char c	$codeword(c)$
\$	00000
a	1
b	001
c	00001
d	0001
r	01

$$rank(11, a, WT) = rank(11, 1, b_\epsilon) = 2$$

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 - Searching

```
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; // 接尾辞配列接尾辞配列接尾辞配列
sdsl::construct(csa, "this-file.cpp", 1);
std::cout << "count("配列") : "
     << sdsl::count(csa, "配列") << 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;
```

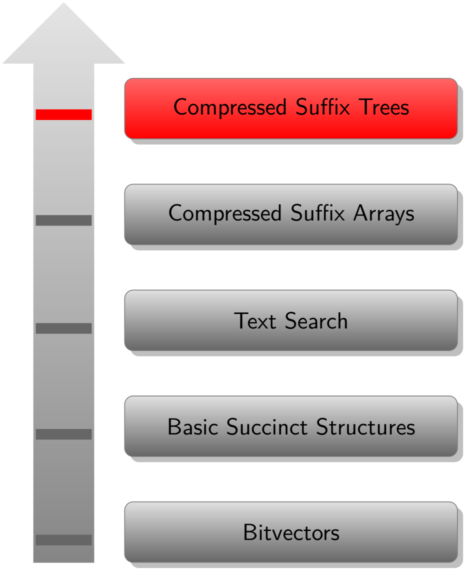
CSA - Usage Resources

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 Suffix Arrays
Text Search
Basic Succinct Structures
Bitvectors

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.[4]

Related, machine translation

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

[4]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 $P(w_i|w_1,\ldots,w_{i-1})$, often with a Markov assumption, i.e., $P \approx P^{(k)}(w_i|w_{i-k},\ldots,w_{i-1})$.

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

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

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

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:

