University of Konstanz
Department of Computer and Information Science

Master Thesis

A Visual Analytics Approach for Comparing Tree-Structures.

in fulfillment of the requirements to achieve the degree of

Master of Science (M.Sc.)

Johannes Lichtenberger
Matriculation Number :: 01/584875
E-Mail :: ⟨firstname⟩.⟨lastname⟩@uni-konstanz.de

Field of Study :: Information Engineering
Focus :: *Informatik der Systeme*

First Assessor :: *Prof. Dr. M. Waldvogel*
Second Assessor :: *Jun.-Prof. Dr. Tobias Schreck*
Advisor :: *M.Sc. Sebastian Graf*

For my parents,
Helmut and Regina Lichtenberger

Acknowledgements

First of all I would like to thank Prof. Dr. Marcel Waldvogel for providing the opportunity to write this thesis. Next, I would like to thank Jun.-Prof. Dr. Tobias Schreck and Dr. Florian Mansmann for many helpful discussions and ideas.

Furthermore, I would like to thank Sebastian Graf for his advice and guidence throughout my master thesis. Thanks also for many helpful discussions.

Special thanks to Marcus Wenz for reviewing my master thesis manuscript and some ideas which came up during several informal presentations of the prototype.

Discussions with DiSy group members Lukas Lewandowski, Patrick Lang, Sebastian Belle and Thomas Zink also provided very helpful suggestions.

I want to thank my parents for their great support and financial help during my studies.

Abstract. Todays storage capabilities facilitate the accessibility and long term archival of increasingly large data sets usually refered to as "Big Data". Tree-structured hierarchical data is very common, for instance phylogenetic trees, filesystem data, syntax trees and often times organizational structures. Analysts often face the problem of gathering information through comparison of multiple trees. Visual analytic tools aid analysts by combining visual clues and analytical reasoning. Visual representations are ideal as they tend to stress human strength which are great at interpreting visualizations.

We therefore propose a prototype for comparing tree-structures which either evolve through time or usually share large node-sets. Our backend Treetank is a tree-storage system designed to persist several revisions of a tree-structure efficiently. Different types of similarity measures are implemented adhering to the well known tree-to-tree edit problem.

The aggregated tree-structure is input to several interactive visualizations. A novel Sunburst-layout facilitates the comparison between two revisions. It provides several interaction options such as zooming as well as drilling down into the tree by selecting a new root node. Using hierarchical edge bundles to visualize moves reduces clutter from edge crossings.

Several filtering-techniques are available to compare even very large tree-structures up to many hundred thousand or even millions of nodes. Small multiple displays of the Sunburst-layout aid the comparison between multiple trees.

A short evaluation and a study of three application scenarios as well as performance evaluations proves the applicability of our approach. It surpasses most other approaches in terms of generability and scalability due to our database driven approach which allows for a fast ID-based difference algorithm optionally using hashes for filtering changed subtrees.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Ever growing amounts of data require effective and efficient storage solutions as well as scalable, interactive methods to gain new insights through exploratory analysis or to prove assumptions. Almost all data is subject to change. Nowadays storage is cheap and adheres to Moore's law[2] of doubling about every 18 months supporting the storage of several snapshots of time varying data. Furthermore existing storage solutions minimize the impact of storing such potentially very large data-sets.

Hierarichal information in the form of tree-structures is inherent to many datasets. It is almost always mapped through primary-/foreign-key relations in relational databases. Whereas this might be sufficient in many situations it introduces an additional artificial mapping. Instead, using either a graph-DBMS for directed acyclic graphs (DAG)s or a native XML-DBMS for tree-structures facilitates a straight forward approach of storing data as well as efficient traversal methods and other domain specific advantages (for instance Dijkstra's algorithm for shortest path search in graph databases and most often extensive XQuery support in XML-DBMS).

