Compression of Structured Big Data

Challenges and Solutions

Stefan Böttcher
University of Paderborn

Talk outline

1. My personal research path to
   Compression of Structured Big Data
   (searching for new open research problems, …)

2. Some Big Data Challenges
   (focus on problems that motivate compression)

3. (main part)
   An overview of selected compression techniques
   - principles applied in well known compressors
   - grammar-based compression and its application to
     text, trees, and graphs
   - recompression

My Research Path to Compression of Structured Big Data (1)

My starting point:
+ 5+ years research in relational database systems
+ search for new open problems

Shift from relational databases to XML databases
- transaction synchronisation, ...
  → nearly orthogonal to data model → easy to transfer

+ non-orthogonal concepts, here: relying on data structure,
  e.g. queries, access control, views, ...
  → new solutions required → (potentially new) research topics

  new results on
    XML access control, XML query optimization, …

My Research Path to Compression of Structured Big Data (2)

Shift from XML databases to compressed XML

access control, ...
→ nearly orthogonal to compression → easy to transfer

non-orthogonal concepts, here: relying on data access,
e.g. queries, caching, XML schema...
→ new solutions required → (potentially new) research topics

new results on

- XML caching
- XML encoders
- schema-based XML compression
- grammar-based XML compression
- parallel multi-query optimization
- ...

My Research Path to Compression of Structured Big Data (3)

Shift from compressed XML to compressed strings and graphs

- grammar-based compression nearly orthogonal to data type
  → easy to transfer

- non-orthogonal concepts, here: relying on data structure,
  e.g. queries, modification, ...
  → new solutions required → (potentially new) research topics

- new results on
  - IRT – an updatable BWT
  - parallel compression of strings
  - compression of commutative trees
  - compressing graphs

Big Data Examples

Financial transactions

Genom data

Weather forecast

Sensor data

Social networks

Big text data ←

Big Data Processing Examples

Pattern detection, e.g.
  crime detection in financial transactions,
  behaviour derivation from genom data, …

Prediction, e.g.
  predictive maintainance,
  market development, …

Data aggregation and data transformation, e.g.
  wheather forcast,
  big data transmission into clouds

Archiving big data, e.g.
  string data (documents, genom data, …),
  graph data (social networks, …), …

Some Big Data Processing Challenges

Too much data for efficient

- storage
- transmission
- information extraction
- search of patterns
- transformation
- data cleaning

➔ most algorithms will take too long

Some Big Data Processing Challenges

Too much data for efficient	Nevertheless, we want to
storage	- store
transmission	- transmit
information extraction	- extract information
search of patterns	- search patterns
transformation	- transform
data cleaning	- modify big data
➔ most algorithms will take too long	➔ any way out?

Some Big Data Processing Challenges …

“Companies are creating so much data,
it has to be shipped in trucks“

e.g. DigtalGlobe‘s data transfer into the cloud
→ a truck, full of Amazon‘s snowball devices
needs 10 days to ship the data into Amazon‘s cloud

in comparison to uploading the data
which currently needs 300 years

source:
http://www.xing-news.com/reader/news/articles/736131

Are there alternatives? → (next slides)

What is compression?

Substitute a larger data set by an “equivalent“ shorter data set

Lossless compression:

larger data set can be reconstructed from shorter data set
(e.g. gzip, bzip2, …)

Lossy compression:

larger data set cannot be reconstructed from shorter data set,
but shorter data set is sufficiently detailed for the given task
(e.g. mp3, …)

Lossless compression principle

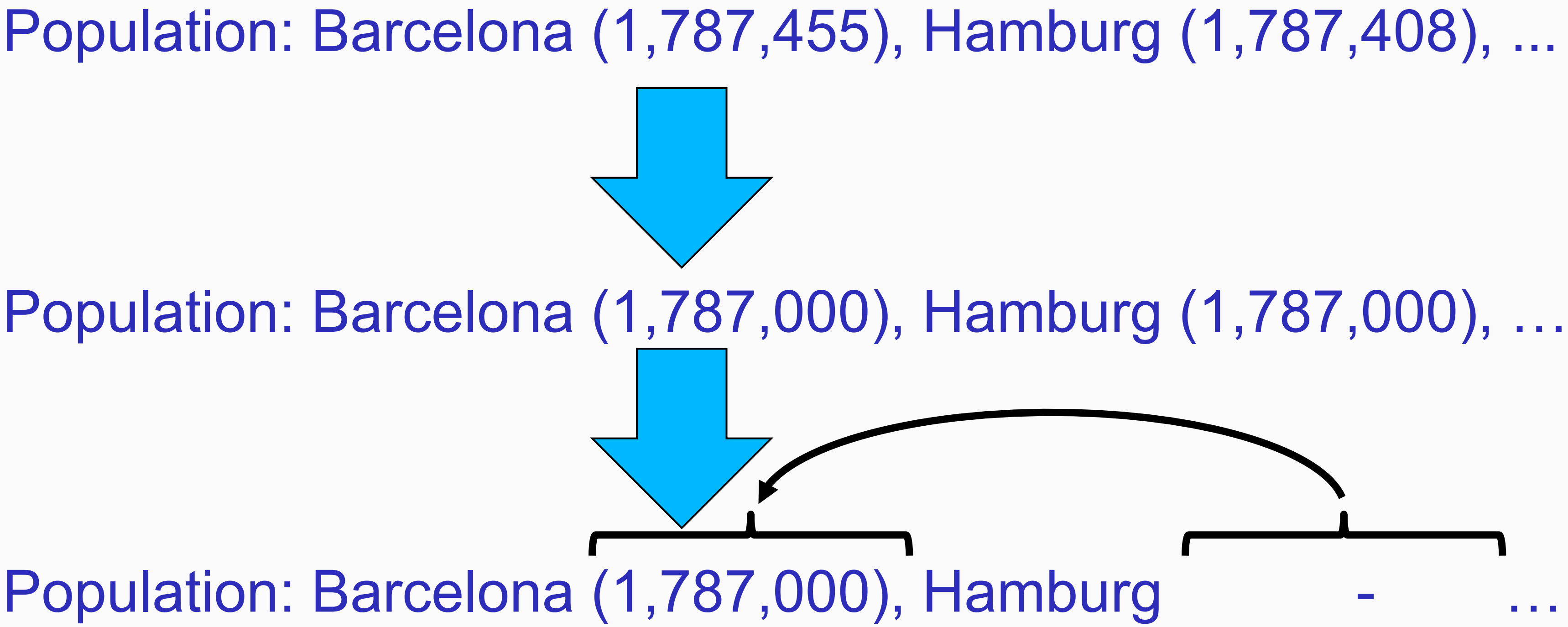
find repeated patterns in input data set
and replace repeated patterns by pointers, shortcuts, or …

Barcelona is the capital city of Catalonia … . Barcelona is a …

Barcelona is the capital city of Catalonia … . - - a …

Lossy compression principle

find similar data or similar patterns of data in input data set and replace similar data by a unique data presentation (e.g. pointers, shortcus, …)



Population: Barcelona (1,787,455), Hamburg (1,787,408), ...

Population: Barcelona (1,787,000), Hamburg (1,787,000), …

Population: Barcelona (1,787,000), Hamburg - …

Why compression?

Goal: handle (search, …) big data efficiently

- faster memory access
  - Trend towards main memory databases
    - ➔ smaller memory footprint
  - If relevant data fits into main memory
    - ➔ faster computation possible

- faster data transmission
  - ➔ shorter data transmission time

- faster algorithms (e.g. pattern search)

➔ search repeated patterns only once

Why/When can Algorithms on Compressed Strings be Faster?

