The imagination driving Australia's ICT future.

NATIONAL ICT AUSTRALIA

XML Tree Structure Compression

Sebastian Maneth

NICTA & University of NSW

Joint work with N. Mihaylov and S. Sakr

Melbourne, Nov. 13th, 2008

Australian Government

Department of Communications, Information Technology and the Arts

Australian Research Council

NICTA Members

NICTA Partners

Outline -- XML Tree Structure Compression

1. Motivation

2. XMill's compression of XML tree structure

3. Pattern based tree compression

→ DAGs
→ sGraphs (= Straight Line cf Tree grammar)

4. Binary coding

5. Some algorithms on SLT grammars

1. Motivation

→ large part of an XML document consists of markup in the form of begin and end-element tags, describing the tree structure of the document

→ most XML file compressors **separate** the tree structure from the rest of the document (data values) and **compress them separately**

(for data values, classical compression methods can be used)

In this work

→ want to find effective (file) compression method for the tree structure of an XML document

2. XMill

Well-known XML file compressor: XMill [Liefke, Suciu, SIGMOD 2000]

Idea → separate data values from tree structure
→ group similar data items together into containers
(similarity is based on tree structure path to the item)
→ compress all containers using conventional compression
backends, such as Gzip/Bzip2/PPM

How is the tree structure compressed?

Use (byte-aligned) symbols per each begin-element tag, and one fixed symbol for every end-element tag.
Compress result string using Gzip/Bzip2/PPM

2. XMill

How is the tree structure compressed?

Example

<book>
<chapter></chapter>
<chapter><section/><section/><section/></chapter>
<chapter><section/><section/></chapter>
</book>

End element tag: /

Becomes

0 1 / 1 2 / 2 / 2 / / 1 2 / 2 / / /

Plus the symbol table ["book", "chapter", "section"]

0 1 2

Compress using Gzip/Bzip2/PPM

3. Our Approach: Sharing of Tree Patterns

Use **in-memory (pointer-based) tree compression,**
& write *suitable binary encoding* to disk (possibly plus Gzip/Bzip2/PPM backends)

Pointer-based tree compressions considered:

1) **DAGs** (Directed-Acyclic Graphs)

→ obtained by sharing *common subtrees* of the XML tree structure
use standard algorithm based on hashing distinct subtrees

2) **Sharing graphs** [Lamping, POPL 1990]

→ obtained by sharing *common connected subgraphs* in XML tree
use *BPLEX algorithm* [Busatto, Lohrey, Maneth, DBPL 2005]

3. Our Approach: Sharing of Tree Patterns

Pointer-based tree compressions considered:

1) **DAGs** (Directed-Acyclic Graphs)

→ share *common subtrees*
use standard algorithm
(lin time: [Downey, Sethi, Tarjan 1980])

- minimal DAG is unique
- can be computed in (amortized) linear time -- folkore ("hash consing")
- same as minimal tree automaton for {t}

2) **Sharing graphs** [Lamping, POPL 1990]