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

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

Construction and querying

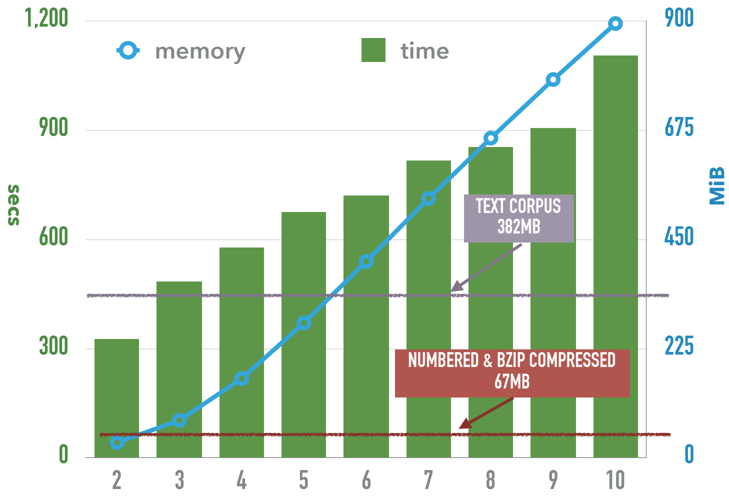
Probabilities computed ahead of time

- 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

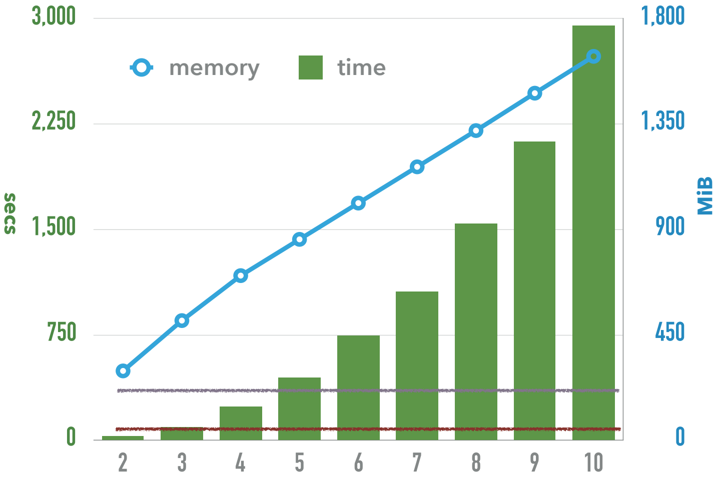
Querying

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

$T =$ 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

$T =$ 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

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

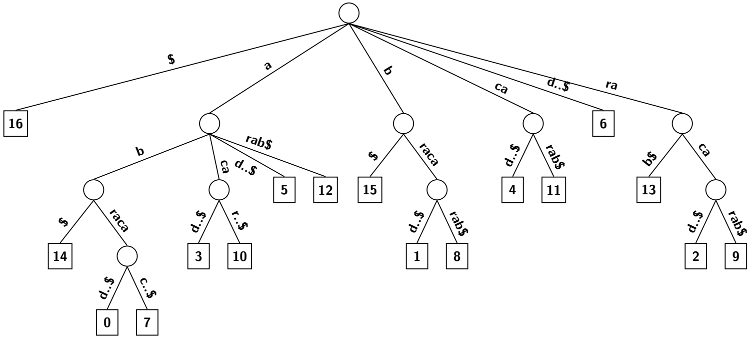
Sufficient statistics captured in suffix structures

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

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

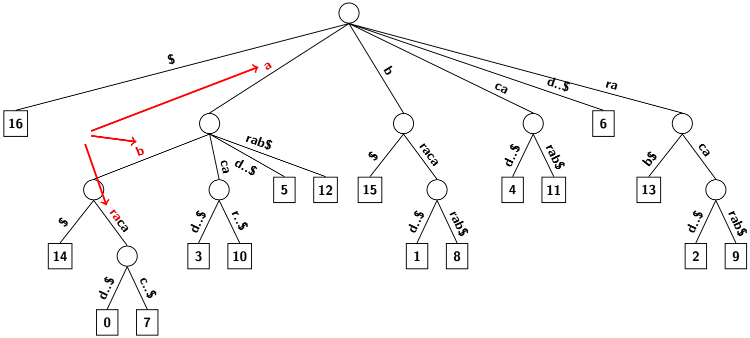
- $c(\texttt{abra}) = 2$ from CSA
  range between $lb = 3$ and $rb = 4$, inclusive
- $N_{1+}(\bullet\,\texttt{abra}) = 2$ from BWT (wavelet tree)
  size of set of preceeding symbols $\{\$, \text{d}\}$

Occurrence counts from the suffix tree



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

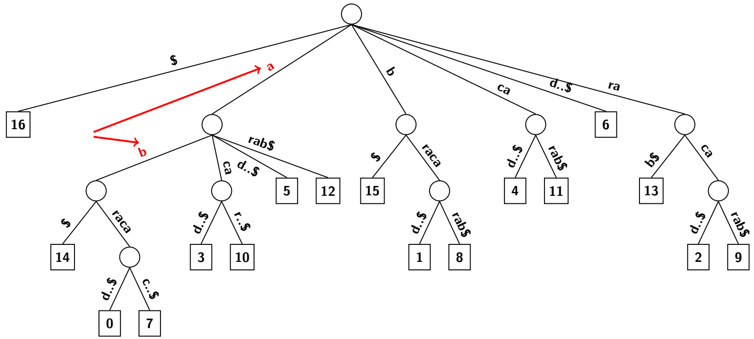
Occurrence counts from the suffix tree



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:

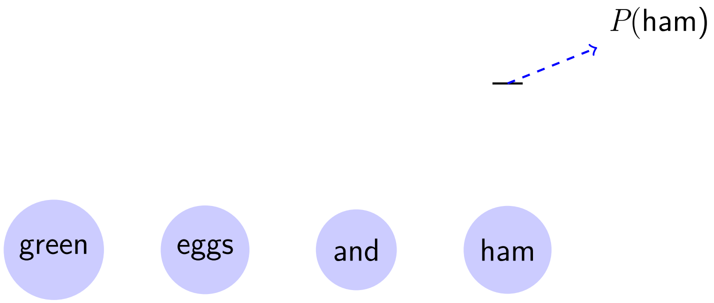
- count, $c(\alpha) = (rb - lb + 1)$
- occurrence count, $N_{1+}(\bullet\,\alpha) = \text{interval-symbols}(lb, rb)$