Significant speed-up expected if …

speed of algorithms depends on size of input, e.g.,
search in big text data collections
depends on the number of characters to be read/processed

processing repeated text
(find, read, evaluate, transform, …)
can be combined into a single step for all the data
(find once, read once, evaluate once, transform once, …)

Very likely to be relevant for big text data

Example: Word Count

Speed of (sub-)string count in text
depends on size of input,
i.e., number of characters

faster on
compressed text

How often is “Barcelona“ mentioned?
➔ 1. find “Barcelona“, 2. count incoming pointers to “Barcelona“

How often is “Bar“ mentioned?
➔ 1. find all words containing “Bar“, 2. count incoming pointers

Generalizing the Word Count Example to Sub-String Search

Count number of strings “Barcelona“

↓

Find all locations of strings “Barcelona“

↓

Navigate from all locations of strings “Barcelona“ to next word

↓

Search for sequences of compressed words, e.g. “Barcelona is“

↓

Search for sub-strings crossing pattern boundaries, e.g. “elona i“

↓

…

still faster on compressed text?

Generalizing to Arbitrary Algorithms on Compressed Text?

Instead of individual solutions for specialized algorithms …

What is common to all algorithms on massive compressed text?

Although text is compressed:
- access to certain text fragments (nobody can read all the text…)

- access by content or by position, e.g. relative to other content

- read access and write access to text fragments

- support of operations on massive text (multi-read, multi-write,…)

Offer efficient basic operations on massive compressed text

Basic Operations for Algorithms on Compressed Text

Algorithms on massive compressed text data
rely on basic operations on sub-strings like

still faster on compressed text?

locate all positions of a given sub-string

navigate from one huge set of positions to another

read sub-strings at a huge number of given positions

transform sub-strings at a huge number of given positions
(including copy, insert, delete, update, … of sub-strings
at a huge number of given positions)

even these elementary operations are a challenge on massive compressed text data

Text Compression Tools

gzip, bzip2, … combine some of the following

compression techniques
- replace longer sequences of symbols (characters/words/…) by shorter sequences
- fixed replacement rules (Run Length Encoding,…)
- explicit dynamic dictionaries (Repair, Sequitur)
- implicit dynamic dictionaries (LZ77, LZ78, …)

encoding techniques
- use shorter codes for more frequent symbols (Huffman,…)

additional (pre-)transformations to improve compression
- e.g. based on rotations (MoveToFront, BWT, IRT)

Pre-Transformation by Burrows Wheeler Transform (BWT) or by Indexed Reversable Transformation (IRT)

Input = a b r a c a d a b r a

BWT = r d a r c a a a a b b

computed rotations:

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

sorted rotations:

BWT

a a b r a c a d a b r
a b r a a b r a c a d
a b r a c a d a b r a ← E (end)
a c a d a b r a a b r
a d a b r a a b r a c
b r a a b r a c a d a
b r a c a d a b r a a
c a d a b r a a b r a
d a b r a a b r a c a
r a a b r a c a d a b
r a c a d a b r a a b

+ BWT allows to reconstruct input
+ substrings can be searched by fast LF mapping
+ BWT has more character repetitions than input → easier to compress
++ IRT additionally allows to directly access the Nth word

Compression by Run Length Encoding

BWT	=	r	d	a	r	c	a	a	a	a	b	b		
run length encoding	=	1	1	1	1	1	1	0	0	0	1	0	←	uses bits only
shorter BWT	=	r	d	a	r	c	a				b		←	to save bytes

0-bit = repitition of previous character

1-bit = new character

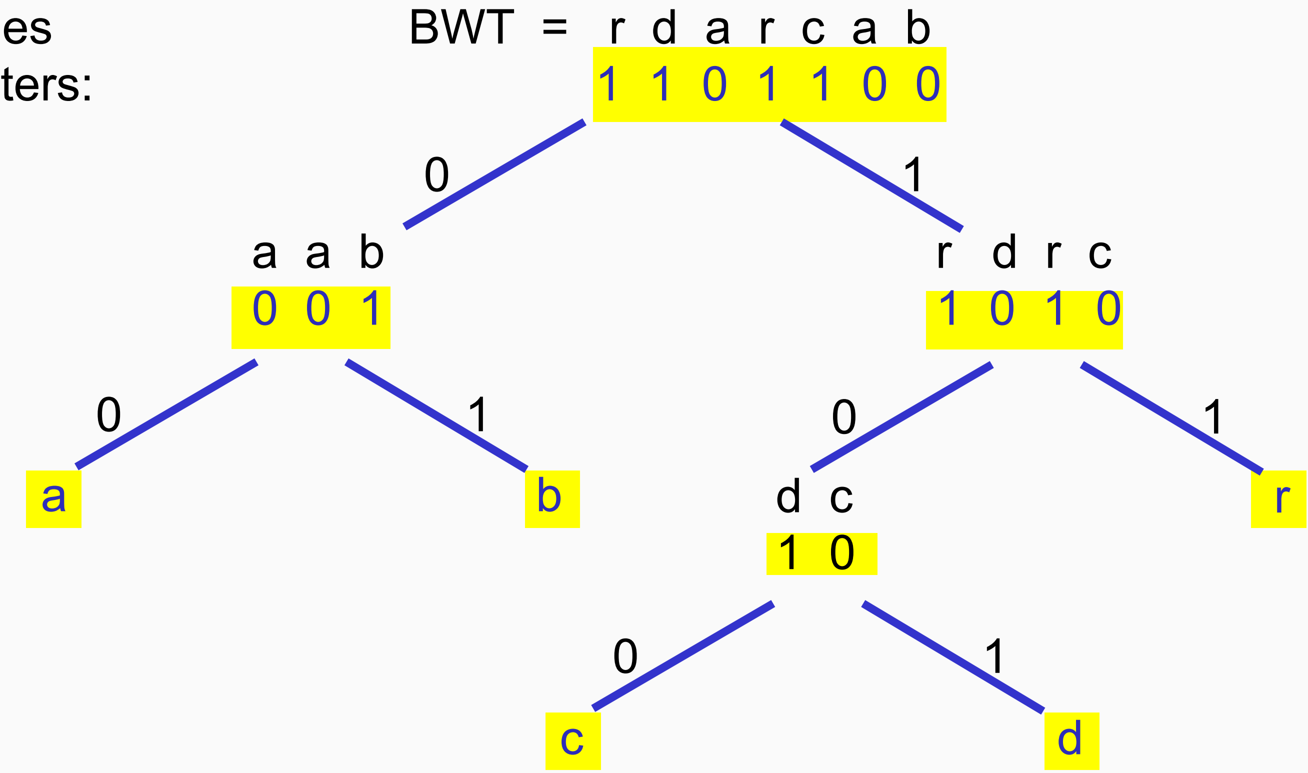
Variants:

text		r	d	a	r	c	a	a	a	a	b	b
alternative encoding	=	1	1	1	1	1	4	0	0	0	2	
shorter text	=	r	d	a	r	c	a				b	

run length encoding	=		1	1	1	1	1	1	0	0	0	1	0
gamma coding	=	0	6						3			1	1

Huffman-Coded Wavelet-Tree

shorter Huffman codes
for more frequent letters:
2 * a → 00
1 * b → 01
1 * c → 100
1 * d → 101
2 * r → 11