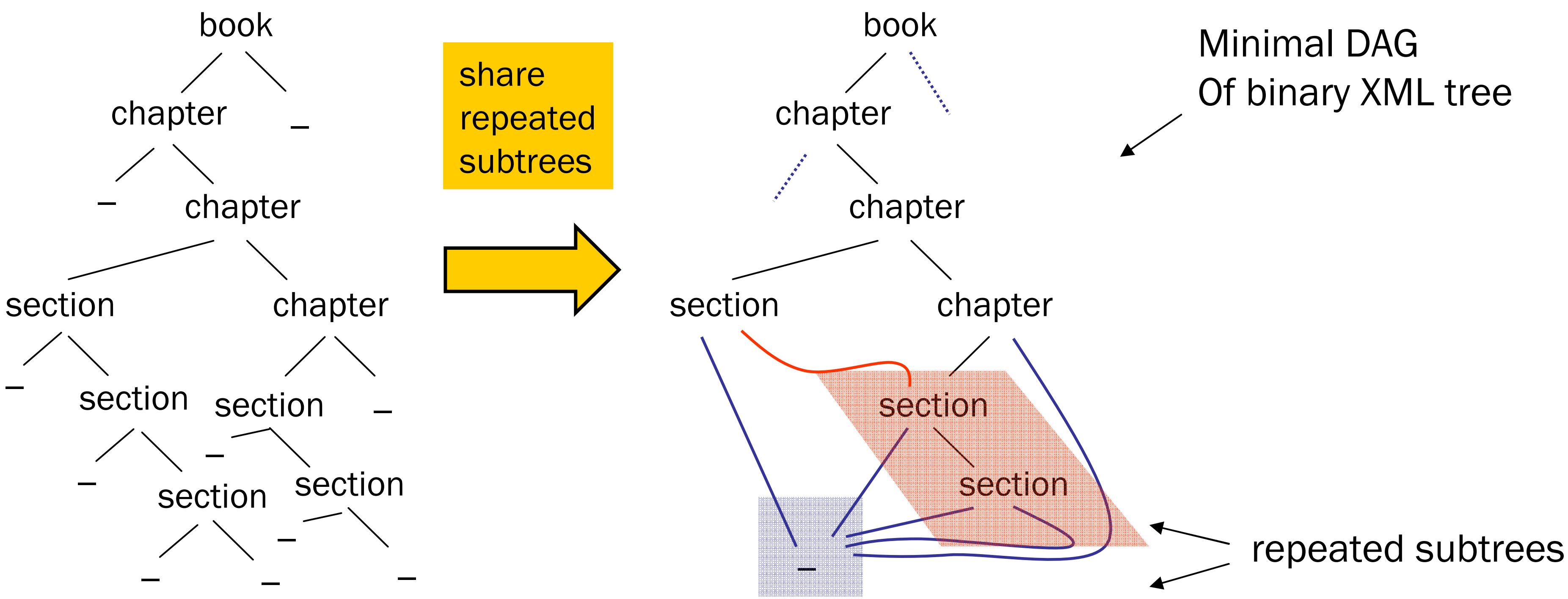
→ share *common connected subgraphs*
→ use *BPLEX algorithm*
[Busatto, Lohrey, Maneth, DBPL 2005]

- minimal sGraph not unique
- NP-complete to compute it (as finding a minmal cf grammar for a string)
- same as minimal cf tree gramar for {t}

Minimal DAG

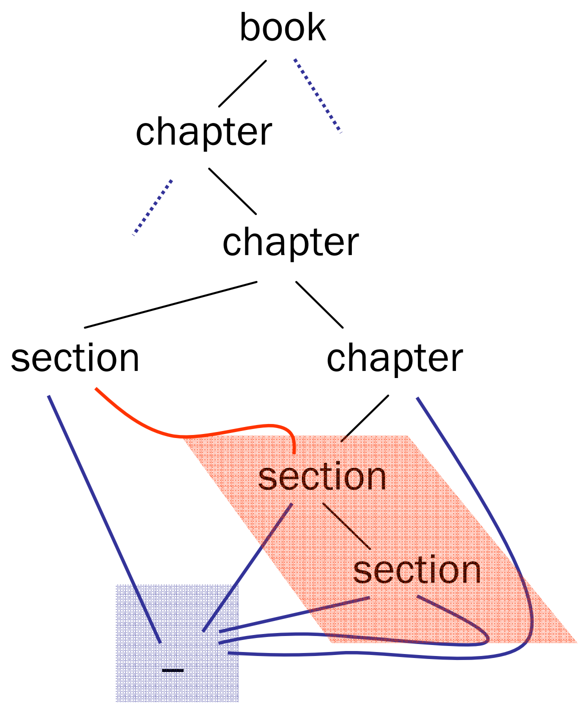
1) **DAGs** (Directed-Acyclic Graphs)
→ obtained by sharing *common subtrees* of the XML tree structure

Example --- working on *binary XML tree* (first-child/next-sibling encoding)



NATIONAL ICT AUSTRALIA

Minimal DAG



(0 : _
1 : section[0, 0]
2 : section[0, 1]
3 : book[chapter[0, chapter[section[0, 2], chapter[2, 0]]], 0])

Sequential representation of minimal DAG

Use own ***suitable binary endocing*** → DAG

Gzip → DAGGzip

Bzip2 → DAGBzip2

PPM → DAGPPM

Final compressed codewords: DAG, DAGGzip, DAGBzip2, DAGPPM

Minimal DAG

→ Test DAG,DAGGzip,DAGBzip2,DAGPPM on diverse XML dataset:

including

* files used by Liefke/Suciu for XMill
* several Wikipedia XML files
* files from EXI W3C working group

Etc.