Comparison of tree-structures In order to be human readable every tree-structure has to be serialized in some form. Utilizing state-of-the-art character based line by line comparsion difference-tools as for instance used within Subversion (SVN[3]) or the GNU diff tool to compare serialized textual tree-structure representations most often does not add up. Even though most of them color-encode the character-based differences or provide other limited graphical representations of the computed differences they are not able to recognize the tree-structure and certain domain specific characteristics. For instance XML (Extensible Markup Language), which is a human readable meta markup language, exemplary for tree-structures in general and used in our prototype, has some inherent features which can not be recognized by such tools. Among those are the *lack of semantic differences* in case two XML documents only differ by an arbitrary amount of whitespace between attributes, namespaces[1] and elements or the permutation of attributes. Changes from empty elements to start tag, end tag sequences (`<root/>` to `<root></root>`) or inversely must not be recognized as a semantic difference as well. The major disadvantage however attributes to the tree-structure itself. Node-boundaries ca not be recognized as these tools incorporate no knowledge about the structure itself. Furthermore moves of nodes or subtrees and differences in the order of child nodes can not be detected. A comparison between two very simple XML documents (or two versions thereof) with GVim, which utilizes a line by line character based comparison algorithm is illustrated in Fig. 1. Several of the aforementioned deficiencies are depicted in this simple example.

[1] special kind of attributes

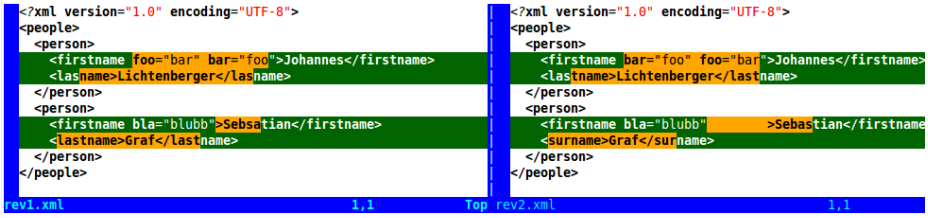


Fig. 1. GVim diff of two XML-document revisions illustrating the deficiencies of line by line character based diff-tools.

1.2 Problem Statement

Analysts often face the problem of having to compare large tree-structures. While coping with rapidly increasing amounts of data is effectively solved by means of Treetank, a tree-storage system used by our prototype, comparison requires sophisticated methods on top of it.

Generally two cases of tree-structures have to be distinguished which our system must be capable of.

- Tree-structures evolving naturally through applying changes.
- Similar tree-structures.

The research task adressed in this thesis is the problem of how to support analyts in comparing tree-structures.

1.3 Approach

A promising solution to the task at hand is to use methods from "Visual Analytics", a term coined by James J. Thomas in [4]. Thomas states that Visual Analytics is "the science of analytical reasoning facilitated by visual interactive interfaces". Thus we provide analytical methods which are inevitable for comparing tree-structures in the first place facilitated by an interactive visual interface. Furthermore interesting patterns can be revealed by custom `XPath` queries.

Whereas hierarichal visualizations have been studied for some time and sophisticated representations have been found, Visual Analytics of comparing tree-structures just recently gained momentum.

Value of visualizations Francis J. Anscombe reveals the value of graphs (which is generalizable to every (useful) kind of data-visualization) by illustrating in a simple example with four data sets (Anscombe's quartet), why graphs are essential to good (statistical) analysis. Using statistical calculations from a typical regression program (mean, variance, correlation and linear regression) shows that each computation yields almost the same result even though fundamental differences are visible on first glance once plotted. Furthermore human

brains are trained to interpret visual instead of textual content which is another facet illustrated by this example. It is almost impossible to gain further insights running through the printed out form of these four datasets [5].

Generalization and refinement of our research task While several data mining tools are available which specify on specific tasks, tree-structures are flexible and come in many flavors. XML is a meta markup language which is capable of describing all kinds of rooted, labeled trees. Thus it is used by our prototype. In fact it is a semi structured, flexible, meta markup language. XML-documents in stark contrast to relational data do not have to adhere to a schema, which has to be planned and implemented beforehand. Due to that it is mandatory that the visual interface offers great flexibility and thus is not restricted to a special use case.

