



On the use of Suffix Trees and Suffix Arrays for Lempel- Ziv Compression

Artur Ferreira

Supervisor Prof. Mário Figueiredo

Joint work with Prof. Arlindo Oliveira (INESC-ID)

Lisbon, 19 December 2008



Outline

1. Goals
2. Lempel-Ziv (LZ) compression family
3. Suffix Trees (ST) and Suffix Arrays (SA)
 - LZ encoding with ST and SA
 - Proposed Algorithms
4. Implementation Details
5. Experimental Results
6. Concluding Remarks

1. Goals

■ Motivation

- Lossless data compression is a large field of research
- Lempel-Ziv (LZ) or dictionary-based algorithms are widely used in this context, for *universal lossless data compression*
- The encoding algorithms need to store and search over a (large) dictionary

■ We want to answer the question:

“Are *Suffix Arrays* suited for LZ Compression?”

■ Propose *Suffix Array* based encoders and compare their performance with *Suffix Tree* based ones

2. Lempel-Ziv (LZ) Compression

- Lempel-Ziv is a family of lossless data compression algorithms
- Proposed by Abraham Lempel and Jacob Ziv

LZ1 Family

LZ77 – Lempel-Ziv 77

Proposed in 1977 by Jacob Ziv and Abraham Lempel

LZSS – Lempel-Ziv-Storer-Szymanski

Modification of LZ77, proposed in 1982, by James Storer and Thomas Szymanski

LZ2 Family

LZ78 – Lempel-Ziv 78

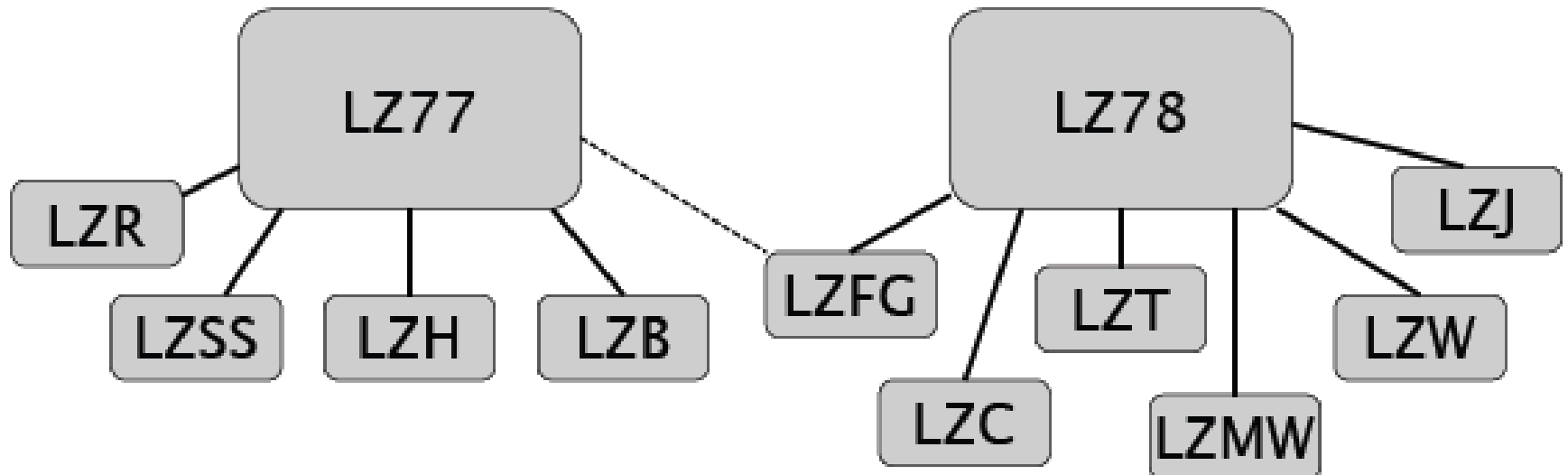
Proposed in 1978 by Jacob Ziv and Abraham Lempel

LZW - Lempel-Ziv-Welch

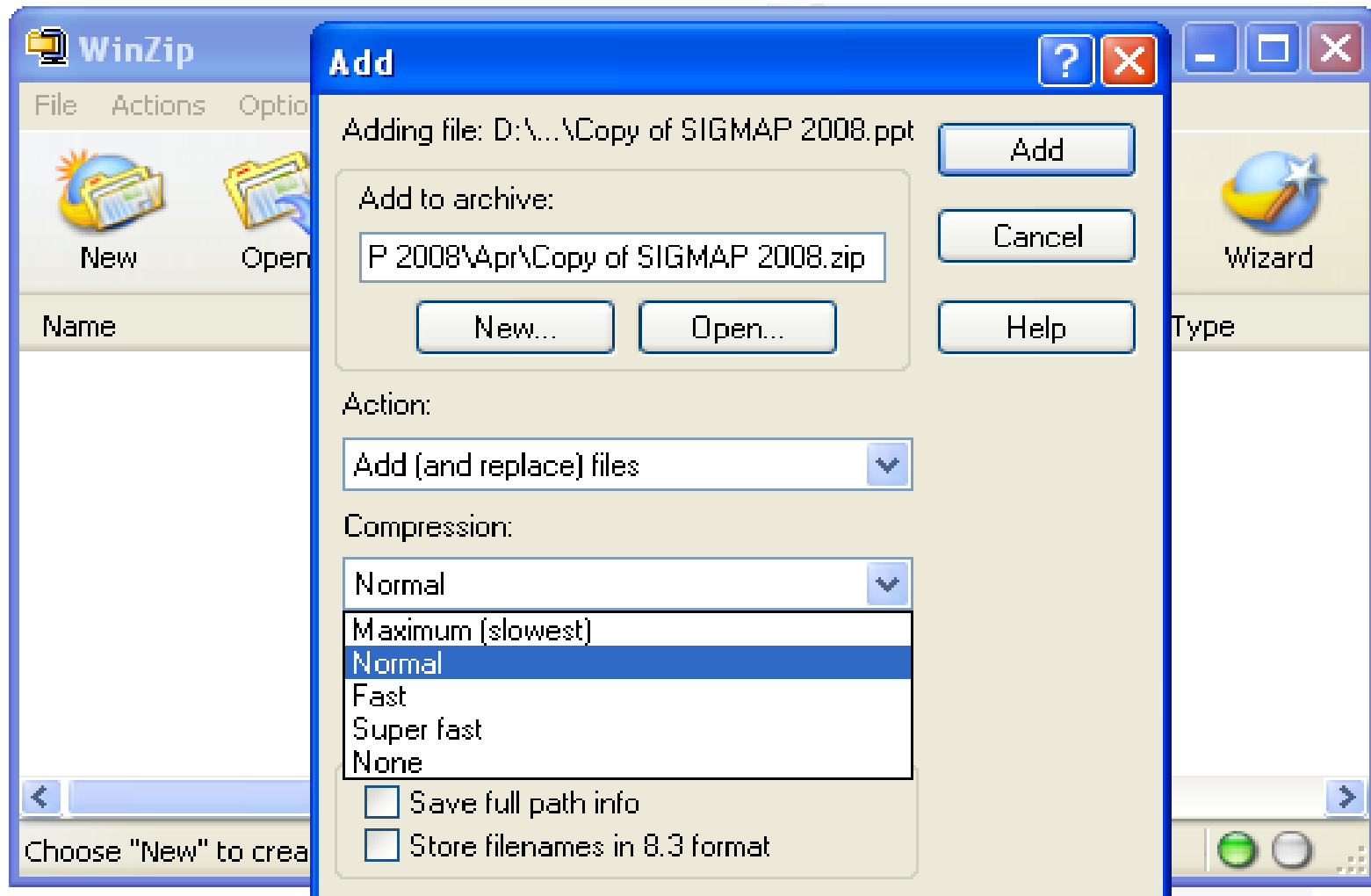
In 1984, Terry Welch proposed a modification of LZ78. Patent!!