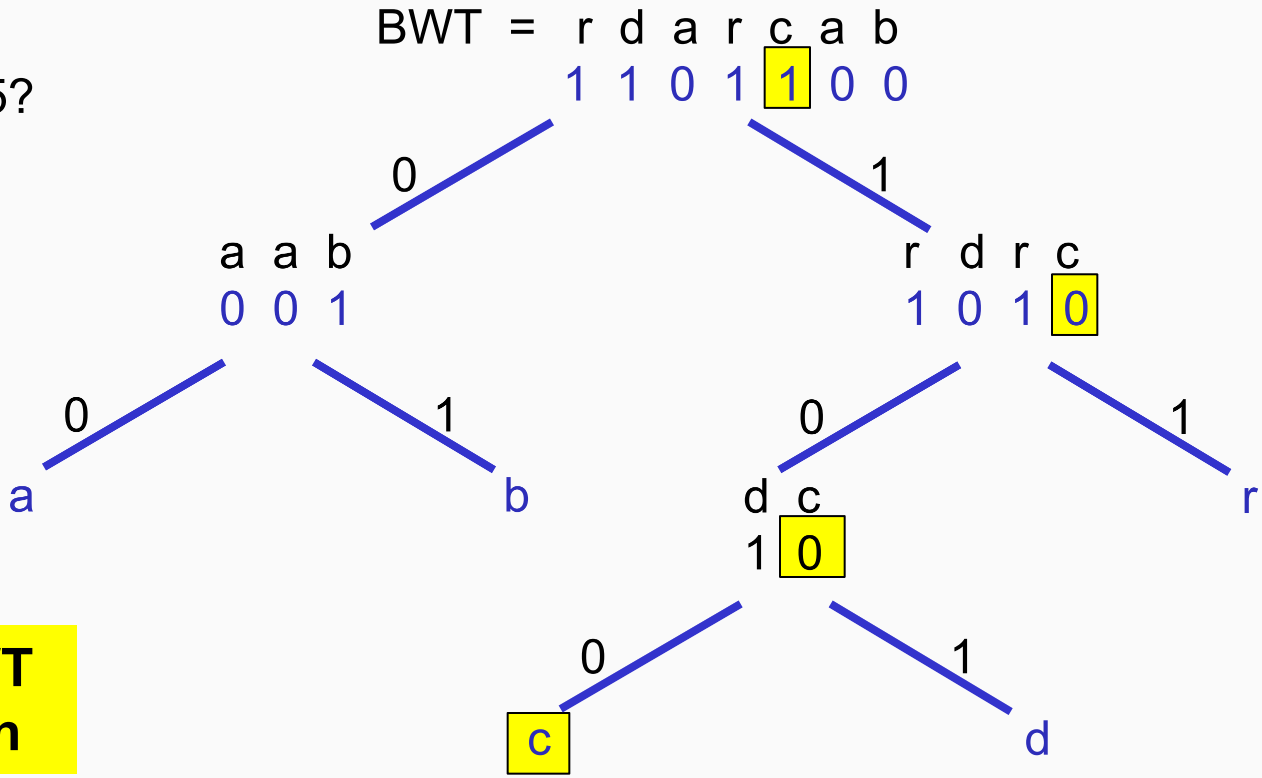
Only the highlighted part is the Wavelet Tree (WT) to be stored

When using an alphabetic Huffman code, the Wavelet Tree is sorted,
i.e. supports searching all symbols (letters, words, …) <= a given constant

Direct access on Wavelet-Tree (WT)

No direct access on
Huffman codes:
e.g. letter at position 5?

11 101 00 11 100 …
r d a r c



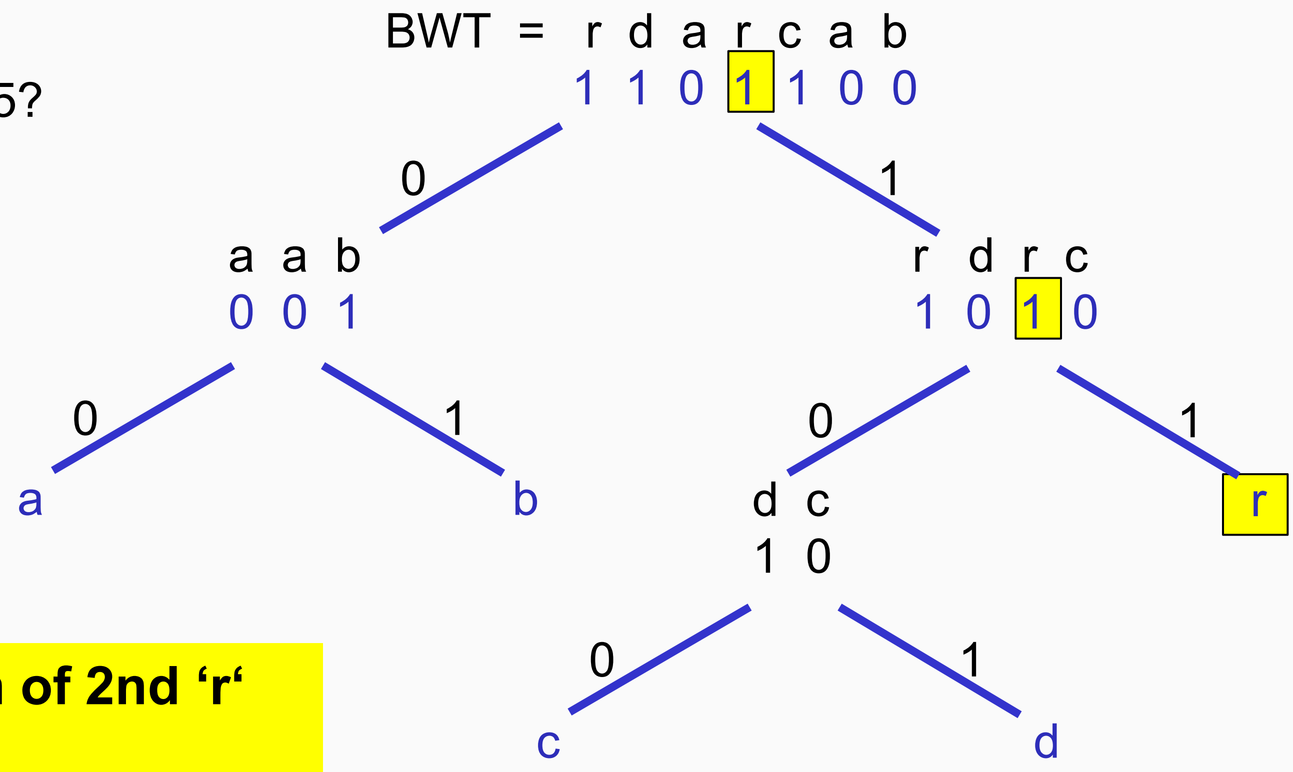
letter at pos 5 in WT
→ top-down search

In comparison to Huffman coding,
on the Wavelet Tree, only the letter at position 5 has to be decoded,
but no previous symbols (letters, words, …) need not be decoded

Search in Wavelet-Tree (WT)

No direct access on
Huffman codes:
e.g. letter at position 5?

11 101 00 11 100 …
r d a r c



search for position of 2nd ‘r‘ in input BWT
→ bottom-up search in the WT

In comparison to Huffman coding,
on the Wavelet Tree, only the 2nd letter has to be decoded,
but no previous symbols (letters, words, …) need not be decoded

Partial Solution: String Encodings

Although encodings (Huffman, Wavelet Tree, …) substitute longer words/symbols by shorter words/symbols, the number of symbols remains the same

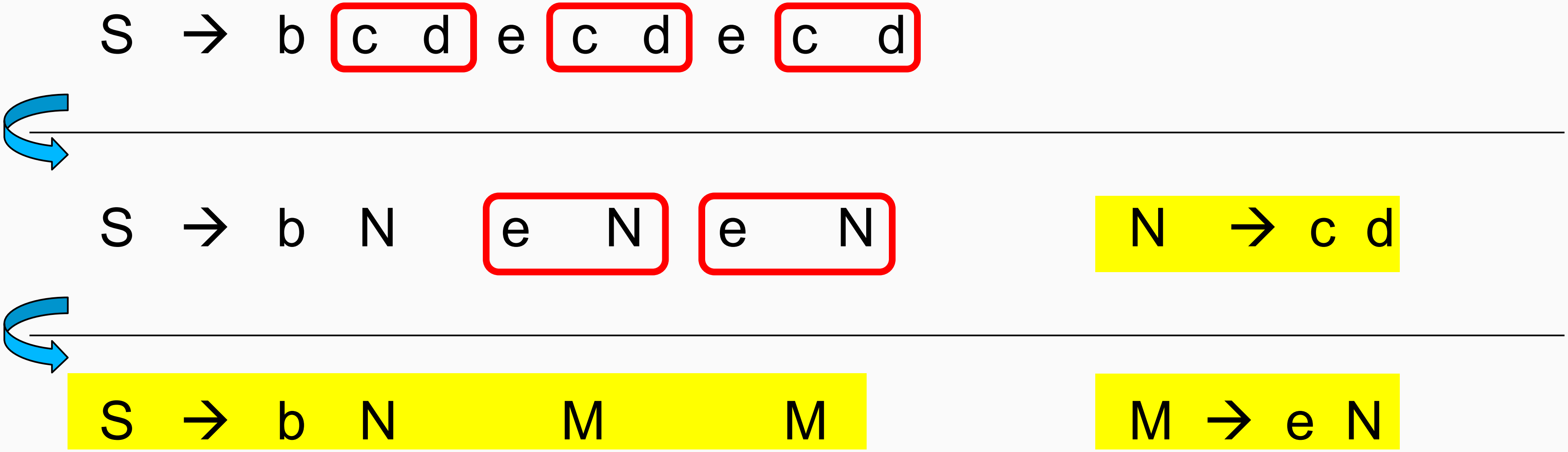
Also pre-transformations do not reduce the number of symbols