The high level goal defined in section (1.2) can be divided into:

- Preprocessing and import of differences without having to rely on unique node identifiers.
- Structural comparison based on `insert-/delete-`operations.
- Comparison of non-structural data (for instance `TextNode` values).
- Extend with `replace`, `update` and `move`-operations (optional).
- Provide visualizations to quickly gain insights into which subtrees/nodes have been changed.

1.4 Contributions

The main aim of this thesis is the research and development of an interactive visual interface supporting analysts in comparing tree-structures along with analytical methods to compute the differences in the first place.

In a nutshell this thesis provides the following computer science contributions:

- Preprocessing of realworld XML data, for instance the revisioned import of (a small fraction of) *Wikipedia* and monitoring changes in a specific Filesystem directory.
- Several storage-enhancements of a database-system tailored to the storage of temporal tree-structures including compacting, a LevelOrderAxis and new edit-operations to support the implementation of an ID-based differencing algorithm and expressive visualizations. Furthermore a new bulk-insertion operation based on an existing component speeds up hashing of subtrees considerably from $O(n^2 + m)$ to $O(n + m)$ due to a simple postprocessing postorder traversal whereas n is the size of the nodes in the inserted subtree and m the number of ancestor nodes of the inserted node.
- Analytical methods (algorithms) to compute structural and non structural differences between similar or evolving tree-structures.
- Several views:
  - A `TextView` which serializes an aggregated tree-structure to a syntax highlighted XML output. Furthermore only the visible area plus additionally space to add a slider is filled.

- A `SunburstView` facilitating the comparison of tree-structures by a novel layout algorithm and several pruning techniques. Furthermore, interaction mechanisms as for instance zooming/panning, a fisheye view, support of XPath-queries and several other techniques are provided as well.
- A `SmallmultipleView` comprised of currently at most four sunburst small multiple displays supporting different modes (incremental, differential, a hybrid mode).

1.5 Conventions

Pseudocode which is used to illustrate algorithms in this thesis is based on a Java-like syntax as our prototype is based on Java. The following conventions in particular apply:

- The logical operator || from Java and other programming languages is denoted by *OR*.
- Similar the logical operator && is denoted by *AND*.
- Variable or reference assignments = are denoted by $\leftarrow$.

1.6 Outline

The thesis is structured as follows:

Chapter 2 describes essential preliminaries and provides an overview of algorithms to compute differences in tree-structures. Next, research efforts in visualizing differences of tree-structures are examined. The chapter concludes with a summary of the visualizations which are examined in respect to various attributes.

Chapter 3 starts off with a short description of numerous enhancements to our storage-backend which support an ID-less diffing algorithm as most tree-structures do not use unique node-identifiers. The algorithm (FMSE) matches nodes based on similarity-functions for leaf- and inner-nodes in the first place and modifies a tree with as few edit-operations as possible to transform the first tree into the second tree or the first revision/snapshot of a tree into the second in subsequent steps. Next, the implementation of FMSE is described. The algorithm is utilized to import differences stored as snapshots in our storage backend. Once the data is imported a diff-algorithm based on unique node identifiers can be utilized, which is described thereafter. The chapter concludes with performance measures of our node-identifier based algorithm and a short summary.

Chapter 4 is introduced with a short description of a tree-aggregation. Detailed descriptions of our visualizations follow. Furthermore several interaction mechanisms are examined.

Chapter 5 demonstrates the feasability of our approaches based on real world data.

Chapter 6 summarizes the results and discusses our approach in relation to the State-of-the-Art. It concludes with suggestions for future work.

2 Preliminaries and State-of-the-Art

2.1 Introduction

Today's storage capabilities facilitate the growing amount of data which is most often collected and stored without filtering or preprocessing. One of the consequences is the information overload problem defined as:

- Irrelevant to the current task at hand.
- Processed or presented in an inappropriate way.

To turn these issues into advantages the science called "Visual Analytics" recently became popular.