2. Lempel-Ziv (LZ) Compression

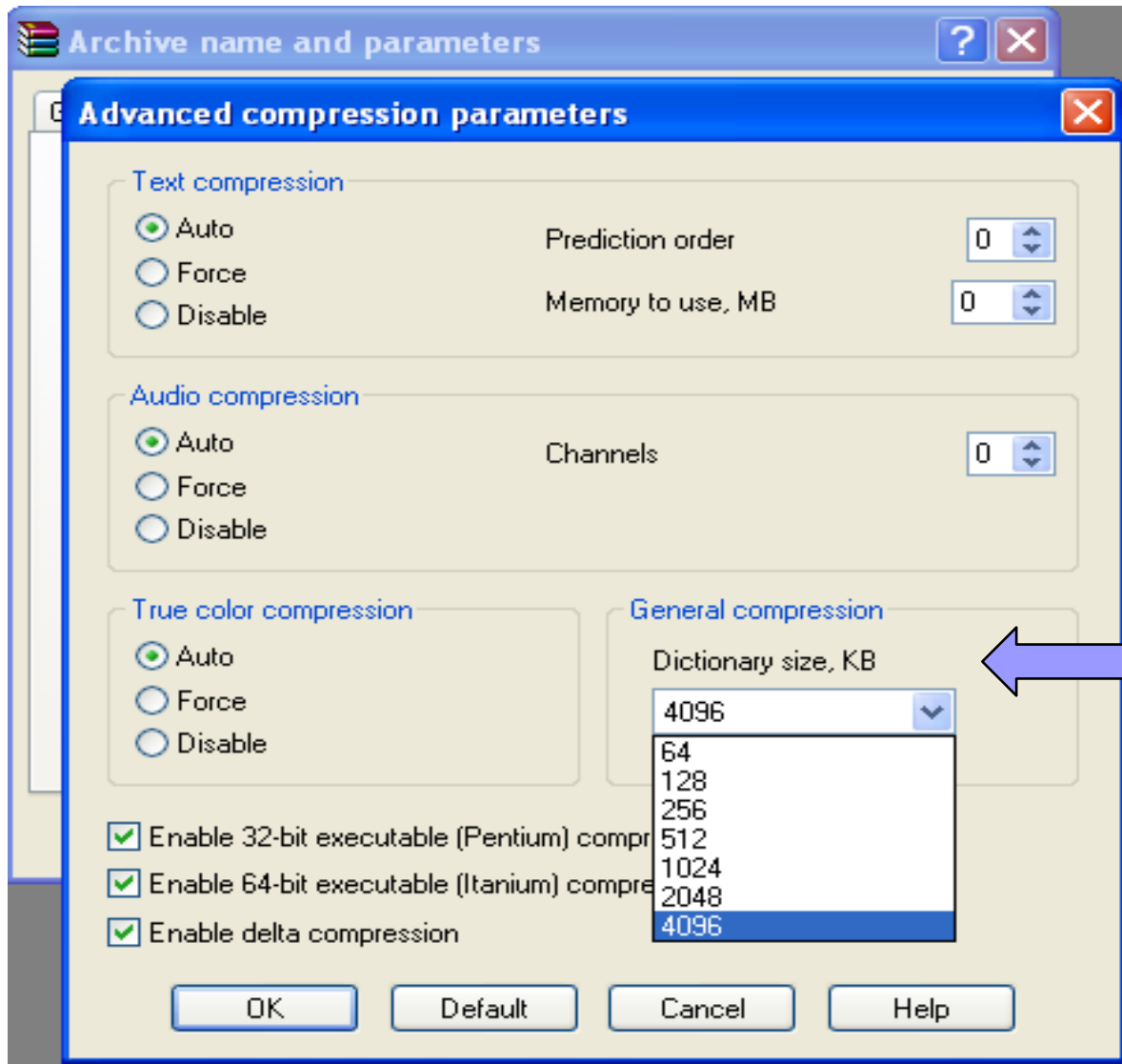
- LZ77 and LZSS
 - WinZip, WinRar, GZIP, Pkzip, Deflate, Inflate...
- LZ78 and LZW
 - GIF, PDF, V.42, Compress (Unix)



2. Lempel-Ziv (LZ) Compression



2. Lempel-Ziv (LZ) Compression



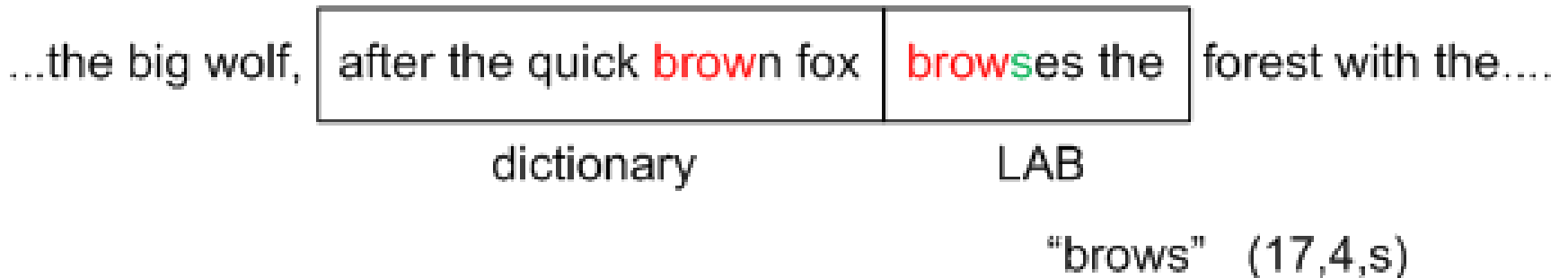
WinRAR

Dictionary Size

It is a compression parameter!

