1. Combination of DAG and Succinct Compression including navigation

1.1. Description

DAG and Succinct Compression are compression techniques that follow two different strategies – grammar based compression on the one hand and encoding based compression on the other hand.

Within this lab exercise, a combination of DAG and Succinct Compression shall be implemented as it was introduced in the lecture in order to achieve a new compression approach that combines the advantages of both compression approaches: Instead of XML, the DAG representation is taken as input for the Succinct Compression, such that the combination of both approaches yields a bit stream representing the tree structure of the DAG compression, the inverted element list representing the labels and an additional list of DAG pointers representing the non-tree edges of the DAG. Implement only those DAG pointers that lead to a decrease in total space needed to store the compressed representation.

Implement compression, storage of the compressed data in a file, loading the compressed data from file and basic navigation based on first-child, next-sibling, parent, type and label. Integrate your implementation to the GUI and Framework as used in Exercise 4.

Students: Sebastian Micus, Fabian Weiβ

1.2. Further materials


2. A Succinct Representation of RePAIR grammars

2.1. Description

RePAIR is the grammar based compression technique introduced in the lecture that reduces the number of edges and nodes within an XML representation by computing a small parameterized tree grammar for an XML tree that avoids repetition of patterns that share identical nodes.

A succinct data structure is a compressed representation of a data structure that at the same time allows efficient operations on the compressed representation, as e.g. navigation or updates.

Within this exercise, implement a succinct storage format for RePAIR grammars as it was introduced in the lecture containing a combined symbol table with arity index for all
terminals and nonterminals, bits distinguishing variables from symbols and two different encodings (i.e. one encoding for the start rule, and the other encoding for linear rules). Input of your implementation is a RePAIR grammar as it was used in Exercises 5 and 6, i.e., a Hashtable containing a mapping from each rule address to the right hand side RHS of a rule, whereas RHS is an expression consisting of either a nonterminal or a terminal and their arguments, where an argument can be a parameter, a nonterminal, a nullpointer or an expression.

Students: Mario Mohr, Felix Steffen, Steffen Weber

2.2. Further materials


3. Binary navigation on RePAIR grammars

3.1. Description
RePAIR is the grammar based compression technique introduced in the lecture that reduces the number of edges and nodes within an XML representation by computing a small parameterized tree grammar for an XML tree that avoids repetition of patterns that share identical nodes.

RePAIR provides a compressed in-memory representation that allows navigational operations as e.g. provided by the DOM interface while saving memory storage costs by avoiding the expensive storage of redundant edges.

Extend your RePAIR implementation of Exercise 6 by basic navigation based on find-root, first-child, next-sibling, parent, type and label as introduced in the lecture. Integrate your implementation to the GUI and Framework as used in Exercise 4.

Number of students: 3

3.2. Further materials


4. XPath navigation and updates based on first-child, next-sibling and parent

4.1. Description
XPath is one of the standard languages when accessing XML documents and is widely used in other standards as e.g. XSLT, XQuery. Whenever a new XML representation – as e.g. a newly developed compression format for XML – wants to support direct XPath access to its
content, it is sufficient, that this compressed format supports basic navigation operations via the axes first-child, next-sibling and parent and via access to the type and label of a node. Those basic operations are sufficient, as each query containing the XPath forward axes child, self, descendant, descendant-or-self, following-sibling, following, and attribute can be transformed into expressions using the basic navigation operations.

Implement an XPath evaluator that reads an XPath query XP that conforms to the following EBNF grammar and that computes the result of XP by navigation via first-child, next-sibling, parent, label and type.

- `cxp ::= '/' locationpath`
- `locationpath ::= locationstep ('/' locationstep)*`
- `locationstep ::= x `::' t | x `::' t `[` pred `]`'
- `pred ::= locationpath | locationpath '=' const`

“cxp” is the start production, “x” represents an axis (attribute, self, child, descendant-or-self, descendant, following, following-sibling), “const” represents a constant, and “t” represents a “node test” (either an XML node name test or “*”, meaning “any node name”).

Combine your XPath evaluator with your solution to Exercises 2 (using the classes SuccinctNavigator) to test your implementation.