James J. Thomas and Kristin A. Cook coined the term "Visual Analytics"[4] and defined it as: "Visual analytics is the science of analytical reasoning facilitated by interactive visual interfaces." It combines (semi-)automatic analytical analysis with interatice visualization techniques, thus emphasizes both cognitive human and electronical data processing strenghts.

Whereas the information seeking mantra is described as "overview first, zoom/filter, details on demand" Keim et al defined the Visual Analytics mantra as:

"Analyse First - Show the Important - Zoom, Filter and Analyse Further - Details on Demand"[1]

This implies and confirms the important role of humans in the analysis process. As meantioned in the introduction humans are trained to interpret visual impressions but often fail in the same way to construe inappropriate representations (Anscombe's quartet is a very good example).

Our Visual Analytics pipline is largely influenced by the proposal of Keim et al. (depicted in Fig. 2).

Comparing tree-structures by a Visual Analytics approach requires analytical reasoning through the computation of differences in the first place. In order to support large tree-structures we decided to use a treebased storage system which is capable of storing transaction-time snapshots efficiently.

2.2 Storage backend

Treetank[6] is an effective and efficient secure storage system tailored to revisioned tree-structures. Currently it supports the import of XML documents which is commonly referred to as *shredding*. To process stored data the W3C recommendations XPath 2.0, XQuery 1.0 and XSLT 2.0[2], as well as a cursor like Java-API using transactions is supported. Treetank offers a pointer-based encoding and a transaction-based API to navigate between in the tree-structure or modify nodes similar to the Document Object Model (DOM[7]).

[2] parts of the XPath 2.0 recommendation have been implemented. Alternatively the Saxon binding can be used which also offers XPath 2.0, XQuery 1.0 and XSLT 2.0 support

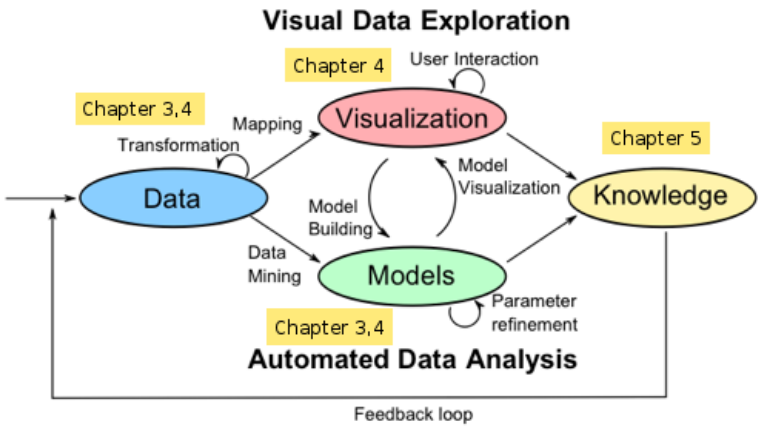


Fig. 2. Visual Analytics Process proposed by Keim et al. Presented in [1].

The architecture is based on three exchangeable vertical aligned layers, a transaction-layer, a node-layer and an I/O-layer. It supports *Snapshot-Isolation* through Multiversion Concurrency Control (MVCC) and thus multiple readers in parallel to currently at most one writer. Furthermore the well known ACID properties are supported (Appendix A.2). Several consistency checks as of now guarantee well formed XML during serialization. While importing large XMark-instances [8] which are commonly used for benchmarking we encountered a space-overhead due to our pointer based encoding. Appendix A.1 details a number of persistent storage enhancements.

One of the most interesting properties of Treetank for our purpose is *versioning*, that is storing and managing snapshots of tree-structures. Furthermore we utilize hashes of all nodes during database/resource generation, unique node-IDs as well as query-capabilities. However, Treetank does not provide any information about the changes of a node. The revisioning algorithms merge *NodePage*s, with the same unique ID together and override existing nodes with the latest version. NodePages organize a specified number of nodes in memory. Deleted nodes are introduced to guarantee the correctness after the merge-phase. The combination of NodePages relies on the specific revisioning algorithm. Thus, the merge-phase of NodePages usually refers to the latest full dump of all NodePages or the previous revision. As a direct consequence we are not able to simply utilize the page-layer. However a recent introduction of hook-mechanisms facilitates the indexing of changes to support comparing subsequent revisions (however, not yet implemented). Yet, such an index is only useful for comparing consecutive revisions.