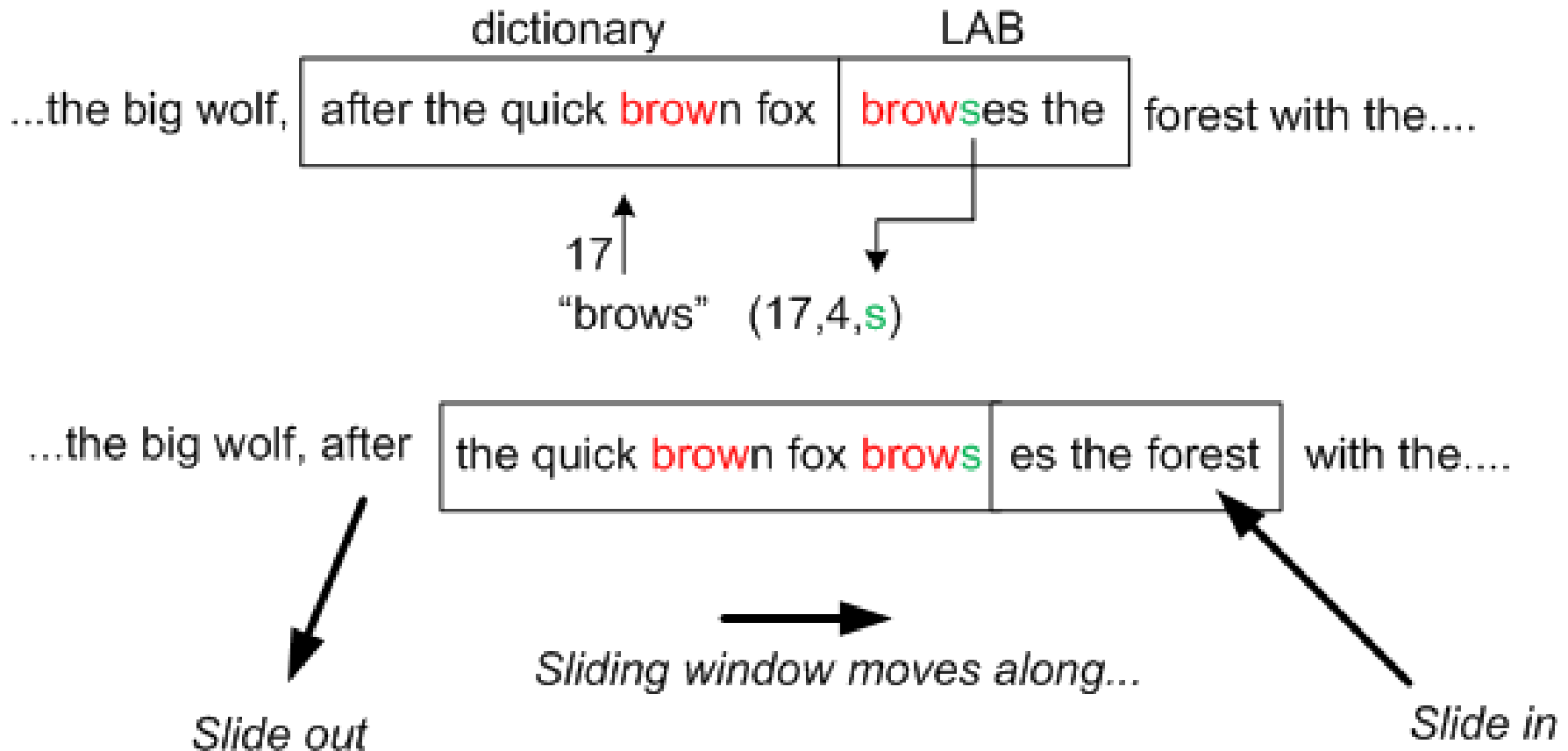
2. Lempel-Ziv (LZ) Compression

- The LZ77 encoder uses a *sliding window* over the text; the *sliding window* is divided into 2 components:
 - **dictionary** – the encoded text
 - **LAB** (look-ahead-buffer) – the text to encode
- The encoded text moves from the LAB to the dictionary



2. Lempel-Ziv (LZ) Compression

- Encoding of “brows”
- The beginning of LAB is searched over the dictionary



2. Lempel-Ziv (LZ) Compression

- LZ77 encoding produces tokens (**position, length, symbol**):
 - **position** – beginning of the sub-string in the dictionary
 - **length** – size of that sub-string
 - **symbol** – first character that breaks the match
- Each token occupies $\log_2(|Dictionary|) + \log_2(|LAB|) + 8$ bits
 - Better compression when we have large lengths
 - Poor compression when we don't have a match
- Typically, $|Dictionary| \gg |LAB|$
 - |Dictionary| in bytes: 128, 256, 512, 1024, 2048, 4096,....
 - |LAB| in bytes: 8, 16, 32,...

2. Lempel-Ziv (LZ) Compression

□ LZ77 encoding

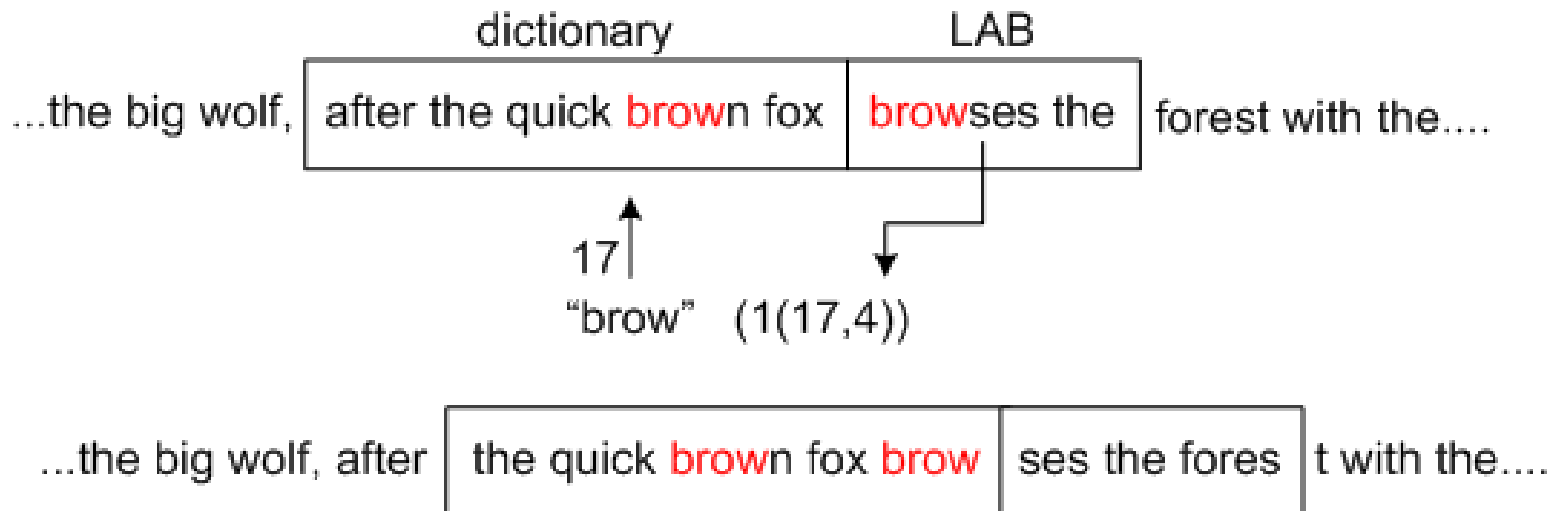
- involves finding the longest **sub-string** of the dictionary that matches the longest **prefix** of the LAB
- a token has the start of the sub-string (**position**) and its size (**length**)
- compression ratio is proportional to the length

□ LZ77 decoding

1. get the sub-string **s** in the dictionary, at index **position**, and with **length** characters
2. write sub-string **s|symbol** on the output
3. insert **s|symbol** at the end of the dictionary

2. Lempel-Ziv (LZ) Compression

- **LZSS** encoding produces tokens in the format **(flag code)**
 - **(1 (position, length))**, for sub-strings
 - **(0 (symbol))**, for a non-match character
- **LZSS decoding** is the same as LZ77 decoding, adapted to the format of the token



b)

2. Lempel-Ziv (LZ) Compression

- LZ77/LZSS decoding has no search! The decoder does not need any special data structure
- (Strong) asymmetry between encoder and decoder
 - **Typically: Encoding time >> Decoding Time**
- Data structures to hold the **encoder dictionary**, allowing fast search are necessary
 - **Suffix Trees (ST)** have been used (N. Larson)
 - We propose the use of **Suffix Arrays (SA)**

3. Suffix Trees and Suffix Arrays

■ Suffix Tree

- Definition and properties
- Use of Suffix Trees for LZ77 / LZSS **encoding**
- Existing algorithms

■ Suffix Arrays

- Definition and properties
- Use of Suffix Array for LZ77 / LZSS **encoding**
- Proposed Algorithms: A1 [1] and A2 [2]

[1] A. Ferreira, A. Oliveira, M. Figueiredo, “**Suffix Arrays: a competitive choice for fast Lempel-Ziv Compression**”, SIGMAP2008, Porto, Portugal, July.

[2] A. Ferreira, A. Oliveira, M. Figueiredo, “**On the Use of Suffix Arrays for Memory-Efficient Lempel-Ziv Data Compression**”, IEEE-Data Compression Conference, DCC2009, USA, March. Accepted.

3. Suffix Trees and Suffix Arrays

■ Suffixes of a string D

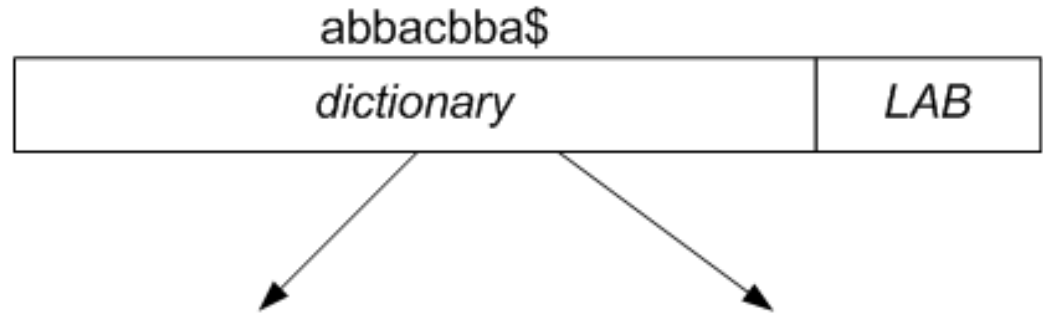
□ Let $D = \text{abbacbbba}\$$ be a string with length 9