Extend your XPath evaluator and your Succinct implementation by updates, such that the following update commands can be performed on the succinct format without evitable decompression:

- **insert `<XML-Fragment>` as first-child of `<XPath-Expression>`**
  Inserts the XML fragment `XML-Fragment` as first-child to all nodes that are a result of the XPath expression `XPath-Expression`
- **insert `<XML-Fragment>` after `<XPath-Expression>`**
  Inserts the XML fragment `XML-Fragment` as next-sibling to all nodes that are a result of the XPath expression `XPath-Expression`
- **delete `<XPath-Expression>`**
  Deletes the sub-trees rooted with a node that is a result of the XPath expression `XPath-Expression`

`<XML-Fragment>` may contain a single tree or a list of trees to be inserted. N

Students: Vitali Gripp, Andreas Martens

4.2. Further materials

5. Looking forward

5.1. Description
XPath is one of the standard languages when accessing XML documents and is widely used in other standards as e.g. XSLT, XQuery. Whenever queries have to be evaluated on XML data streams – or when the memory that is available to evaluate the XML data is relatively small compared to the document – navigation can be only performed in a forward fashion.
The approach presented in [1] provides a set of rewrite rules such that any XPath query XP can be transformed in an equivalent XPath query XP’ that does not contain any backward axes, i.e., that does no longer contain the axes parent, ancestor, ancestor-or-self, preceding and preceding sibling.

Within this Lab exercise, a Java implementation of the rewrite rules provided in [1] shall be created that reads an XPath expression XP according to the EBNF grammar defined in chapter 2.1 of [1] and calculates an equivalent XPath expression XP’ without any backward axes.

Students: Michael Kionka, Sebastian Stey, Michael Timmerhaus

5.2. Further materials

6. Navigation and updates on ORDPATH-numbered XML representation

6.1. Description
ORDPATH is an XML numbering scheme that supports efficient navigation and updates. ORDPATH is used in SQL Server 2005 and SQL Server 2008 to represent the hierarchical XML documents in two-dimensional database tables.

Implement an XML representation in form of a two-dimensional table containing for each node its ORDPATH ID and its label. Implement the basic navigation via first-child, next-sibling, parent, type and label

Integrate your implementation to the GUI and Framework as used in Exercise 4.

Extend your implementation by updates, such that the following update commands can be performed on your ORDPATH-label-table:

- **insert</XML-Fragment> as first-child of</ORDPATH-ID>**
  Inserts the XML fragment </XML-Fragment> as first-child to the node with node ID </ORDPATH-ID>

- **insert</XML-Fragment> after</ORDPATH-ID>**
  Inserts the XML fragment </XML-Fragment> as next-sibling to the node with node ID </ORDPATH-ID>

- **delete</ORDPATH-ID>**
  Deletes the sub-trees rooted with the node with node ID </ORDPATH-ID>

<XML-Fragment> may contain a single tree or a list of trees to be inserted.

Students: David Maicher, Sebastian Sindelar, Jürgen Tessmann

6.2. Further materials
7. Optimizing the RePAIR implementation

7.1. Description
RePAIR is the grammar based compression technique introduced in the lecture that reduces the number of edges and nodes within an XML representation by computing a small parameterized tree grammar for an XML tree that avoids repetition of patterns that share identical nodes.

Within Exercise 6 only a simplified version of the RePAIR compression was computed, that only searches for patterns within the further DAG rules.

Extend your solution for Exercise 6 in such a way, that within each iteration step the whole RePAIR grammar is searched for matching patterns. Furthermore extend your solution in such a way that it can compress a chain of $n$ next-siblings with identical labels and identical first-children with the help of $\log(n)$ rules (c.f. slide 36 of 2.4. in [1]).

Number of students: 3

7.2. Further materials


8. Updates on DAGs

8.1. Description
DAG compression is an efficient, grammar-based compression approach that combines all equivalent XML nodes (i.e., those nodes that have the same label and equivalent first-child and next-sibling nodes) to a single DAG node.

As in general, a DAG node does not represent a single XML node, but a set of XML nodes instead, a DAG node has to be isolated before any updates can be performed, such that it represents a single XML node only. In order to update a current context node represented in a DAG of a compressed XML document, isolate the path to the current context node from the DAG and alter the leaf node of the isolated path.

Extend your implementation of Exercise 4 by updates, such that the following update commands can be performed on your DAG:

- insert $<XML-Fragment>$ as first-child of $<XPath-Expression>$
  Inserts the XML fragment $XML-Fragment$ as first-child to all nodes that are a result of the XPath expression $XPath-Expression$

- insert $<XML-Fragment>$ after $<XPath-Expression>$
  Inserts the XML fragment $XML-Fragment$ as next-sibling to all nodes that are a result of the XPath expression $XPath-Expression$

- delete $<XPath-Expression>$
  Deletes the sub-trees rooted with a node that is a result of the XPath expression $XPath-Expression$
may contain a single tree or a list of trees to be inserted.

Students: Ingo Püschl, Florian Rittmeier

If you are 3 students: additionally try to merge the path to the updated node into the DAG.