with time complexity

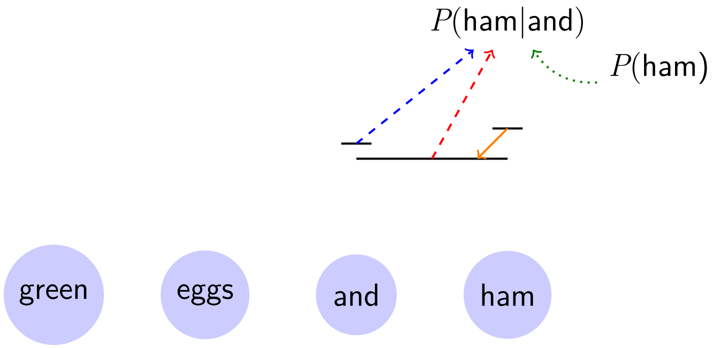
- $o(1)$; and
- $O(N_{1+}(\bullet\,\alpha) \cdot \log \sigma)$ where σ is the size of the vocabulary

What about the other required occurrence counts?

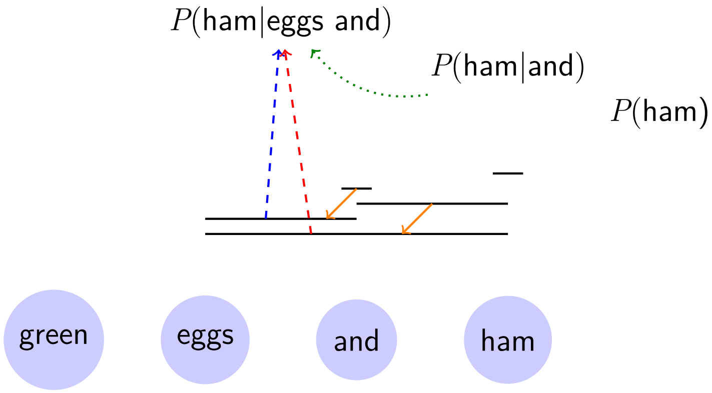
Querying algorithm: one-shot



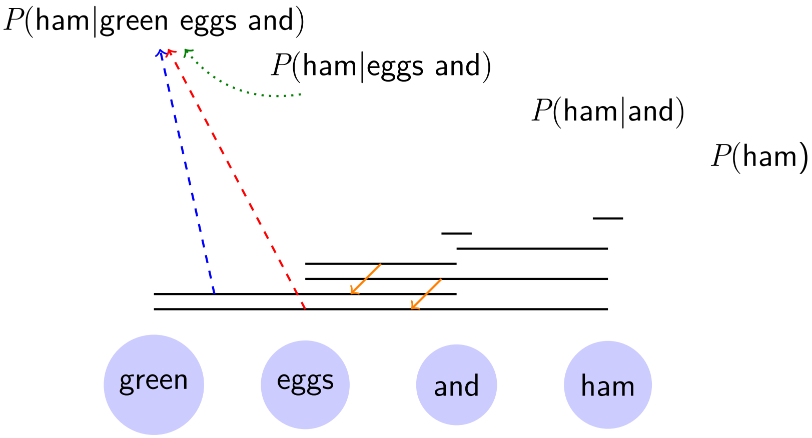
Querying algorithm: one-shot



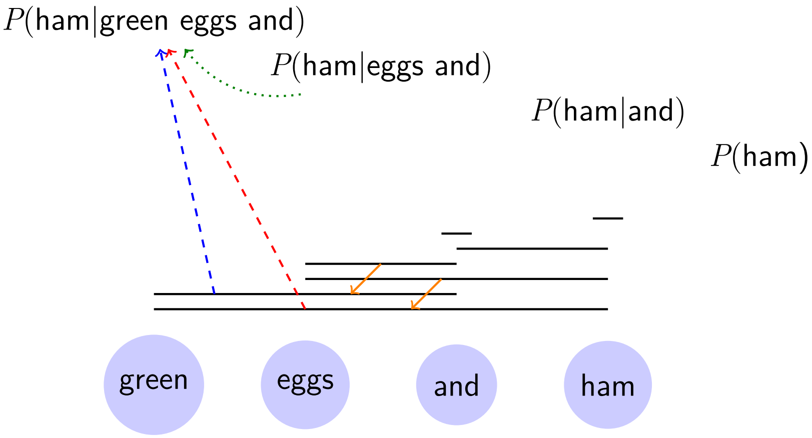
Querying algorithm: one-shot



Querying algorithm: one-shot