Documents used in Experiments

Document	Size (KB)	Tags	# Nodes	Depth
1998statistics.xml	717	47	54,581	7
Catalog-01.xml	6,624	51	372,459	9
Catalog-02.xml	65,875	51	3,705,071	9
Dictionary-01.xml	3,481	25	513,574	9
Dictionary-02.xml	34,311	25	5,077,549	9
EnWikiNew.xml	7,834	21	665,825	6
EnWikiQuote.xml	5,034	21	437,682	6
EnWikiSource.xml	21,849	21	1,902,189	6
EnWikiVersity.xml	9,530	21	828,229	6
EnWikTionary.xml	160,373	21	14,520,656	6
EXI-Array.xml	7,156	48	226,524	10
EXI-Factbook.xml	2,087	200	86,581	6
EXI-Invoice.xml	457	53	26,130	8
EXI-Telecomp.xml	5,402	39	177,634	7
EXI-Weblog.xml	2,216	13	178,375	4
JST_gene.xml	7,932	27	388,029	8
JST_snp.xml	24,667	43	1,169,686	9
Lineitem.xml	30,270	19	1,985,776	4
Medline.xml	80,248	79	5,394,921	8
Mondial.xml	409	23	22,423	5
Nasa.xml	9,958	62	792,467	9
NCBI_gene.xml	13,042	51	645,917	8
NCBI_snp.xml	135,853	16	6,879,757	5
Sprot.xml	206,993	49	21,634,330	7
Treebank.xml	31,450	252	3,843,775	38
USHouse.xml	144	44	11,889	17

→ Size (KB) means
XML tree structure only.

Original files are much larger:
457MB (Sprot.xml)
190MB (NCBI_snp.xml) etc

Note

→For each text and attribute
node we have a special place
Holder node in the tree structure.

Minimal DAG

→ Test `DAG,DAGGzip,DAGBzip2,DAGPPM` on diverse XML dataset:

including
* files used by Liefke/Suciu for XMill
* several Wikipedia XML files
* files from EXI W3C working group

Etc.

Most important observation:

Minimal DAG does **not** give best compression!

→ Only share subtrees of a *certain size* (more than TRESH-many nodes)

Minimal DAG

→ Test DAG,DAGGzip,DAGBzip2,DAGPPM on diverse XML dataset:

including
  * files used by Liefke/Suciu for XMill
  * several Wikipedia XML files
  * files from EXI W3C working group
Etc.

Optimal TRESH-values for our datasets:

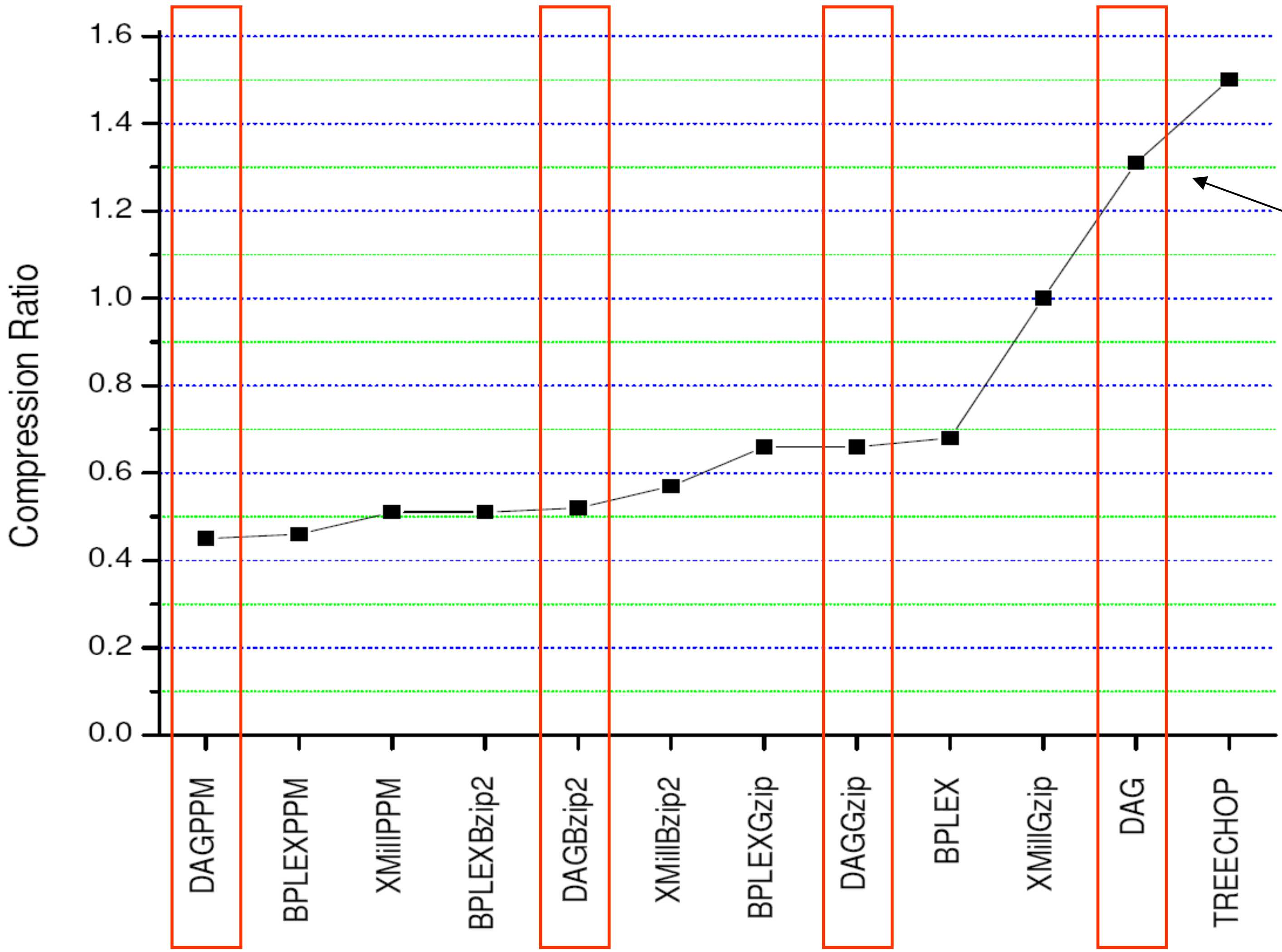
TRESH=14 for DAG
TRESH=1000 for DAGGzip
TRESH=3000 for DAGBzip2 and DAGPPM