→ both steps alone leave still too many symbols for big text data

→ we techniques to reduce the number of symbols/shortcuts

… and there are alternatives to Run Length Encoding

Alternative: Grammar-based string compression



S → b c d e c d e c d

S → b N e N e N   N → c d

S → b N M M   M → e N

replacing (most frequent) digram occurrences uses a “look for smallest repeated pattern first“ – approach

substitute larger frequently occurring patterns in multiple steps

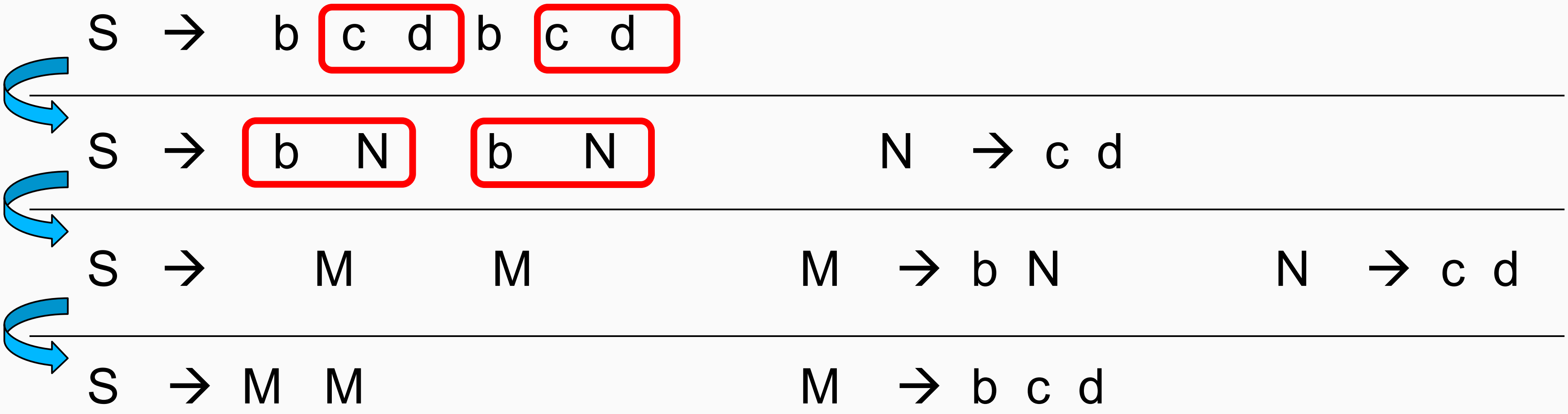
Algorithms on Grammar-Compressed Strings

In the optimal case,
string-grammars are exponentially smaller than a text,
i.e., a text with N characters/words/symbols/…
can be represented by a string grammar of size log(N)

Basic operations are supported:

- read
    - find position(s) of given content
    - determine content at position(s)
    - navigate to surrounding position(s)
- modify / transform
    - insert text at given position(s)
    - update text at given position(s)
    - copy text at given position(s)
    - delete text at given position(s)

(Re-)Compression by replacing a most frequent digram



S → b c d b c d

S → b N b N   N → c d

S → M M   M → b N   N → c d

S → M M   M → b c d

(Re-)Compression Algorithm for strings / trees / graphs :

```
while at least one digram occurs more than once
    choose a most frequent digram D  ( e.g.  c  d  )
    (if re-compression: → isolate all occurrences of D by smart inlining)
    replace each occurrence of digram D by a new nonterminal N,
            which is thereafter treated as a terminal, i.e. not cut-off again
    introduce a grammar rule      ( e.g.  N → c  d  )
    inline rules called only once  ( e.g.  N → c  d  )
```

When can Algorithms Become Faster on Compressed Input?

Speed of many algorithms depends on size of input, e.g. for

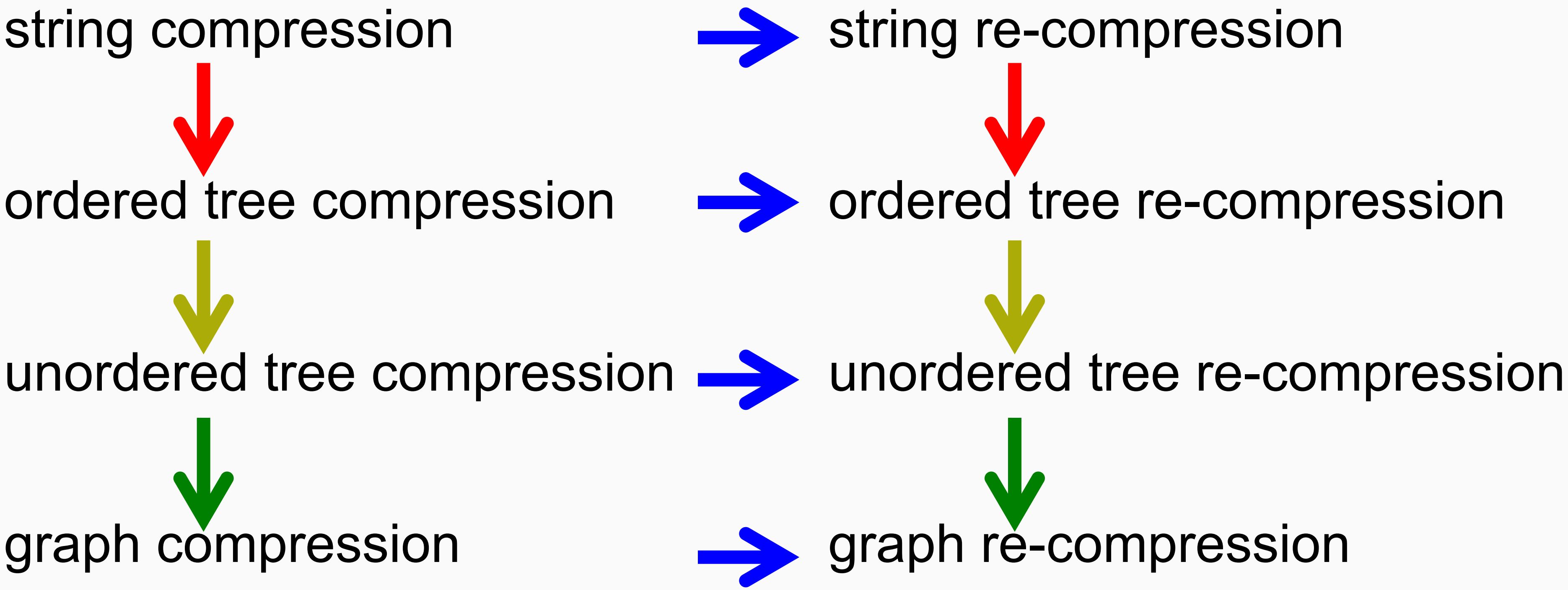
Strings – number of characters

Trees , graphs – number of nodes and edges

Goals:
minimize number of characters / edges … by
storing repeated patterns only once (=compression)

transform algorithms,
such that they need a smaller amont of data accesses

Overwiew of steps towards re-compressed graphs



string compression → string re-compression

ordered tree compression → ordered tree re-compression

unordered tree compression → unordered tree re-compression

graph compression → graph re-compression

---

compression → re-compression
strings → ordered trees
ordered trees → unordered trees
unordered trees → graphs

From Algorithms on Strings to Algorithms on Trees

Algorithms on massive tree-structured data
rely on elementary operations on nodes and edges of trees, e.g.