$P(\text{ham}|\text{green eggs and})$
$P(\text{ham}|\text{eggs and})$
$P(\text{ham}|\text{and})$
$P(\text{ham})$
green
eggs
and
ham

Querying algorithm: one-shot


$P(\text{ham}|\text{green eggs and})$
$P(\text{ham}|\text{eggs and})$
$P(\text{ham}|\text{and})$
$P(\text{ham})$
green
eggs
and
ham

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!

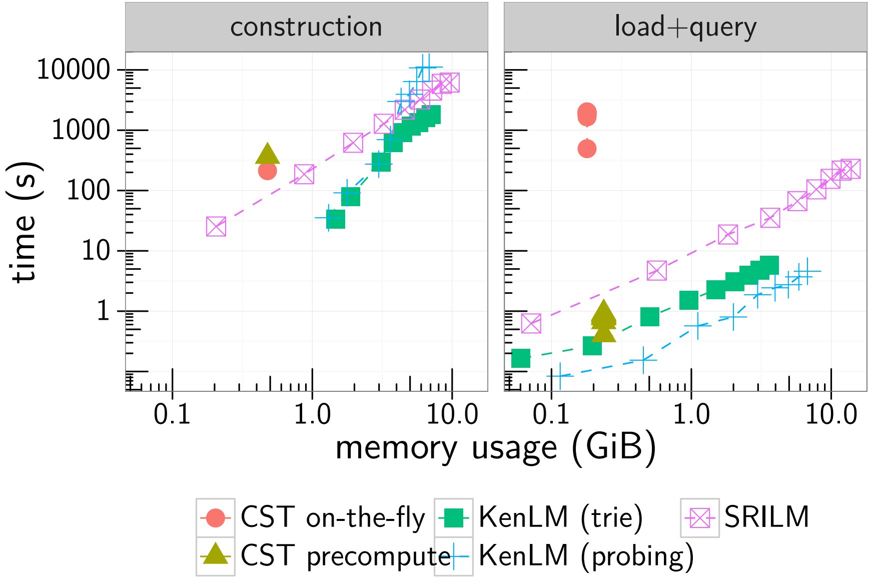
Technique

- 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 $v[\mathrm{rank}_1(bv, i)]$