The next section describes a few preliminaries for comparing tree-structures.

2.3 Analysis of differences

Line by line textual diffs are based on algorithms which solve the Longest Common Subsequence (LCS) problem. Whereas they are sufficient to track changes in flat text-files, tree-structures need more sophisticated methods as pointed out in the introduction.

The *Extendable Markup Language* (XML) is a textual data format for encoding and structuring documents in machine- and human-readable form. Its inherent data structure is a rooted, ordered, labeled tree.

Definition 1 *A rooted* ***tree-structure*** *is an acyclic connected graph, which starts with a root-node whereas every node has zero or more children and one parent-node with the exception of the root-node. We define a tree T as $T = (N, E, root(T))$ whereas N denotes all nodes, E denotes edges, the relation between child- and parent-nodes and $root(T)$ is a root-node.*

Definition 2 *A rooted, ordered, labeled tree is a tree-structure which extends the rooted tree-definition by defining a specific order for child nodes (that is extending the parent/child edge relation E). Furthermore each node has a label. Thus, T is an ordered, labeled, tree iff $T = (N, E, root(T), \Lambda(n) \in \Sigma)$. Σ is a finite alphabet and n is a node in the tree.*

Thus a tree is more restricted than a hierarchy based on a directed acyclic graph (DAG) in which every node except the root[3] has one or more parent nodes.

Many algorithms have been developed to determine differences in tree-structures for instance to provide deltas, which represent a compact version of the changes to the original document.

Next, some essential terms are defined to set the stage for the upcoming sections.

The tree-to-tree correction problem is to determine a (minimal) sequence or set of edit-operations to transform a source tree into a destination tree.

Definition 3 *An edit-operation is an atomar operation which changes a tree.*

A delta/edit-script is defined as:

Definition 4 *A delta/edit-script is a sequence or set of (elementary) edit-operations which when applied to one version v1, yields another version v2.*

Definition 5 *A symmetric delta is a directed delta which is invertible.*

In the following we use the term *delta* and *edit-script* interchangeably in the generic form meaning directed delta. Each edit operation is usually defined with a fixed cost (usually unit cost).

Definition 6 *A minimal edit script is a minimum cost edit script.*

[3] which has no parent

Besides providing a minimal or close to minimal edit script further metrics of a diff-algorithm are the CPU runtime and the compactness of the delta in terms of storage-space (e.g. it is in most cases sufficient to define edit operations on subtrees, such as a delete- and move-operation which usually removes respectively moves the whole subtree).

Some of the most popular approaches to detect differences in XML-documents which do not require unique node-identifiers (node-IDs) are described next.

DocTreeDiff[9] is designed for difference detection in document-centric XML documents. The algorithm computes the Longest Common Subsequence (LCS) on leaf nodes which are on the same level using hash-values. Subsequently ancestor nodes which differ are updated. Then inserted and deleted nodes are determined based on unmatched leaf nodes in the respective tree. As only leaf nodes in the first step are matching candidates which are on the same level, that is the number of nodes on the path up to the root node, the algorithm does not detect same subtrees if an inner node is added (or a leaf node is inserted and a subtree is moved and appended to the new node which is sufficient for a tree model which only allows insertions on leaf nodes). Furthermore moves are detected in a postprocessing step instead of applied on the fly. The runtime complexity is $O(leaves(T)D + n)$ whereas T is the sum of nodes in both trees and D is the number of edit operations. The space complexity is $O(T + D)$.

DeltaXML[10] Nodes are matched according to their type, the level in the tree and the Longest Common Sequence (LCS). Furthermore matching PCDATA nodes are optionally prefered. Similar, attribute-IDs in the deltaxml-namespace may be used to mark nodes with unique IDs. Marking all nodes with an ID generates a minimum edit script. However, a complete description of the algorithm is not published.