Most important observation:

Minimal DAG does **not** give best compression!

→ Only share subtrees of a *certain size* (more than TRESH-many nodes)

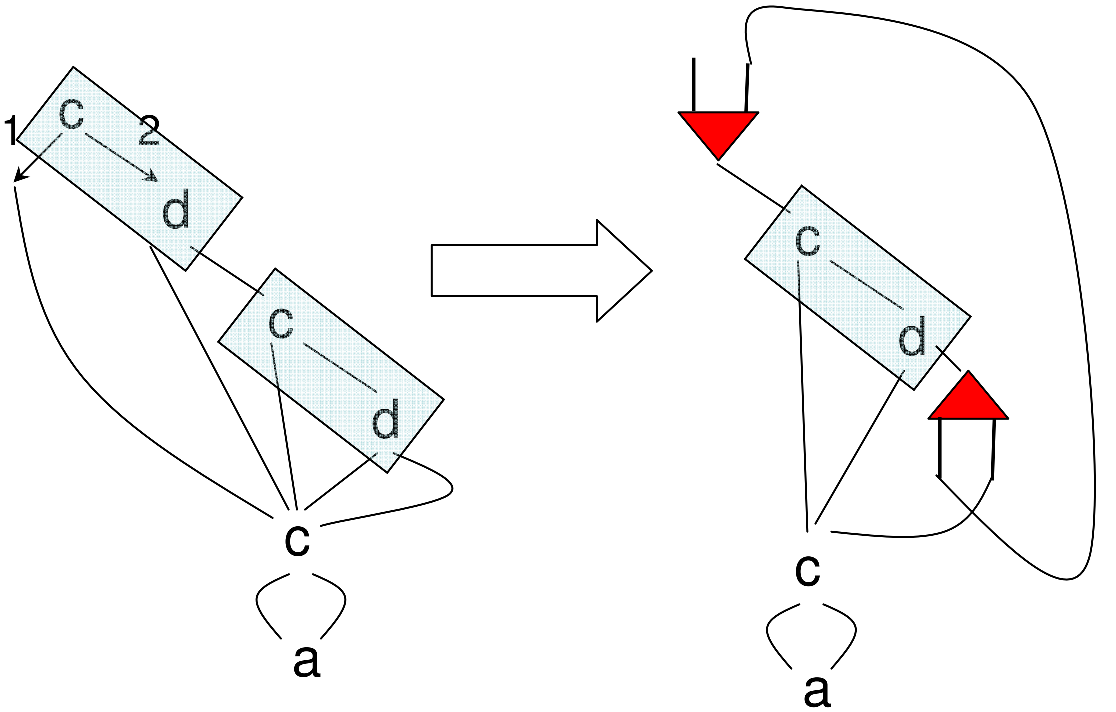
DAGs, Results



With no backend compression, **DAG** gives *worse* compression than **XMillGzip** (=baseline = 1.0)

Sharing Graphs (SLT grammars)