- locate positions of all nodes (or edges) having a given label
- navigate from one huge set of nodes (or edges) to another
- read labels of a huge given set of nodes (or edges)
- transform sub-trees at a huge number of given positions (including copy, insert, delete, update, … of sub-trees at a huge number of given positions)

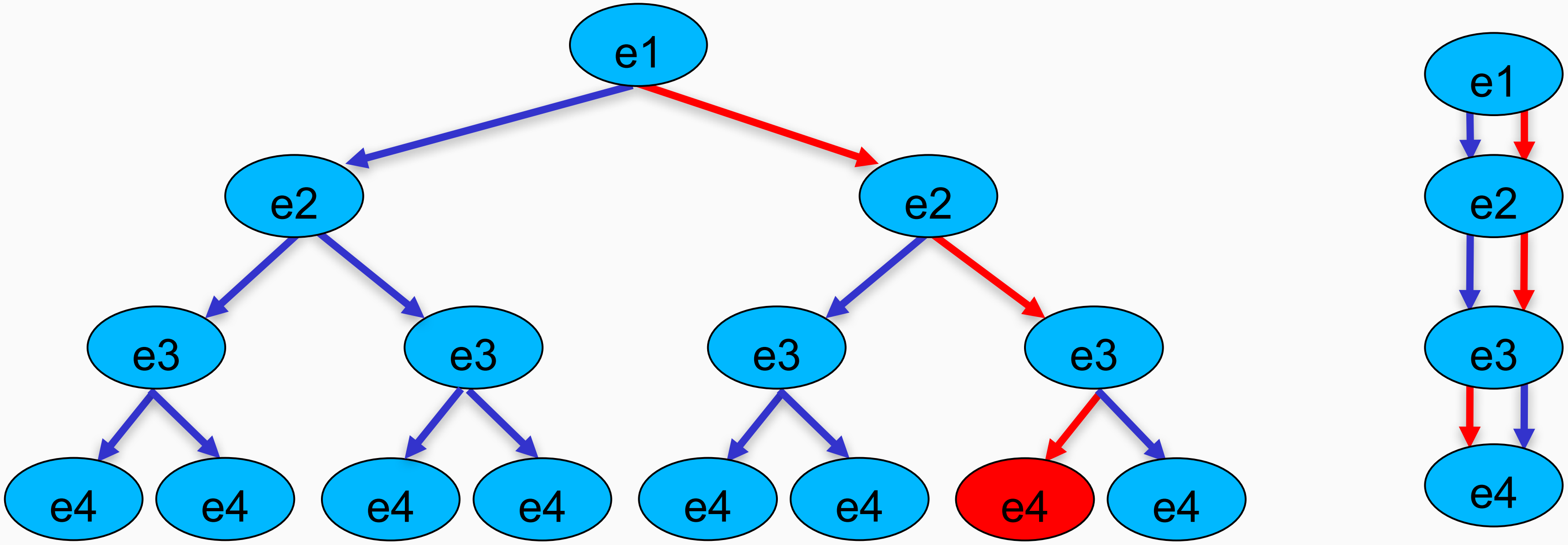
still faster on compressed trees?

Can we extend efficient algorithms on compressed strings
to efficient algorithms on compressed trees?

Directed Acyclic Graph (DAG) - Compressed Trees

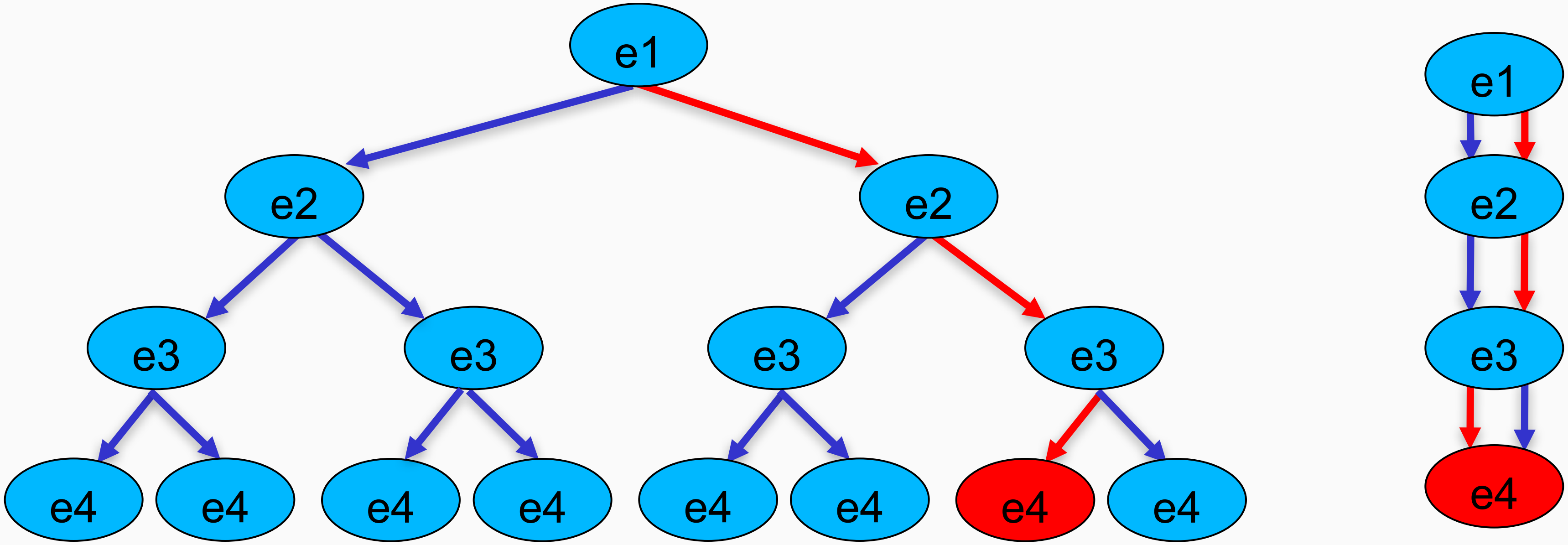
Each node N in a tree, can be represented by
- a path from the root node of the tree to that node N
- a path in the DAG from the DAGs root to a DAG-node corresponding to that node N