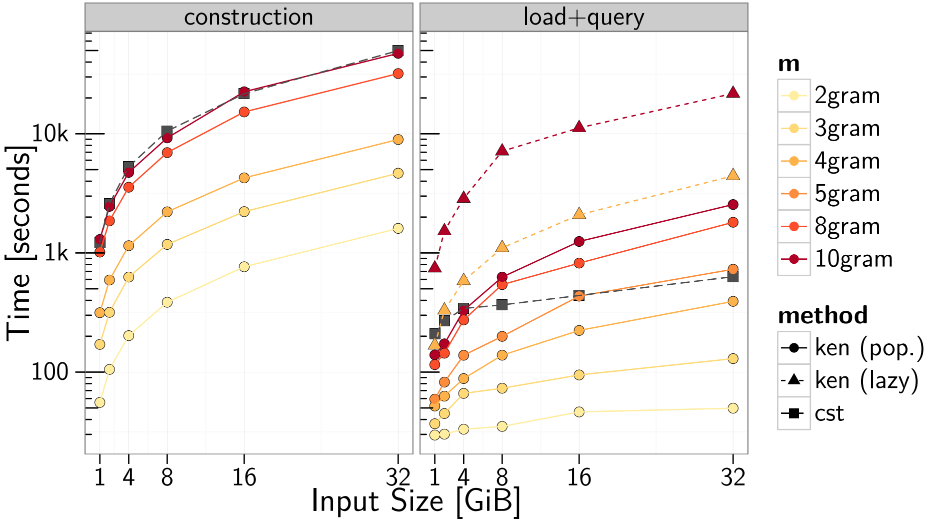
Effect of caching

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

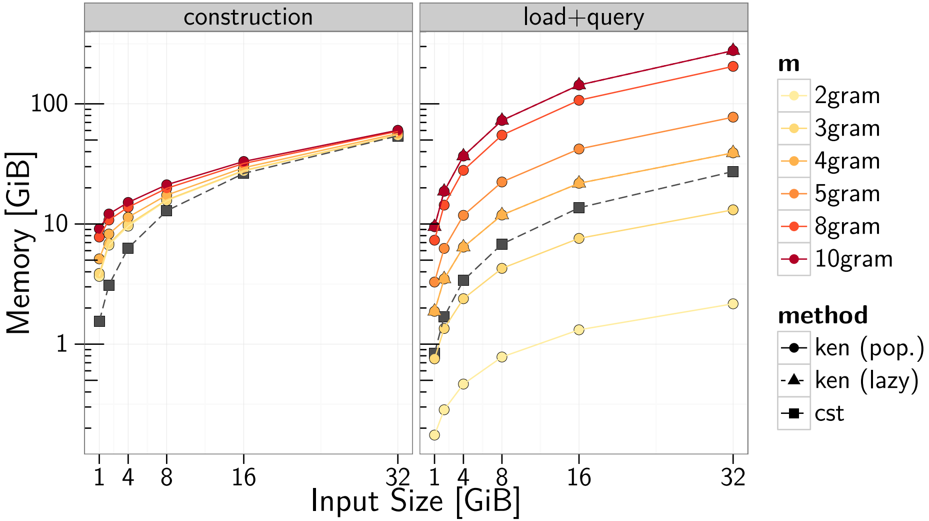
Timing versus other LMs: Small DE Europarl



Timing versus other LMs: Large DE Commoncrawl



Memory versus other LMs: Large DE Commoncrawl



Perplexity: usefulness of large or infinite context

newstest de corpus

Training	**size (M)** **tokens**	**sents**	**perplexity** $m=3$	$m=5$	$m=10$
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

Code example: `cst-csa-concordance.cpp`

Finding concordances for an arbitrary k-gram pattern:

Outline

- find count of k-gram in large corpus
- show tokens to left and to right, with their count
- find pairs of tokens occurring to left and right

How it works

- numbers words in corpus, builds a CSA & CST
- backward searching for pattern
- degree, edge etc calls to query next word to right
- querying WT for symbol to left

Code example: cst-csa-concordance.cpp

```
// map tokens to integers, and write to disk [snip]
// flip the vocabulary index [snip]

// construct a CST from the numbered file
typedef csa_wt_int<wt_huff_int<> > csa_type;
typedef cst_sct3<csa_type> cst_type;
cst_type cst;
construct(cst, "news-commentary-v11.en.numbered", 0);

// query the CSA to find the pattern "aspire to"
csa_type::size_type l=0, r=cst.csa.size()-1;
vector<uint64_t> pattern = { vocab_index["aspire"],
                             vocab_index["to"] };
bool ok = backward_search(cst.csa, l, r,
                          pattern.begin(),
                          pattern.end(),
                          l, r);
```