Idea, share repeated (connected) subgraphs in binary XML tree.
[Lamping, POPL 1990]



S → D(D(A))

D(y) → c(A , d(A , y))

A → c(B , B)

B → a

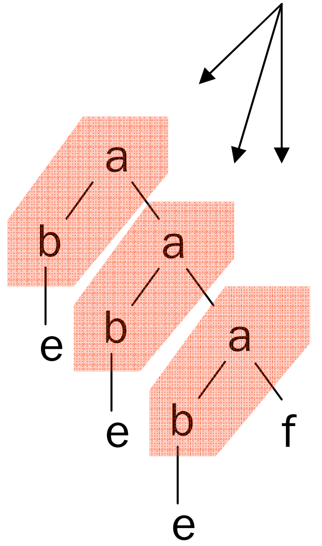
Context-free tree grammar

Sharing Graphs (SLT grammars)

Idea, share repeated (connected) subgraphs in binary XML tree.
Represent them as trees with parameters.

Example

repeated subgraph



share repeated subgraphs

using BPLEX algorithm

```
(1: a[b[y1],y2]
 2: 1[c,1[d,1[e,f]]])
```

Sharing graph
(in tree-grammar notation)

Note in general these subgraphs are NOT substrings!

Sharing Graphs (SLT grammars)

Known, for usual XML documents:

BPLEX algorithm produces *pointer-structures* (sharing graphs) with Approx. *2-3 times less pointers* than the DAG.

BPLEX

Brute force linear algorithm

Search in a fixed window for patters of size · `MaxPatSize` and with at most `MaxNumParam` many "outgoing edges".

NATIONAL ICT AUSTRALIA

Sharing Graphs (SLT grammars)

Known, for usual XML documents:

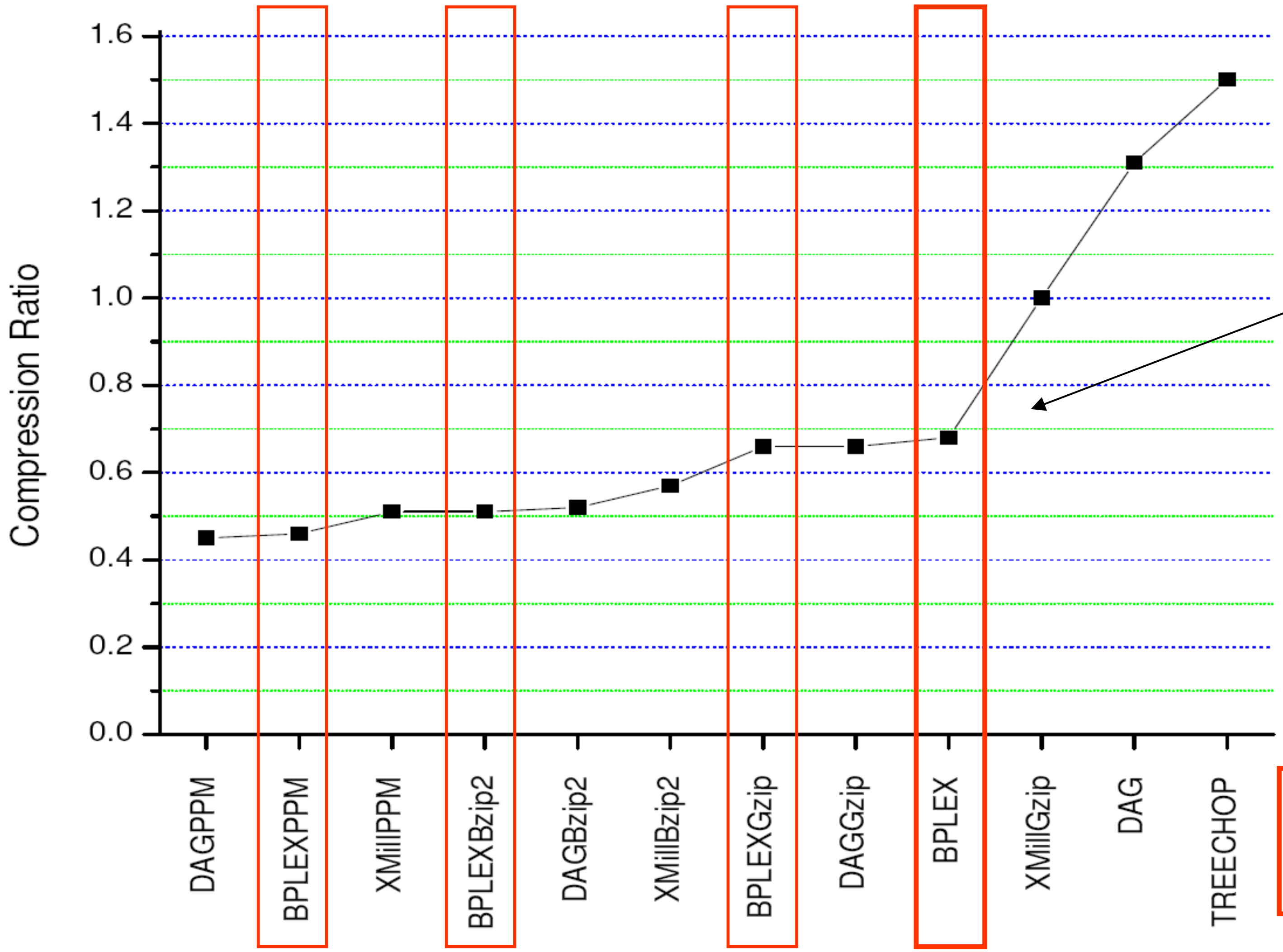
BPLEX algorithm produces *pointer-structures* (sharing graphs) with Approx. *2-3 times less pointers* than the DAG.