A DAG node can correspond to multiple nodes of a tree



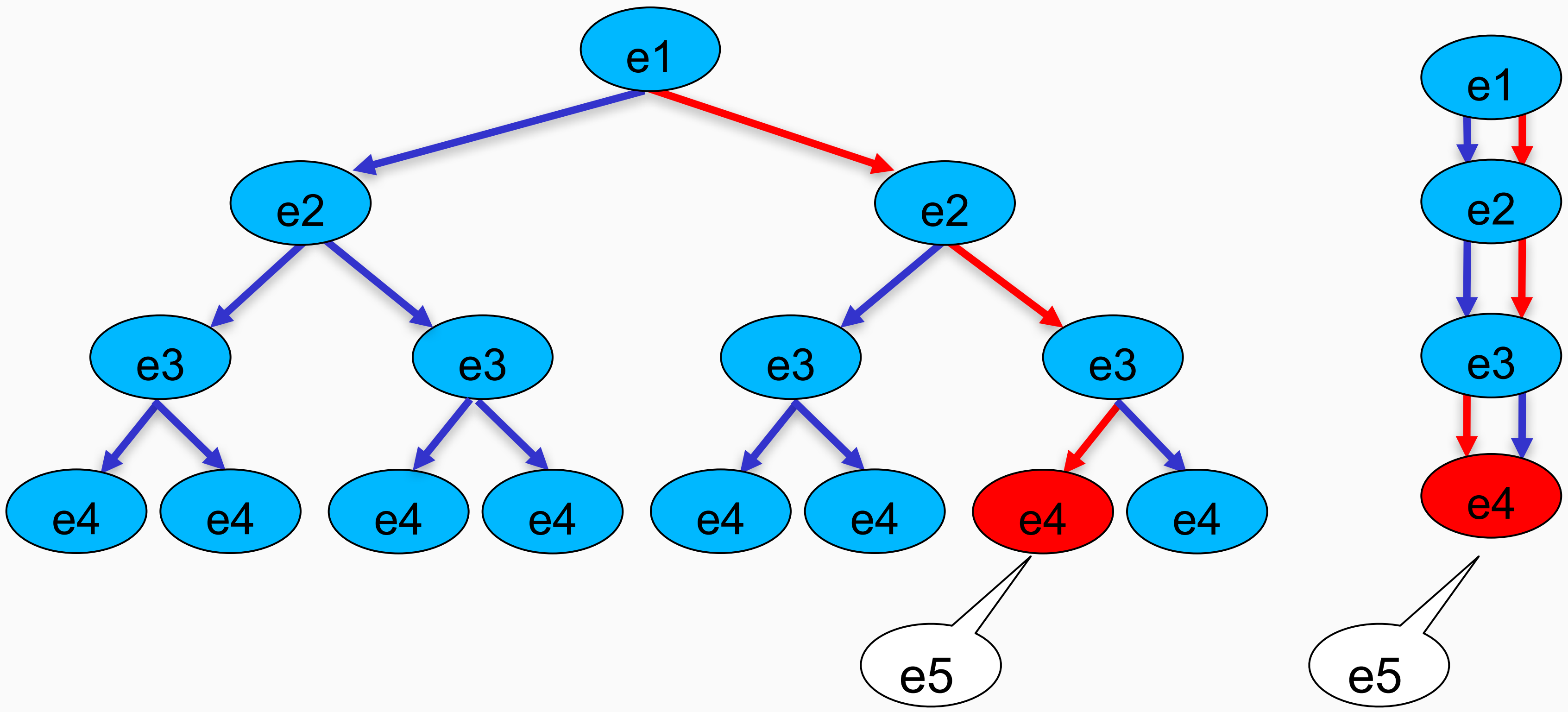
Faster Algorithms on DAG-Compressed Trees

In the optimal case,
Directed Acyclic Graphs (DAGs) are exponentially smaller, i.e.,
a tree with N nodes and N-1 edges
can be represented by a DAG of size log(N)



Runtime of algorithms visiting each node once (e.g. label count),
may be reduced from N to log(N) effort in the optimal case

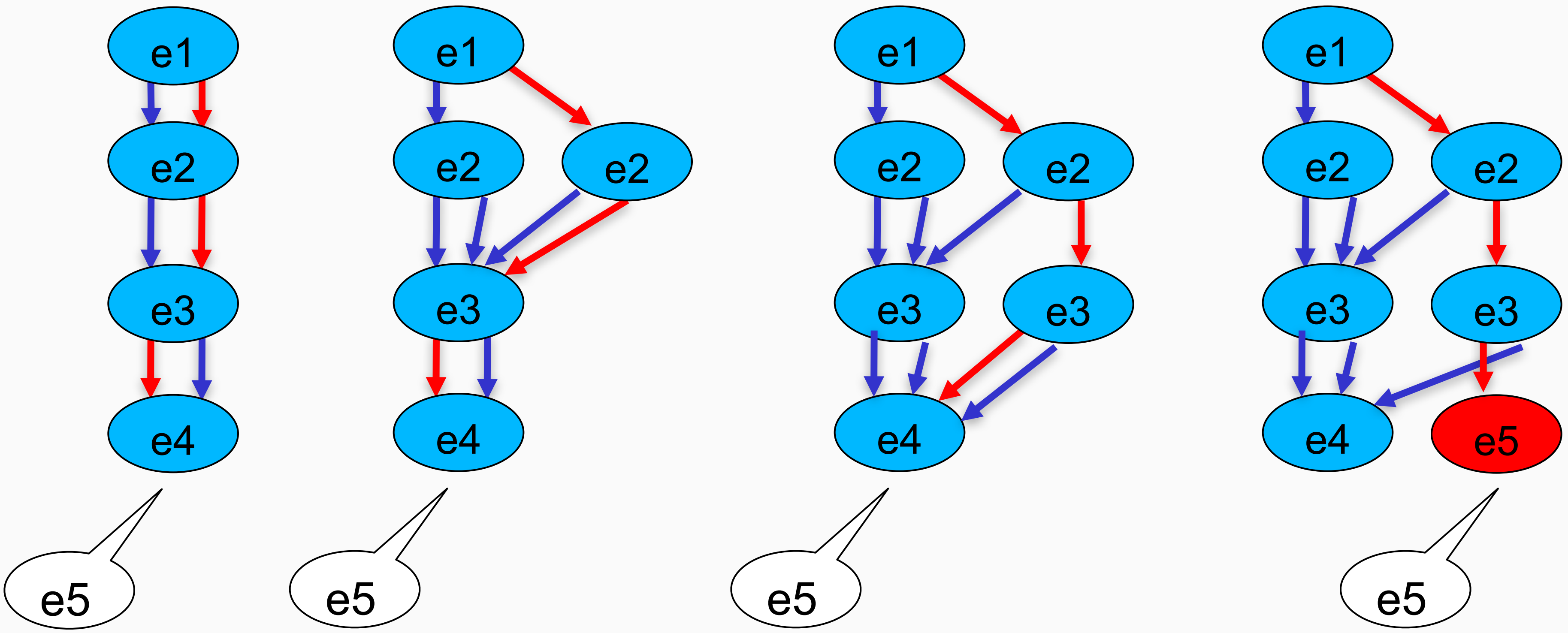
Updates on DAG-Compressed Trees?



Update, after isolation of the path in the DAG corresponding to the tree node to be updated

Updates on DAG-Compressed Trees after path isolation

E.g. isolate the red path of the DAG:
Top down on the red path, copy all the nodes having incoming edges of different colors together with their outgoing edges

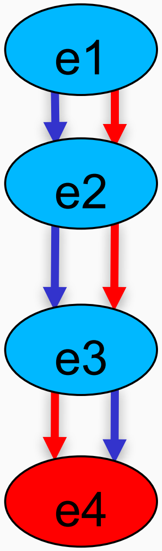


After path isolation, update in the DAG possible

Compression by Tree Grammars (without parameters)

The tree can be represented by a Tree Grammar, i.e. a grammar where the right-hand-side of grammar rules represent (repeated) sub-trees.