□ The set of 9 suffixes of D is

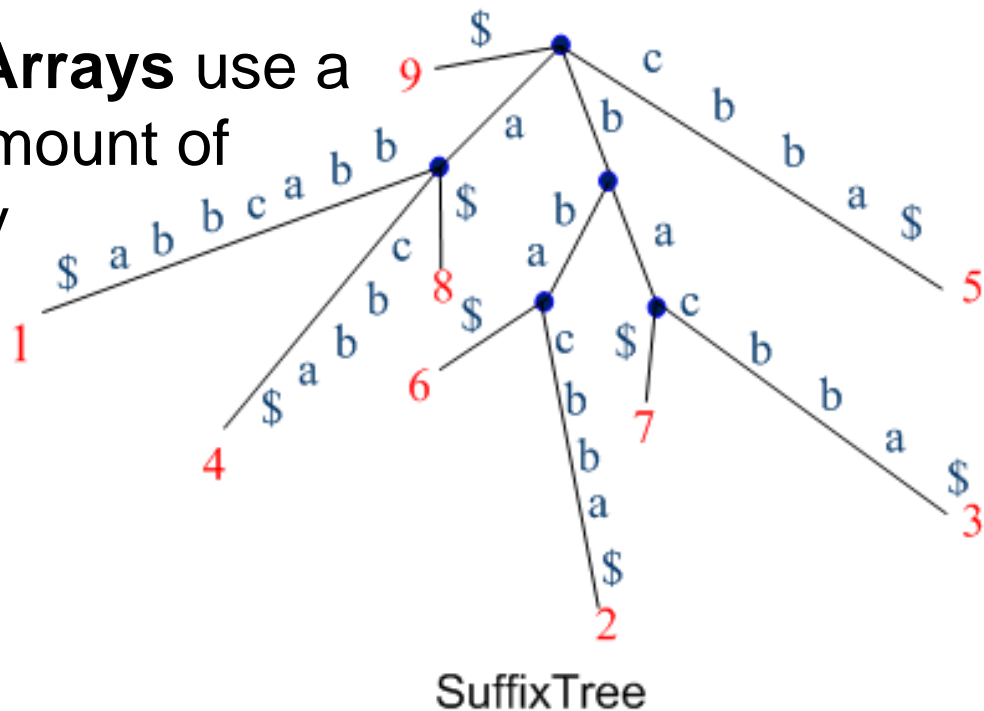
1. abbacbba\$
2. bbacbba\$
3. bacbba\$
4. acbba\$
5. cbba\$
6. bba\$
7. ba\$
8. a\$
9. \$

3. Suffix Trees and Suffix Arrays

- **Suffix Trees** suited for fast sub-string searching



- **Suffix Arrays** use a small amount of memory

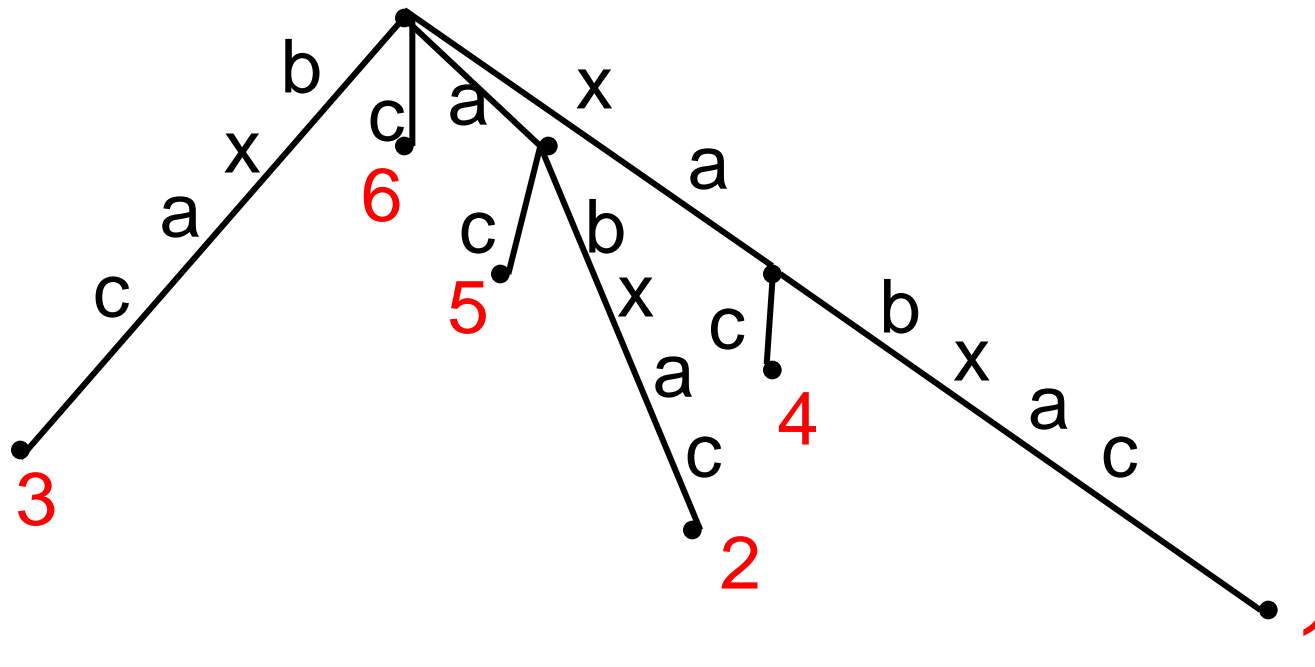


- 9. \$
- 8. a\$
- 1. abacbba\$
- 4. acbba\$
- 7. ba\$
- 3. bacbba\$
- 6. bba\$
- 2. bbacbba\$
- 5. cbba\$

Suffix Array

3. Suffix Trees and Suffix Arrays

- A **Suffix Tree** is built from a string D (1973, Weiner)
- It's a keyword tree - it keeps the set of suffixes of D
- Each leaf node has the suffix number
- Build in **linear time** with Ukkonen's algorithm



D = xabxac

1. xabxac
2. abxac
3. bxac
4. xac
5. ac
6. c

3. Suffix Trees and Suffix Arrays

LZ77 encoding with ST: algorithm

Input: D, dictionary, a m-character string.

S, n-character string to encode.

Output: LZ77 description of S.

1. Build an ST for D
2. Number each node v with cv , the smallest suffix number on v 's sub-tree; cv is the left-most position of D of any copy of the sub-string in the path from the root to v
3. To get the description (position, length) for sub-string $S[i \dots m]$:
 - a) Follow the (only) path matching the prefix of $S[i \dots m]$
 - b) Search ends at point p , with depth $depth(p)$
 - c) Output token (cv , $depth(p)$, $S[j]$), with $j=i+depth(p)$.

(position, length, symbol)

3. Suffix Trees and Suffix Arrays

Suffix Array (SA) (1991, Manber and Myers)

- Let D be a m -character string
- The SA of D , named P , is the array of integers from 1 to m , stating the lexicographical order of the suffixes of D
- For D =**mississippi** with 11 characters, we have P ={**11,8,5,2,1,10,9,7,4,6,3**}
- Can be computed from an ST by sorting

11. i
8. ippi
5. issippi
2. ississippi
1. mississippi
10. pi
9. ppi
7. sippi
4. sissippi
6. ssippi
3. ssissippi