Consider `BPLEX,BPLEXGzip,BPLEXBzip2,BPLEXPPM`

→ again, do **not** use "minimal sharing graphs", but introduce a TRESH value, similar as for DAGs

Then, optimal performance on our datasets by using
TRESH=14 for `BPLEX`
TRESH=14 for `BPLEXGzip`
TRESH=10,000 for `BPLEXBzip2`
TRESH=30,000 for `BPLEXPPM`

SLT grammars, Results



With no backend compression, BPLEX gives *better* compression than **XMillGzip!!** (=baseline = 1.0)

On average, size is 68% of XMillGzip.

SLT grammars, Results

Document	Size (KB)	Tags	# Nodes	Depth
1998statistics.xml	717	47	54,581	7
Catalog-01.xml	6,624	51	372,459	9
Catalog-02.xml	65,875	51	3,705,071	9
Dictionary-01.xml	3,481	25	513,574	9
Dictionary-02.xml	34,311	25	5,077,549	9
EnWikiNew.xml	7,834	21	665,825	6
EnWikiQuote.xml	5,034	21	437,682	6
EnWikiSource.xml	21,849	21	1,902,189	6
EnWikiVersity.xml	9,530	21	828,229	6
EnWikTionary.xml	160,373	21	14,520,656	6
EXI-Array.xml	7,156	48	226,524	10
EXI-Factbook.xml	2,087	200	86,581	6
EXI-Invoice.xml	457	53	26,130	8
EXI-Telecomp.xml	5,402	39	177,634	7
EXI-Weblog.xml	2,216	13	178,375	4
JST_gene.xml	7,932	27	388,029	8
JST_snp.xml	24,667	43	1,169,686	9
Lineitem.xml	30,270	19	1,985,776	4
Medline.xml	80,248	79	5,394,921	8
Mondial.xml	409	23	22,423	5
Nasa.xml	9,958	62	792,467	9
NCBI_gene.xml	13,042	51	645,917	8
NCBI_snp.xml	135,853	16	6,879,757	5
Sprot.xml	206,993	49	21,634,330	7
Treebank.xml	31,450	252	3,843,775	38
USHouse.xml	144	44	11,889	17

Becomes 1094 Bytes!

Becomes 213 KB!

Becomes 3.3 KB

SLT grammars, Results

Note, the "*suitable binary encoding*" in BPLEX to obtain 68% of XMillGzip, is a Huffman-coding of a natural representation of the pattern trees.

This encoding can be used with little overhead, to execute queries (such as XPath or XQuery, or any DOM program) directly on the compressed structure.
→ On average for a tree traversal, constant slow-down (c=4)
→ Per operation slow-down at most |G| ☹
→ Can be made constant, using only linearly more space (based on clever LCA algos)
[Gasienic, Kolpakov, Ptapov, Sant DCC 2005–poster]

Gives rise to a **VERY SMALL queryable representation**, smaller than any other queryable representation known from the literature.