Example:



S → e1 (E2 , E2)

E2 → e2 (E3 , E3)

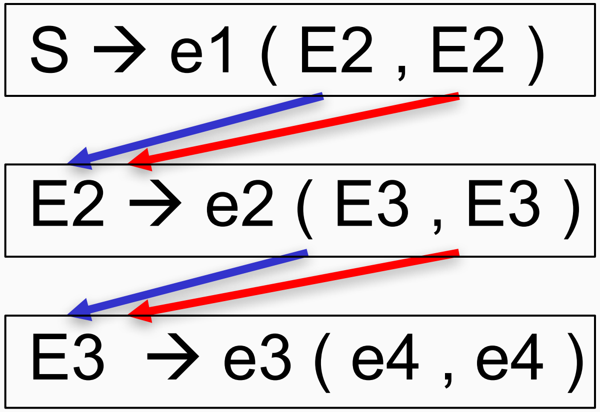
E3 → e3 (e4 , e4)

The last grammar rule states:
the nonterminal E3 is a shortcut
for an e3 node having a first child e4 and a second child e4

S, the nonterminal of the grammar‘s start rule, is a shortcut
for the whole compressed tree

Algorithms on Tree Grammars (without parameters)

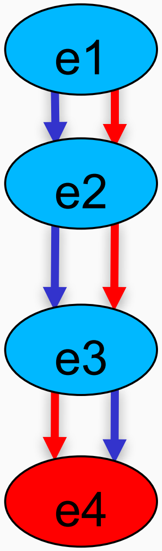
Example (continued):



S → e1 (E2 , E2)

E2 → e2 (E3 , E3)

E3 → e3 (e4 , e4)

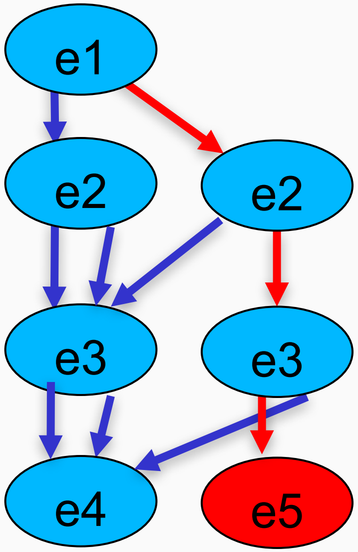


In order to simulate operations on the compressed tree, algorithms on Tree Grammars read the grammar rules (nodes) and follow the edges calling other rules

Less than one grammar rule (without parameters) per DAG node

Compression by Tree Grammars with parameters

Example:



S → e1 (E2(e4) , E2(e5))

E2(y1) → e2 (E3(e4) , E3(y1))

E3(y1) → e3 (y1 , e4)

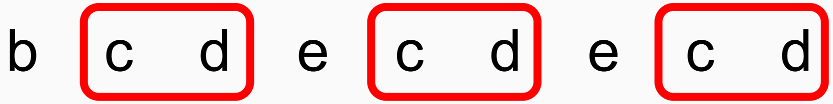
Each tree grammar rule with parameters (e.g. E2(y1) → …)
is a short-cut for multiple tree grammar rules without parameters

- ➔ in the optimal case, exponentially fewer tree grammar rules (with parameters) than DAG nodes
- ➔ in the optimal case, exponentially less runtime

Digrams for trees generate Tree Grammar rules

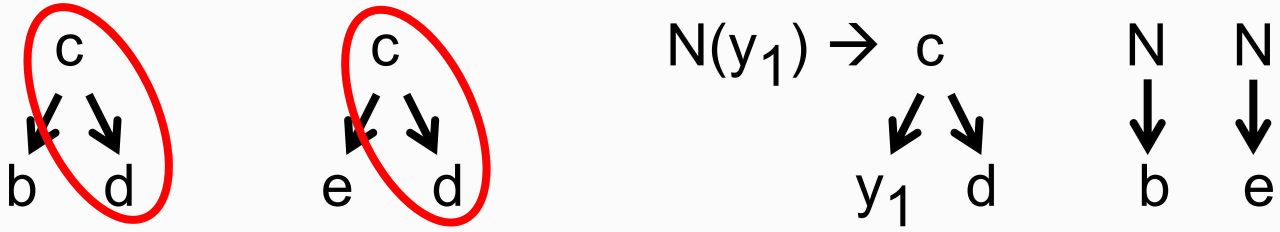
A digram is a pair of typed items (c,d) in a given relationship r

String:



digram (c,d) with r is “d follows c“

Tree:



digram (c,d) with r is “d is the second child of c“

selecting digrams consisting of inner tree nodes results in Tree Grammar rules with parameters

Algorithms on Grammar-Compressed Trees

In the optimal case, Tree Grammars are exponentially smaller than a DAG, i.e., a DAG with N nodes and edges can be represented by a Tree Grammar of size log(N) rules [several contributions by Markus Lohrey and Sebastian Maneth]

Basic operations are supported on Tree Grammars:

- read
  - find position(s) of given content
  - determine content at position(s)
  - navigate to surrounding position(s)
- modify / transform
  - insert, update, copy or delete tree at given position(s)

Goal: execute massive operations in O(size of the grammar)

→ up to exponentially faster than on DAG / Tree

From Algorithms on Grammar-Compressed Trees to Algorithms on Grammar-Compressed Graphs

Can we extend efficient algorithms on compressed trees to efficient algorithms on compressed graphs?

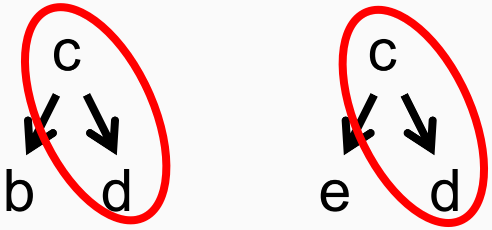
New challenges, as graph structure is more complex, i.e.,

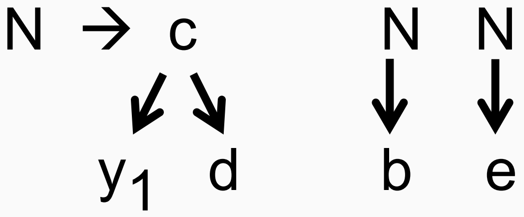
- graph may contain multiple paths from A to B
- graph may contain cycles
- graph may be partitioned
- graph may be difficult to partition into tractable sub-graphs

Digrams for ordered trees and for unordered trees

Intermediate step: unordered tree

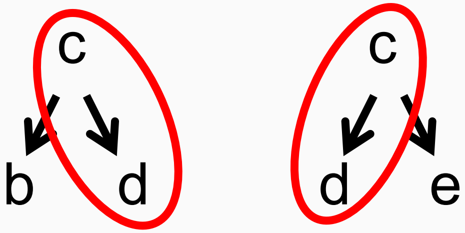
Tree:





digram (c,d) with r is “d is the second child of c“

Unordered Tree:



edge order
does not matter -
like in graphs

digram (c,d) with r is “d is a child of c“

Digrams for a graph with labeled nodes and labeled edges

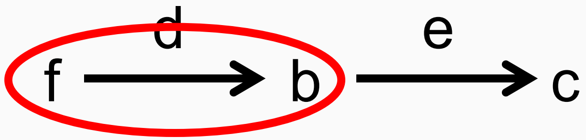
A digram is a pair of typed items (c,d) in a given relationship r

Graph: $f \xrightarrow{d} b \xrightarrow{e} c$

Digrams for a graph with labeled nodes and labeled edges

A digram is a pair of typed items (c,d) in a given relationship r

Graph:



digram (f,b) with r is “nodes f and b are connected by
a hyperedge from f to b“

digram (d,e) with r is “there is a node shared by an incoming
hyperedge d and an outgoing hyperedge e“

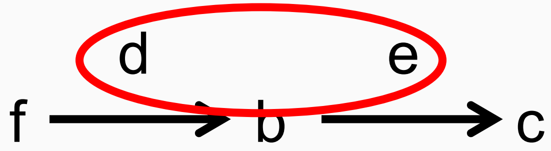
digram (b,e) with r is “node b has an outgoing hyperedge e“

digram (d,b) with r is “node b has an incoming hyperedge d“

Digrams for a graph with labeled nodes and labeled edges

A digram is a pair of typed items (c,d) in a given relationship r

Graph:



digram (f,b) with r is “nodes f and b are connected by a hyperedge from f to b“

digram (d,e) with r is “there is a node shared by an incoming hyperedge d and an outgoing hyperedge e“