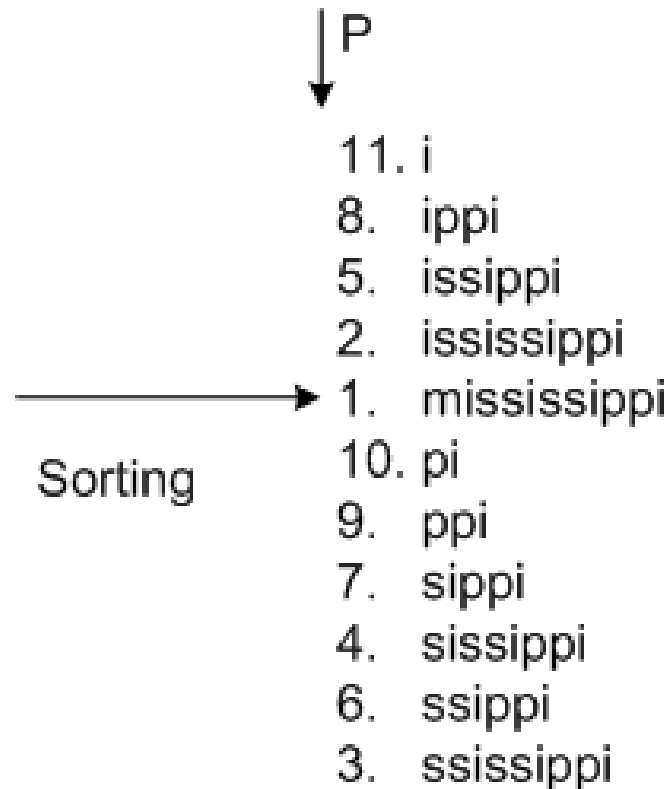
3. Suffix Trees and Suffix Arrays

Suffix Array: Graphical Idea

D=mississippi

1. mississippi
2. ississippi
3. ssissippi
4. sissippi
5. issippi
6. ssippi
7. sippi
8. ippi
9. ppi
10. pi
11. i

a)



b)

3. Suffix Trees and Suffix Arrays

Suffix Arrays

- Simple and space efficient
- Linear-time construction algorithms have been proposed
- Optimized construction algorithms for large alphabets
- Due to its simplicity, have been successfully used in:
 - search, indexing, plagiarism detection, information retrieval, biological sequence analysis
 - encoding data with anti-dictionaries

3. Suffix Trees and Suffix Arrays

LZ77 encoding with SA: Algorithm 1

Input: D, dictionary, a m-character string.

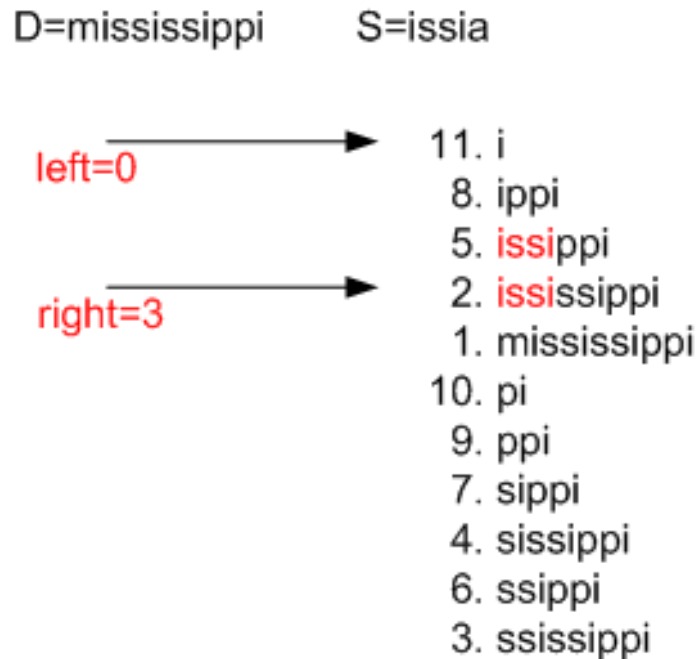
S, n-character string to encode.

Output: LZ77 description of S.

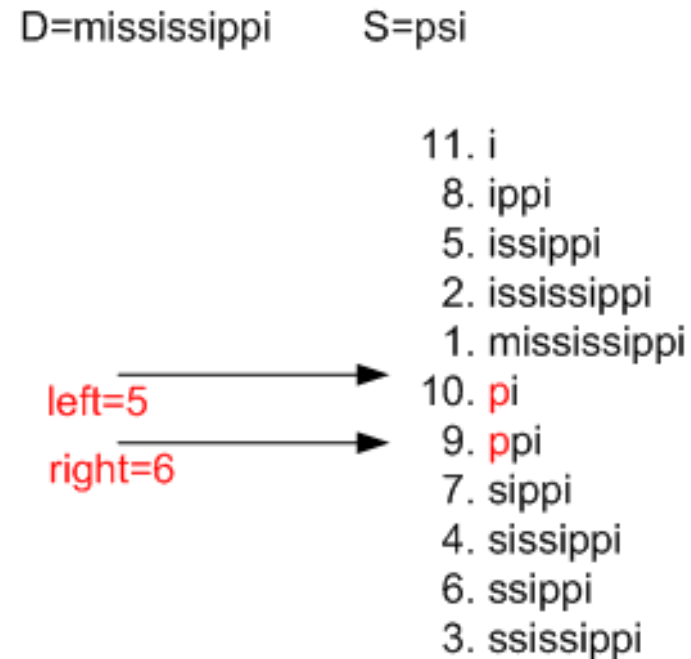
1. Build a SA for D, and named it P
2. To get the description for $S[i \dots n]$, search P until we find:
 - a) the first position *left* in which the first character of the suffix matches $S[i]$, that is, $D[P[\textit{left}]] = S[i]$;
 - b) the last position *right* which the first character of the suffix matches $S[i]$, that is, $D[P[\textit{right}]] = S[i]$;
 - c) if there is no match, output $(0,0,S[i])$ and move to $S[i+1]$.
3. From the set of suffixes between $P[\textit{left}]$ and $P[\textit{right}]$, choose $k \in [\textit{left}, \textit{right}]$, such that it matches the largest prefix of substring $S[i \dots n]$; let p be the length of the match
4. Output token $(k, p, S[j])$, with $j=i+p$.

3. Suffix Trees and Suffix Arrays

LZ77 encoding with SA: Algorithm1 - example



Token (5,4,a) or
(2,4,a)



Token: (10,1,s) (0,0,i)
or (9,1,s) (0,0,i)

- Perform binary search to find all occurrences of S in D

3. Suffix Trees and Suffix Arrays

LZ77 encoding with SA: Algorithm 2 (Use of LCP)

- Compute the SA (P) and LCP (L) for Dictionary|LAB
- P gives position and L gives length

Dictionary	LAB
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	17 18 19 20
t h i s _ i s _ t h e _ d i c t	_ t h e

Suffix Array P = {12, 5, 17, 8, 15, 13, 20, 11, 19, 10, 2, 14, 3, 6, 4, 7, 16, 18, 9}
pos

LCP L = {0, 1, 1, 4, 0, 0, 0, 1, 0, 2, 1, 0, 1, 3, 0, 2, 0, 1, 3, 2}

17 is the index of the text to encode

17 is located at position pos = 3 on SA P

Possible tokens

(P(2), L(3)) = (5,1) and (P(4), L(4)) = (8,4)

Longest Match

3. Suffix Trees and Suffix Arrays

LZ77 encoding with SA: Algorithm 2 (Use of LCP)

Input: D, dictionary, a m-character string.

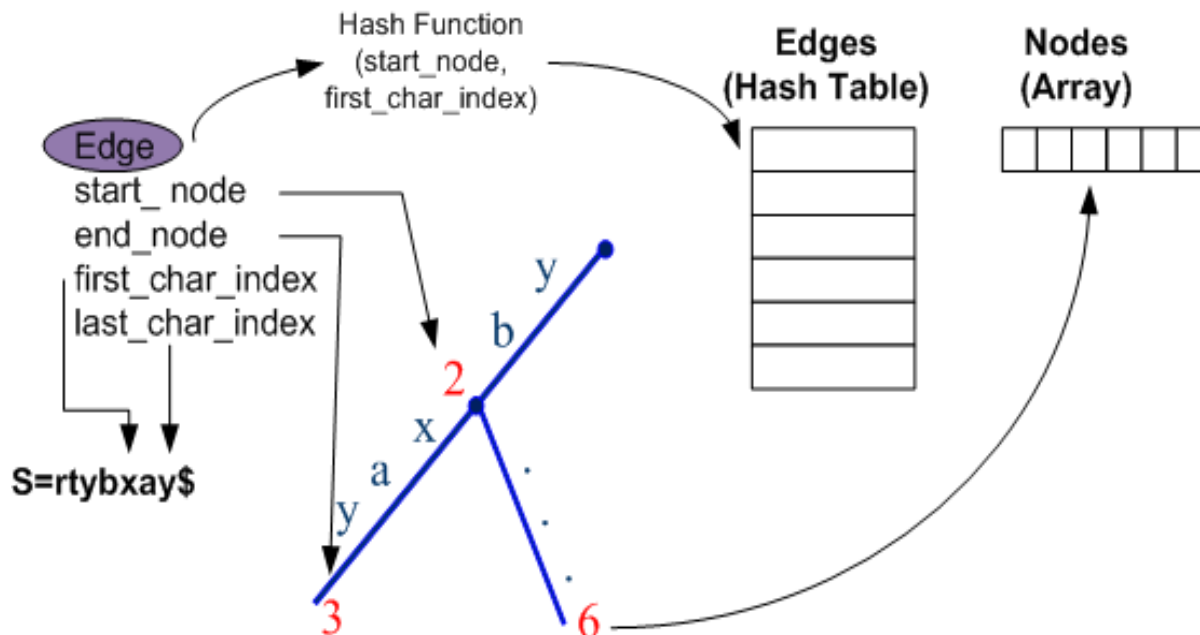
S, n-character string to encode.