4. Binary Coding of BPLEX Grammars

The “*suitable binary encoding*” in BPLEX to obtain 68% of XMillGzip:

Example (now *binary tree* to avoid brackets.)

```
( 0: _
  1: a[b[y1,0],y2]
  2: 1[c,1[d,1[e,d]]]  )
```

← Not encoded (fixed for all grammars)

prologue	symbol table	grammar code word
s u x 0 3	a 0 b 0 c 0 d 0 e 0 0	000001101110101111010111011111100011

- s u x: fixed file type string
- 0: encoding type (one byte); type=0 means fixed-length and non-byte-aligned
- 3: needed for fixed-length coding: bits/symbol number

a=000, b=001, c=010, d=011, e=100, y=101, 0=110, 1=111

16 bytes +
36 bits
= 21 bytes

Binary Coding of BPLEX Grammars

The "*suitable binary encoding*" in BPLEX to obtain 68% of XMillGzip:

Example (now *binary tree* to avoid brackets.)

```
( 0: _
  1: a[b[y1,0],y2]
  2: 1[c,1[d,1[e,d]]]  )
```

← Not encoded (fixed for all grammars)

Not this one. But use Huffman here.

prologue | symbol table | grammar code word

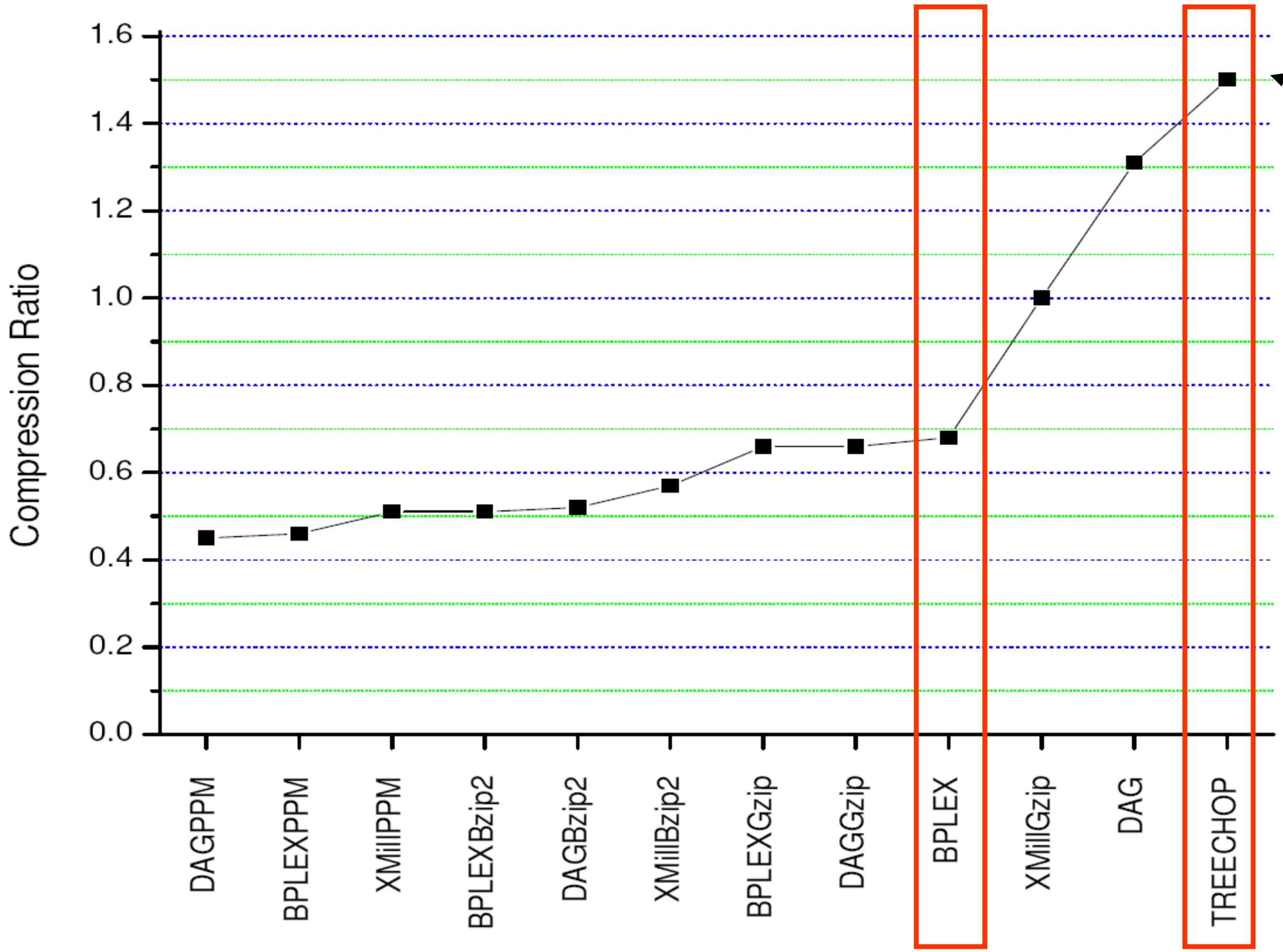
`s u x 0 3` | `a 0 b 0 c 0 d 0 e 0 0` | `000001101110101111010111011111100011`