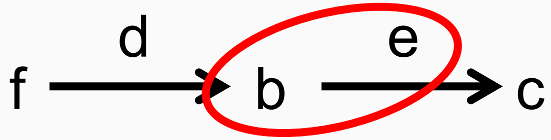
digram (b,e) with r is “node b has an outgoing hyperedge e“

digram (d,b) with r is “node b has an incoming hyperedge d“

Digrams for a graph with labeled nodes and labeled edges

A digram is a pair of typed items (c,d) in a given relationship r

Graph:



digram (f,b) with r is “nodes f and b are connected by a hyperedge from f to b“

digram (d,e) with r is “there is a node shared by an incoming hyperedge d and an outgoing hyperedge e“

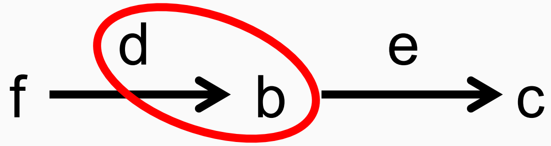
digram (b,e) with r is “node b has an outgoing hyperedge e“

digram (d,b) with r is “node b has an incoming hyperedge d“

Digrams for a graph with labeled nodes and labeled edges

A digram is a pair of typed items (c,d) in a given relationship r

Graph:



digram (f,b) with r is “nodes f and b are connected by an edge from f to b“

digram (d,e) with r is “there is a node shared by an incoming edge d and an outgoing hyperedge e“

digram (b,e) with r is “node b has an outgoing edge e“

digram (d,b) with r is “node b has an incoming edge d“

Graph Grammars with parameters

Grammar rules with parameters,

the right-hand-side of which are graphs

which represent repeated sub-graphs

the parameters represent the connections of a repeated sub-graphs with its environment

First results:

graph grammar is smaller than graph (less nodes and edges)

(some) algorithms that traverse nodes and edges are faster on (some) compressed graphs

What is re-compression of compressed data?

Transform compressed data
into a more (better) compressed format
without full decompression of the data

Why Re-Compression of a Compressed Text/Tree/Graph?

Your algorithm produces an intermediate result, i.e.

- transforms big (text/tree/graph) data into
  big (text/tree/graph) data in a different format

or

- transforms just a sub-set of big (text/tree/graph) data into
  big (text/tree/graph) data in a different format

The produced data may be still too big for shipping, …
but a new (better) compression
fitting to the selected data sub-set may be sufficient
to do next processing step (e.g. ship the data)

Use compression instead of a truck to ship the data

Why Re-Compression of a Compressed Text/Tree/Graph?

large graphs ➔ “long time“ to find a “good“ compression

idea: instead:
do any compression “fast“ and in parallel on small sub-graphs
➔ get compressed sub-graphs “fast“

re-compress compressed sub-graphs
➔ re-compression time
depends on size of compressed sub-graph

Re-compression of a compressed string / tree / graph

A string / tree / graph

$S \rightarrow d \ \boxed{c \ d} \ \boxed{c \ d} \ c$

that has been compressed to

$S \rightarrow d \ N \ N \ c$ $N \rightarrow c \ d$

can be recompressed to

$S \rightarrow M \ M \ M$ $M \rightarrow d \ c$

to get a better compression

Re-compress a compressed string: 1. Count digrams

S → d N N c N → c d

digram generator	generated digram	
d N	d c	
N	c d	(occurs twice)
N N	d c	
N c	d c	

➔ (d,c) with r = “d follows c“
is the most frequent digram in decompressed graph

2. Isolate a most frequent digram by smart inlining

Task: isolate most frequent digram (d,c) with r = “d follows c“

$S \rightarrow d \quad c \quad \underline{N \quad c} \quad \underline{N \quad c}$ $\qquad N \rightarrow \boxed{c\ e\ f\ g}\ d$

needed: partial decompression of N to isolate d from N

new rules that isolate d from the end of N: $\quad N \rightarrow N_{-d}\ d$

$N_{-d} \rightarrow c\ e\ f\ g$

$S \rightarrow d\ c\ N_{-d}\ d\ c\ N_{-d}\ d\ c$

trick: inline rewritten rule $N \rightarrow N_{-d}\ d$ instead of $N \rightarrow c\ e\ f\ g\ d$

finally, substitute digrams (d,c) with new nonterminal M:

$S \rightarrow M\ N_{-d}\ M\ N_{-d}\ M \qquad M \rightarrow d\ c$

Recompression of Grammar-Compressed Trees and of Grammar-compressed Graphs

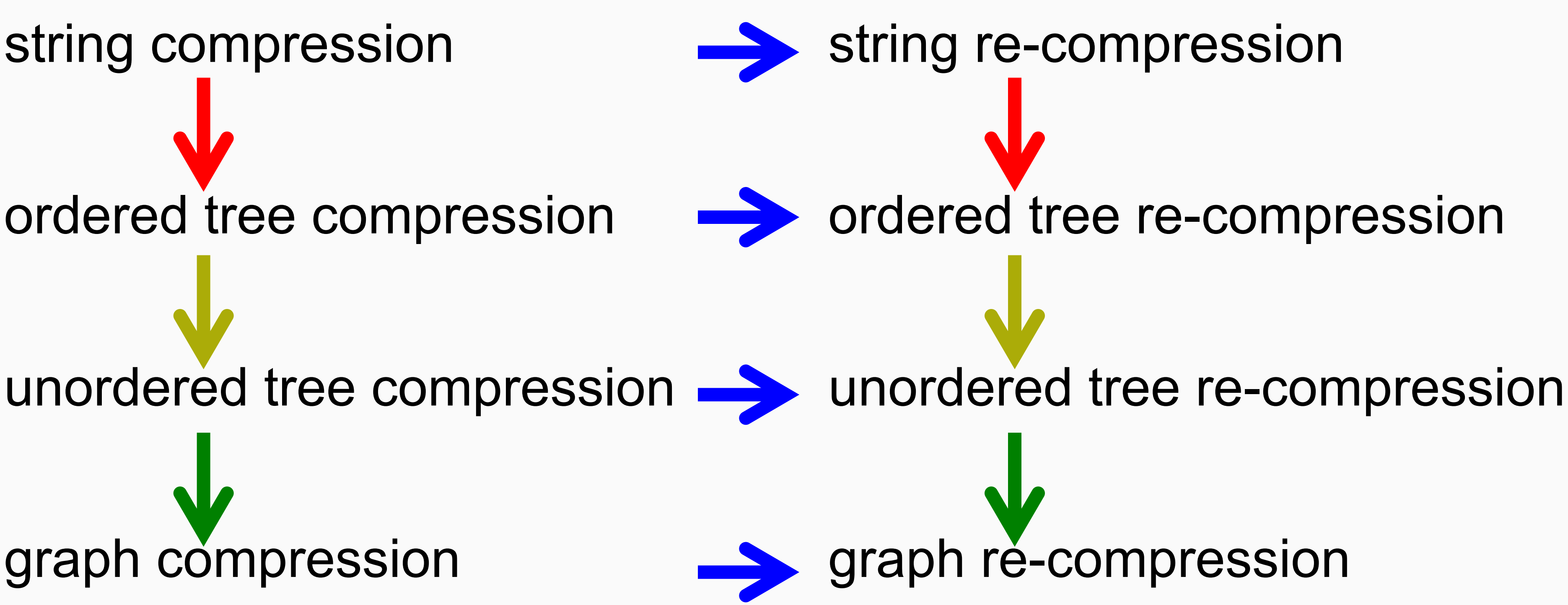
The same two basic steps:

1. Compute the most frequent digram of the decompressed String / Tree / Graph without really decompressing the String / Tree / Graph

2. Isolate all occurrence of the selected digram from the String / Tree / Graph without really decompressing the String / Tree / Graph

First results in [ICDE 2016] , [Dagstuhl 2016]

Overwiew of steps towards re-compressed graphs



string compression → string re-compression

ordered tree compression → ordered tree re-compression

unordered tree compression → unordered tree re-compression

graph compression → graph re-compression

---

re-compression → digram counting, smart decompression

ordered trees → SLT grammars

unordered trees → add commutativity

graphs → node-node, edge-edge & node-edge digrams