Output: LZ77 description of S.

1. Compute SA for D|S and name it P and the LCP of P, named L
2. The description (pos,len) for every sub-string S[i ...n]:
 1. Locate index i over P; name it *pos*.
 2. Use P and L to obtain the two possible tokens $T_1=(P(pos-1),L(pos))$ and $T_2=(P(pos+1),L(pos+1))$;
 3. Validate tokens (see if the suffix number is in the dictionary area);
 1. if both valid, choose and output the one with the longest match
 2. if only one token is valid output it
 3. else (both invalid) output token (0,0,S[i])

4. Implementation Details

- The ST C++ software uses Ukkonen's algorithm
- Edges are stored in an Hash table
- Each edge has 4 integers
- Nodes are stored in an array



```
class Edge {
    int first_char_index;
    int last_char_index;
    int end_node;
    int start_node;
    // .....
};

class Node {
    int suffix_node;
    // .....
};

Edge Edges [ HASH_TABLE_SIZE ];
Node Nodes [ DICTIONARY_SIZE * 2 ]
```

4. Implementation Details

- The SA software is a set of 'C' functions

```
int sarray (int *a, int m);  
  
int *lcp (const int *a, const char *s, int m);  
  
.....
```

- We wrote code for:
 - Modifications on ST data structures (c_v counters)
 - Encoding algorithm with ST
 - Encoding algorithms with SA: A1 and A2

5. Experimental Results

- Test conditions:
 - 2 GB of RAM; Intel Core2 Duo CPU T7300 @ 2 GHz
 - Different size of dictionary and LAB
 - Set of 18 files from standard test sets (Calgary and Canterbury corpus)
- Measurements:
 - **Time [seconds]** = search and encoding time
 - **Memory [bytes]** = encoder data structures
 - ST = Branches + Nodes + Dictionary + LAB
 - SA:
 - A1 = Array + Dictionary + LAB
 - A2 = Array + LCP + Dictionary + LAB
 - **Compression [bpb]** = $8 * \text{Final} / \text{Initial}$

5. Experimental Results

18 standard test files from Calgary and Canterbury Corpus

File	Size [bytes]
bib	111261
book1	768771
book2	610856
news	377109
paper1	53161
paper2	82199
paper3	46526
paper4	13286
paper5	11954
paper6	38105

File	Size [bytes]
progc	39611
progl	71646
progp	49379
trans	93695
alice29	53161
asyoulik	82199
cp	46526
fields	13286
Total	2680580

5. Experimental Results

(Dictionary,LAB)=(128,16)

File	Suffix Tree			Suffix Array - A1		
	Time	Memory	Compression	Time	Memory	Compression
Fields.c (11 150)	0.30	6036	3.90	0.30	660	4.30
Progc (39 611)	1.10	5924	4.37	1.10	660	5.09
Paper1 (53 161)	1.60	4748	4.62	1.40	660	5.56
Progl (71 646)	2.20	5336	4.08	2.00	660	4.39
Paper2 (82 199)	2.60	5000	4.60	2.20	660	4.76
Alice29.txt (152 089)	4.50	5392	4.60	4.10	660	5.45

5. Experimental Results

(Dictionary,LAB)=(1024,256)

File	Suffix Tree			Suffix Array - A1		
	Time	Memory	Compression	Time	Memory	Compression
Fields.c (11 150)	0.16	45336	4.71	0.06	5380	4.07
Progc (39 611)	0.52	42088	4.91	0.20	5380	4.54
Paper1 (53 161)	0.77	45868	4.79	0.30	5380	4.54
Progl (71 646)	0.97	45616	4.05	0.36	5380	3.56
Paper2 (82 199)	1.30	42368	4.77	0.48	5380	4.58
Alice29.txt (152 089)	2.05	42788	4.77	0.92	5380	4.51

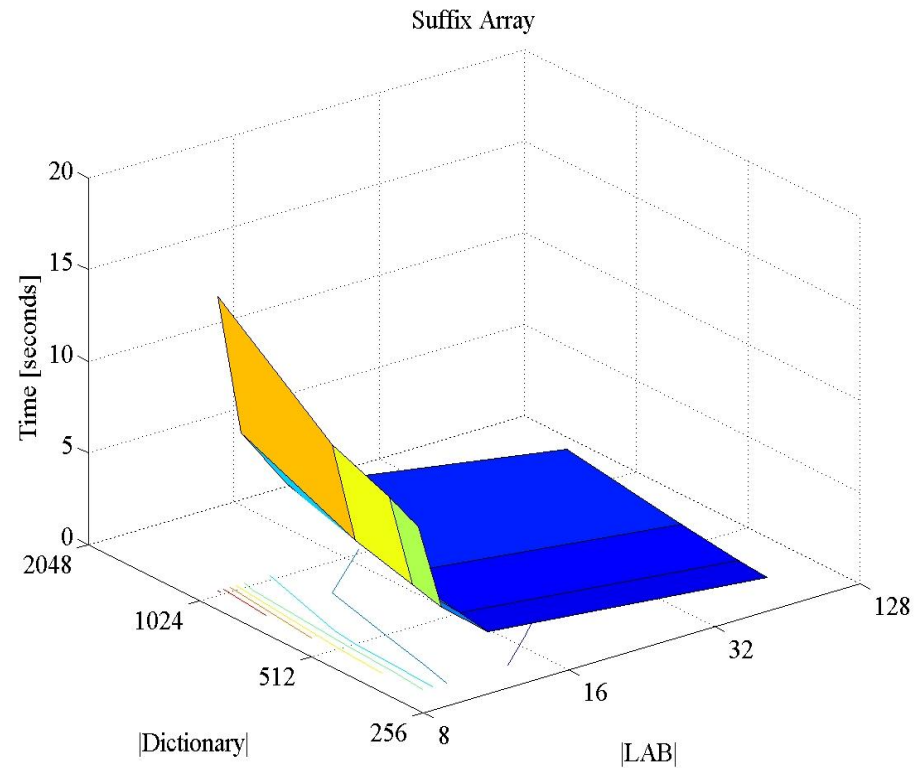
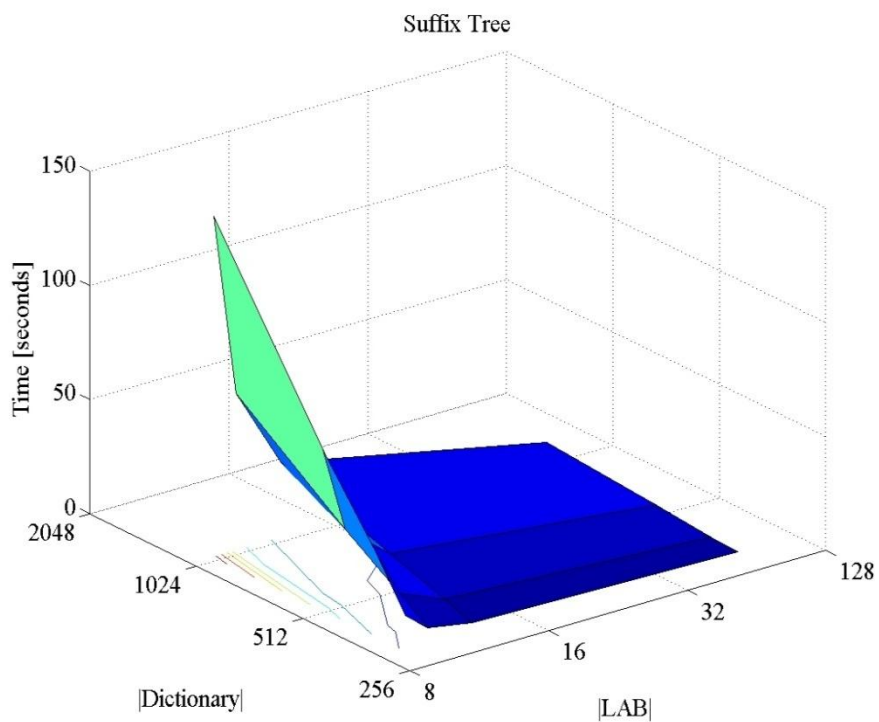
5. Experimental Results