- fixed file type string
- encoding type (one byte) type=0 means fixed-length and non-byte-aligned
- needed for fixed-length coding: bits/symbol number

a=000,b=001,c=010,d=011,
e=100,y=101,0=110,1=111

16 bytes +
36 bits
= 21 bytes

4. SLT grammars, Results



Next best, queryable format.

Approx. 2x bigger than BPLEX

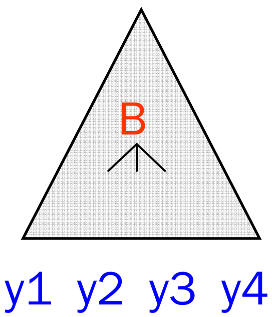
Other queryable formats require **much** more space still!!

5. Algos on SLT Grammars

Context-Free Tree Grammars (generalize cf grammars to trees)
[Rounds70, PhD Fischer68 “macro grammars”]

New: Nonterminals have parameters y1, y2, ..,

A(y1, y2, y3, y4) →



B(y1, y2, y3,) → g(C(y1, y2),y3)

C(y1, y2) → ...

→ [Lohrey, Maneth CIAA 2005]
Finite tree automaton / CoreXPath on Straight-Line tree grammar in time
$O(n^{k+1} |G| |A|)$
k = max number of parameters of NTs

→ Equality check in poly time
(use DFLR grammars and Plandowski's result on cf string grammars)

→ Incremental Updates
[Fisher, Maneth ICDE 2007]

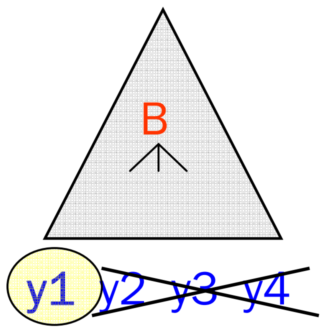
→ matching, unification, etc
[Godoy, Schmidt-Schauss LICS 2008, etc]

5. Algos on SLT Grammars

Context-Free Tree Grammars (generalize cf grammars to trees)
[Rounds70, PhD Fischer68 "macro grammars"]

New: Nonterminals have parameters y1, y2, ..,

A(y1, ~~y2, y3, y4~~) →



Tree with
one "hole"

B(y1, ~~y2, y3,~~) → g(C(y1, ~~y2),y3~~)

C(y1, y2) → ...

New Result (Dagstuhl'08)
[Lohrey, Maneth, Schmidt-Schauss 2009]
Any grammar can be made **1-param**,
with only linear blow up!!

"*singleton tree grammars*"
[Schmidt-Schauss TR 2005]

are using

→ matching, unification, etc
[Godoy, Schmidt-Schauss LICS 2008, etc]

Conclusions

For file compression of XML tree structures, DAGs are suitable.

→ they can be obtained quickly, using hashing
→ using Gzip-backend, they are only 70% of the size of XMillGzip

---

For in-memory compression, e.g., as a queryable data structure,
BPLEX-outputs are extremely well suitable

→ they can be queried with little overhead, for Core XPath queries
even with *speedup* wrt running over uncompressed tree [Lohrey,Maneth2007]

→ using **no** backend, they are only 68% of the size of XMillGzip

→ problematic: BPLEX runs quite slow! A new, fast tree grammar compressor
based on RePair (Moffat et al) is on its way!

Conclusions

Questions

→ How can we obtain *better codings* for DAGs/BLEX grammars?
→ Are there well-known tricks to amortize the cost of a "reference"?
→ Anything known about succinct DAGs?

We tried Kieffer/Yang's grammar transforms. Results were NOT good. ☹

→ Can we use string (grammar) compressors to obtain faster
Approx. algos that produce small tree grammars?

→ Grammars with copying of parameters can give double-exp compression ratios.
Useful for tree compression?

```
(1: a[y1,y1]
 2: 1[1[y1]]
 3: 2[2[y1]]
 4: 3[3[e]])
```

decompress ⇒

Full binary tree of
height 2^3
(size 2^{2^3})

Thank you..

.. for your attention!

For questions, please email

sebastian.maneth@nicta.com.au

-------- THE END ---------