Code example: cst-csa-concordance.cpp

```
// report count and context to right
cout << "count("<<pattern_str<<"):␣" << (r-l+1) << endl;

// lookup corresponding node in CST, o(1)
const auto& node = cst.node(l, r);

// if pattern exactly matches path label (i.e., string depth = pattern size),
// then we can check the degree (#children) in the CST to find continuations to right
if (cst.depth(node) == pattern.size()) {
  // query symbols to the right
  cout << "word␣types␣on␣right:␣" << cst.degree(node) << endl;
  for (const auto& child: cst.children(node)) {
    // read off the first symbol on the path from the parent to the child
    auto symbol = cst.edge(child, pattern.size()+1); // this call can be expensive
    cout << "\tcount=" << cst.size(child) << "␣␣␣␣"
         << pattern_str<< "␣->␣'" << vocabulary[symbol] << "'" << endl;
  }
} else {
  // internal to an edge in the CST; only one continuation
  cout << "word␣types␣on␣right:␣1" << endl;
  auto symbol = cst.edge(node, pattern.size()+1);
  cout << "\tcount="<< cst.size(node) << "␣␣␣␣"
       << pattern_str<< "␣->␣'" << vocabulary[symbol] << "'" << endl;
}
```

Code example: `cst-csa-concordance.cpp`

```
// query symbols to the left
vector<uint64_t> syms(cst.csa.sigma);
vector<uint64_t> lbs(cst.csa.sigma);
vector<uint64_t> rbs(cst.csa.sigma);
csa_type::size_type num_to_left = 0;
interval_symbols(cst.csa.wavelet_tree, l, r+1, num_to_left, syms, lbs, rbs);
cout << "word types on left: " << num_to_left << endl;
for (auto i = 0; i < num_to_left; ++i) {
  cout << "\tcount="<< (rbs[i]-lbs[i]+1) << "    '"
      << vocabulary[syms[i]] << "' <- " << pattern_str  << endl;
}
```

...see code for matches of pair of symbols to left and right

...and `cst-csa-concordance-deep.cpp` which traverses CST to recover larger n-grams to right

External / Semi-External Suffix Indexes

String-B Tree [Ferragina and Grossi'99]

- Cache-Oblivious
- Uses blind-trie (succinct trie; requires verification step)
- Space requirement on disk one order of magnitude larger than text

Semi-External Suffix Array (RoSA) [Gog et al.'14]

- Compressed version of the String-B tree
- Replace blind-trie with a condensed BWT
- If pattern is frequent: Answer from in-memory structure (fast!)
- If pattern is infrequent: perform disk access

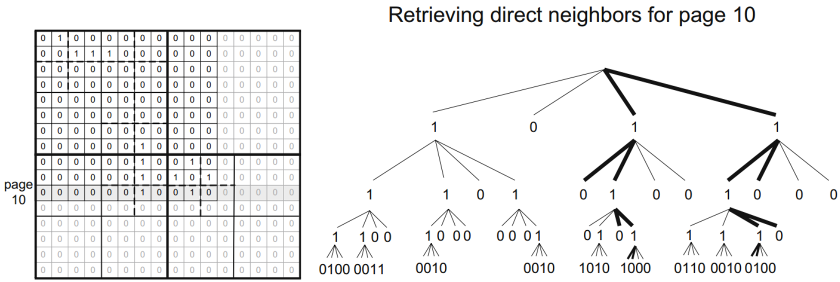
Range Minimum/Maximum Queries

- 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
- MARISA trie: https://github.com/s-yata/marisa-trie
- MARISA trie stats: File: all page titles of English Wikipedia (Nov. 2012) - Size uncompressed: 191 MiB, Trie size: 48 MiB, gzip: 52 MiB

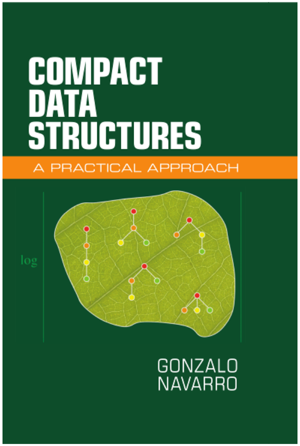
Graph Compression



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

Full-day tutorial at SIGIR 2016:

Succinct Data Structures in Information Retrieval: Theory and Practice
Simon Gog and Rossano Venturini
727 slides!
More extensive coverage of different succinct structures.

Materials: http://pages.di.unipi.it/rossano/succinct-data-structures-in-information-retrieval-theory-and-practice/

Resources III

- 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.