Average time, memory and compression values for the set of 18 test files

Dict.	LAB	Suffix Tree			Suffix Array - A1		
		Time	Memory	Compression	Time	Memory	Compression
128	16	1.70	5440	4.53	3.90	660	5.40
256	16	7.10	11011	4.32	4.00	1300	4.67
1024	32	16.6	44132	4.21	3.00	5156	3.99
1024	256	2.10	44086	4.83	0.90	5380	4.53
2048	1024	1.10	90235	4.28	1.70	11268	5.00
4096	1024	2.20	181984	5.58	4.80	21508	5.09

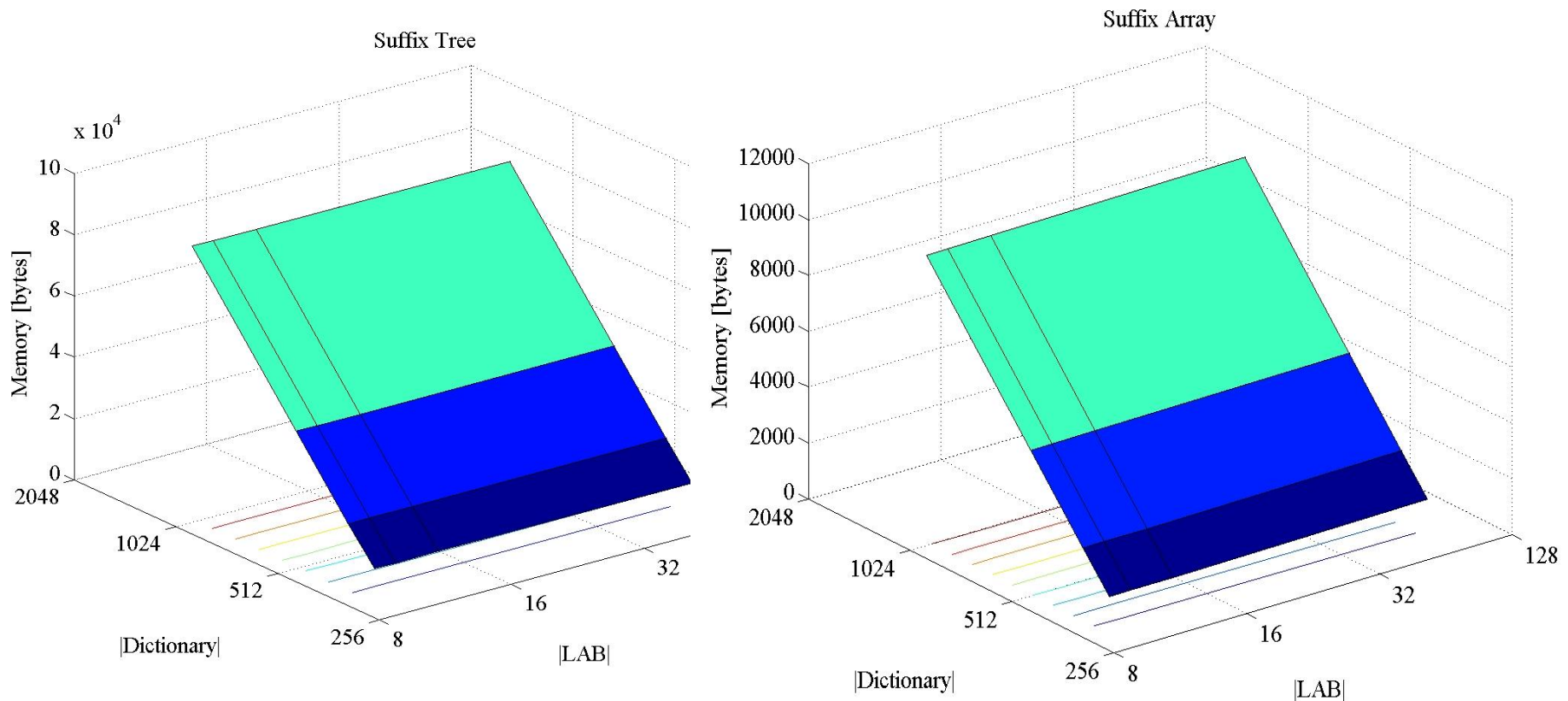
5. Experimental Results

Average encoding time (in seconds) as a function of the length of the dictionary and LAB, for ST and SA-A1 encoder



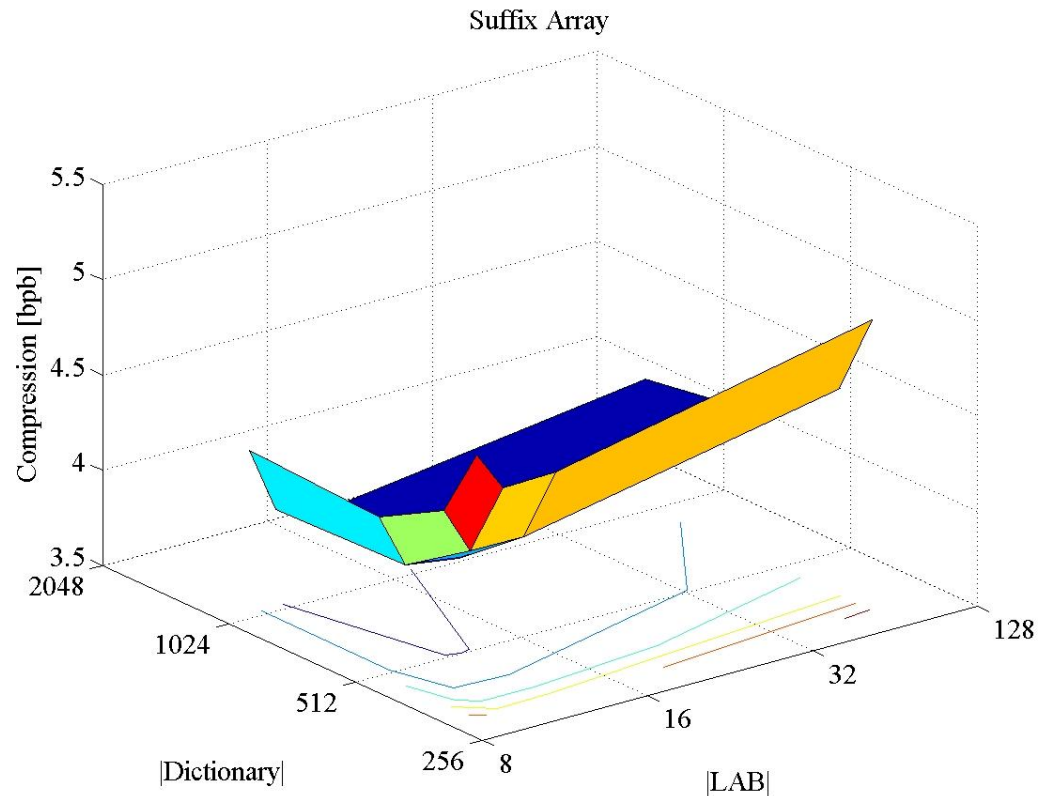
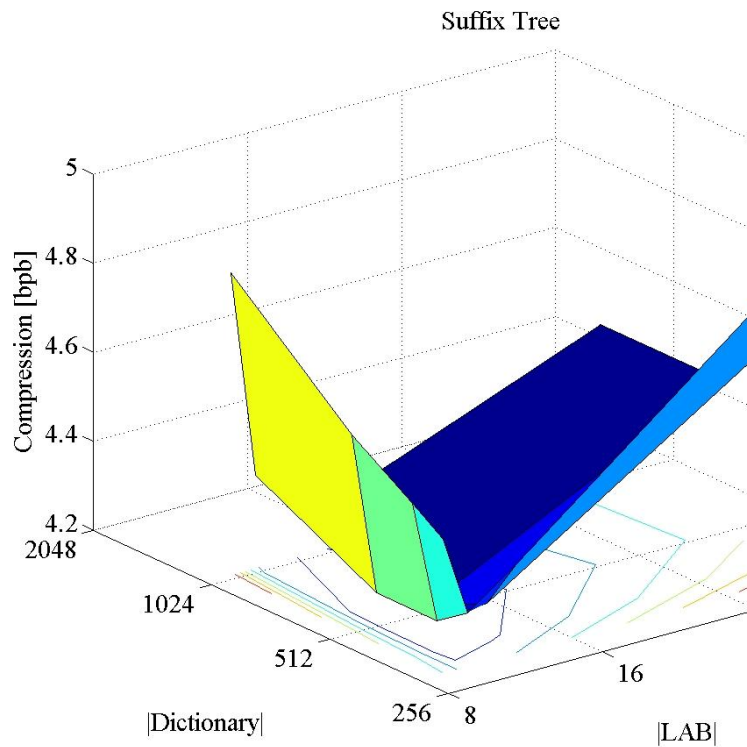
5. Experimental Results