8.2. Further materials


9. Burrows-Wheeler-Transformation with Move-to-Front and Huffman

9.1. Description
Burrows-Wheeler-Transformation [2,3] is a reversible String transformation such that in general the transformed String S’ contains longer runs of same characters as the original String S. Therefore, S’ can be compressed stronger than S using e.g. Huffman encoding [4] in combination with Move-to-Front [2].

Within this Lab exercise, implement the Burrows-Wheeler-Transformation with Move-to-Front and Huffman encoding on top of it (including compression and decompression). Additionally, implement a String search on the BWT transformed String S’ (i.e. with no MoveTo Front or Huffman encoding) that finds the position of each match of a given String f within the transformed String S’. Use the interval search on the BWT as shown in the lecture.

Students: Joseph Hermann Koamdom Ngadjui, Jan Schmalor

9.2. Further materials


10.1. Description
Burrows-Wheeler-Transformation [1,2] is a reversible String transformation such that in general the transformed String S’ contains longer runs of same characters as the original String S. One disadvantage of using Move-to-Front is that the structure of the original String is lost, i.e., we cannot reconstruct only a part of the String, as each character can only be
reconstructed if all preceding characters were parsed. A compressed index that allows the efficient computations of the answers to the questions “What is the position of the i-th character ‘c’?” and “How many times does the character ‘c’ occur within the first i characters?” and therefore allows the reconstruction of parts of the original input is the wavelet-tree [3].

Within this Lab exercise, implement compression and decompression based on the Burrows-Wheeler-Transformation with a wavelet-tree index on top of the transformed String S’ and store the wavelet-tree index instead of S’. Test your implementation with different random Strings of the same size that contain 8, 16, 32, 64, 128 and 256 different characters.

Students: Martin Hett, Marcus Märtens

10.2. Further materials


11. Navigation and updates on Pre/Post-numbered XML representation with CDBS encoding

11.1. Description

Pre/Post is an XML numbering scheme that supports efficient navigation and allows determining the relations descendant/ancestor and preceding/following efficiently. One weakness of the Pre/Post numbering scheme is that inserting a node yields a recomputation of major parts of the document.

The Compact Dynamic Binary String (CDBS) encoding as presented in [2] allows to insert between any two binary Strings SL and SR with SL < SR a third binary String SM such that SL < SM < SR. For each XML node use a pair of CDBS-Strings (PreCDBS, PostCDBS) such that PreCDBS is a CDBS String that is sorted correctly among the PreOrder Strings (i.e. describes a correctly sorted PreOrderN umber) and PostCDBS is a CDBS String that is sorted correctly among the PostOrder Strings (i.e. describes a correctly sorted PreOrderN umber). Such a combination of the Pre/Post numbering with CDBS encoding yields a numbering scheme that supports efficient navigation in combination with unbounded updates.

Implement an XML representation in form of a two-dimensional table containing for each node its label and its Pre/Post-ID in CDBS encoding (represented e.g. by two java.util.BitSet objects). Implement the basic navigation via first-child, next-sibling, parent, type and label

Integrate your implementation to the GUI and Framework as used in Exercise 4.

Extend your implementation by updates, such that the following update commands can be performed on your Pre/Post-label-table:
11.2. Further materials


12. Navigation and updates on fc-ns-BitPath-numbered XML representation

12.1. Description
fc-ns-BitPath is an XML numbering scheme that assigns to each XML node the first-child-next-sibling-path stored as bits (the 0-bit represents a first-child step and the 1-bit represents a next-sibling step). fc-ns-BitPath allows to efficiently compute the ID of the first-child, the next-sibling and the parent of the current context node.

Implement an XML representation in form of a two-dimensional table containing for each node its fc-ns-BitPath (represented e.g. by a java.util.BitSet object) and its label. Implement the basic navigation via first-child, next-sibling, parent, type and label

Integrate your implementation to the GUI and Framework as used in Exercise 4.

Extend your implementation by updates, such that the following update commands can be performed on your fc-ns-BitPath-label-table:

- **insert** `<XML-Fragment>` as first-child of `<fc-ns-BitPath-ID>`
  Inserts the XML fragment `<XML-Fragment>` as first-child to the node with node ID `<fc-ns-BitPath-ID>`

- **insert** `<XML-Fragment>` after `<fc-ns-BitPath-ID>`
  Inserts the XML fragment `<XML-Fragment>` as next-sibling to the node with node ID `<fc-ns-BitPath-ID>`

- **delete** `<fc-ns-BitPath-ID>`
  Deletes the sub-trees rooted with the node with node ID `<fc-ns-BitPath-ID>`

<XML-Fragment> may contain a single tree or a list of trees to be inserted.

Students: Nils Kiene, Daniel Kopic, Jan-Patrick Pater
12.2. Further materials