Average memory (kB) as a function of the length of the dictionary and LAB, for ST and SA-A1 encoder



5. Experimental Results

Compression ratio (bpp) as a function of the length of the dictionary and LAB, for ST and SA-A1 encoder



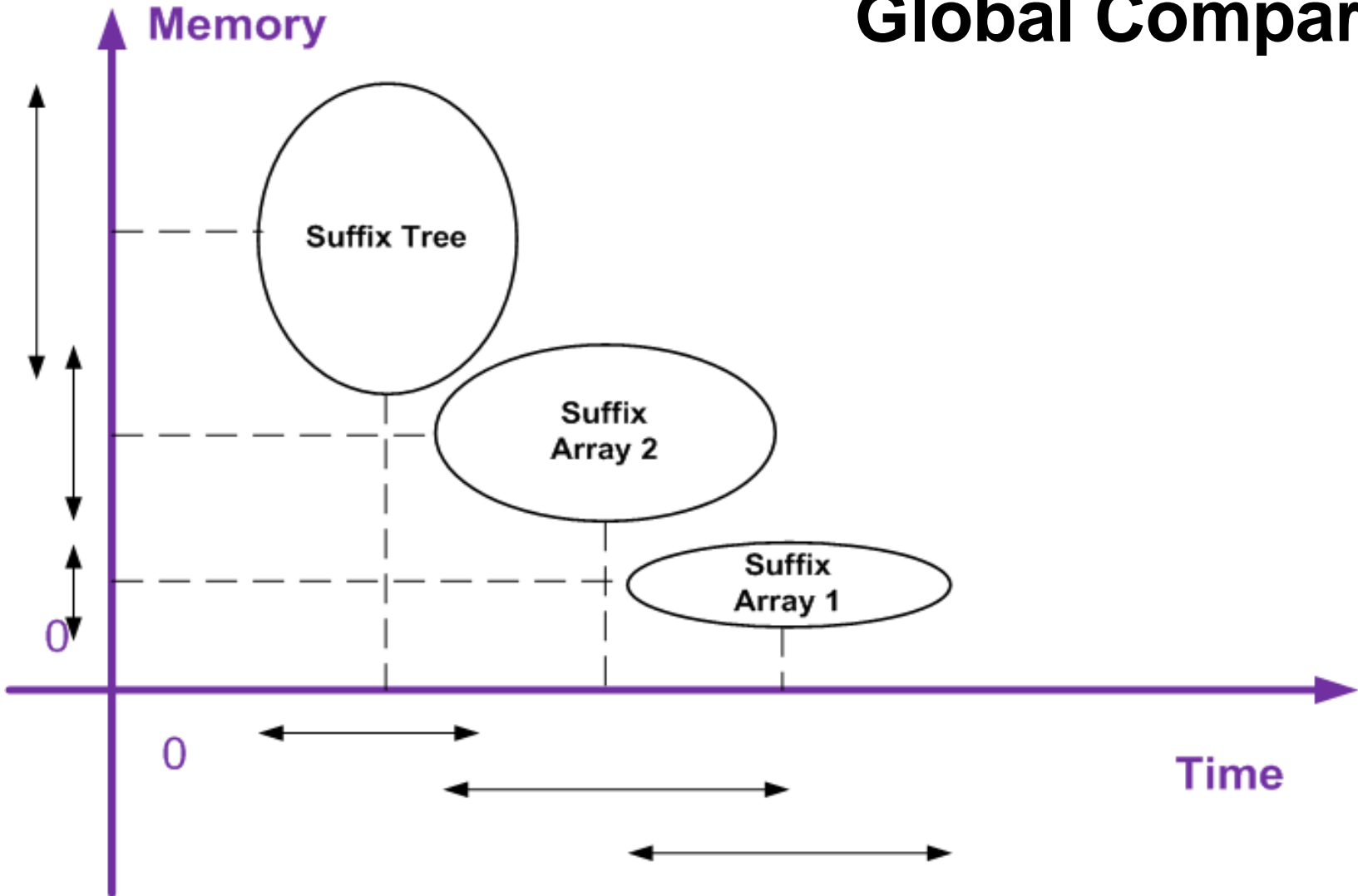
5. Experimental Results

(Dictionary,LAB)=(1024,128)

File	Suffix Array – A1			Suffix Array – A2		
	Time	Memory	Compression	Time	Memory	Compression
Paper5 (11 954)	0.14	3200	5.51	0.06	5248	5.68
Progl (71 646)	0.56	3200	3.91	0.42	5248	4.26
Paper2 (82 199)	0.78	3200	5.47	0.45	5248	5.55
Alice29 (152 089)	1.37	3200	5.41	0.86	5248	5.49
Lcet10 (426 754)	3.46	3200	5.45	2.46	5248	5.55
Plrabn12 (481 861)	4.53	3200	6.05	2.82	5248	6.10
Total	10,84			7,07		
Average			5,30			5,43

5. Experimental Results

Global Comparison



6. Concluding Remarks

- Encoding time grows linearly with the dictionary size
 - $ST < SA$, for small size dictionaries
 - $ST > SA$, for medium and large size dictionaries
- Memory
 - SA requires (much) less memory than ST
 - For ST, the amount of memory depends on the contents of the text; SA has (low) constant memory requirement
- Compression ratio
 - Roughly the same

6. Concluding Remarks

- SA is a good choice when we have:
 - small amount of memory on the encoding device
 - dictionaries larger than roughly 1 kB
- SA allows fast search with constant low-memory requirements
 - Competitive choice for embedded systems!
- Future work
 - Improve SA-encoder A2

References

- [1] A. Ferreira, A. Oliveira, M. Figueiredo, “**Suffix Arrays: a competitive choice for fast Lempel-Ziv Compression**”, SIGMAP2008, July 2008, Porto, Portugal.

- [2] A. Ferreira, A. Oliveira, M. Figueiredo, “**On the Use of Suffix Arrays for Memory-Efficient Lempel-Ziv Data Compression**”, IEEE-Data Compression Conference, DCC2009, March 2009, SnowBird, Utah, USA. Accepted.

C++ source code for Suffix Trees:

<http://marknelson.us/1996/08/01/suffix-trees/>

C source code for Suffix Arrays:

<http://www.cs.dartmouth.edu/~doug/sarray/>