XyDiff[11] Cobena et al. present in [11] a fast heuristic algorithm in the context of Xyleme, an XML database warehouse. Signatures which are hash-values computed based on the value of the current node and all child signatures are used. Inserts and deletes are restricted to leaf nodes in the spirit of Selkow's tree-model [12]. Based on heuristics large subtrees are matched and depending on weights propagated to parent/ancestor nodes. In [9] several deltas are examined whereas the greedy subtree-algorithm yields large deltas. However, tuning parameters as the weights and how far to go up to match parent nodes is not considered.

The CPU runtime of the algorithm is $O(n \log n)$ and the space complexity is $O(n)$ whereas n is the size of both documents.

LaDiff / Fast Match Simple Editscript (FMSE)[13] operates on different versions of LaTeX documents. It is developed in the context of LaTeXto demonstrate and measure the feasibility of an approach to detect changes in

hierarichally structured information. We describe the algorithm in slightly more detail as we decided to implement the algorithm for XML data in Treetank.

Chawathe et al. divides this task into two main problems:

the Good Matching problem is the problem of finding matches between the two trees, which are either equal for some predefined function or approximately equal.

finding a Minimum Conforming Edit Script (MCES) is the second obstacle. An *edit script* is a sequence of edit operations transforming the source file into the target document once applied. Costs are therefore applied to every edit operation.

The algorithms used to solve these two problems operate on rooted, ordered, labeled trees. Four edit operations (`insert`, `delete`, `update` and `move` are defined with unit costs.

The algorithm proves to yield minimum edit scripts in case the assumption holds true that no more than one leaf node is considered equal to a predefined function which compares the values of leaf nodes and the labels match. XML does provide labels in the form of `QName`s for `element`- and `attribute`-nodes and a slightly restricted alphabet for `text`-nodes. Thus either text-node values have to be compared or `QName`s.

Thus the first criterium for leaf nodes is

$$compare(v(x), v(y)) \leq f \; such \; that \; 0 \leq f \leq 1 \tag{1}$$

Inner nodes are match candidates according to the forumla

$$\frac{|common(x,y)|}{max(|x|,|y|)} > t \; and \; label(x) = label(y) \tag{2}$$

$common(x,y) = \{(w,z) \in M | x \; contains \; w \; and \; y \; contains \; z\}$ whereas a

node x contains a node y if y is a leaf node descendant of x and $|x|$ denotes the number of leaf nodes x contains.

The threshold t is defined as $0.5 \leq t \leq 1.0$.

In a first step the good matching problem is solved by means of concatenating nodes/labels bottom up during a postorder traversal and finding a longest common subsequence (LCS). Furthermore, nodes which are unmatched during the LCS-matching are subject to a subsequent comparison. The predefined similarity-function is applied and if the nodes are matched they must be aligned.

After that in a breadth first traversal nodes are inserted, updated or moved. The children of each node are aligned based on the LCS once again. Nodes which are matched but not in the LCS are moved. The order in which operations are applied to the source tree and the edit script is crucial to the correctness of the algorithm. Details are omitted for brevity.

The second step is the deletion of unmatched nodes during a postorder-traversal.

When the assumption which assures the generation of a minimum edit-script does not hold which might be the case for several XML-documents, especially in data centric XML files, the algorithm yields large output-deltas due to mismatches according to Lindholm et al.[14] and Rnnau et al.[9]. This is a direct consequence of the ambigiouty of the LCS as well as of the subsequent matching of nodes. However this is a problem common to almost all differencing algorithms and can be minimized by proper definitions of the similarity functions for leaf- and inner-nodes. An optional postprocessing step reduces mismatches and thus move-operations such that children of matched nodes, which have not the same parent are candidates for matches with children of the same parent in the other tree, thus correcting some misaligned nodes. Note that this step can not reduce errors, which are propagated from mismatched leaf nodes.

It becomes apparent by investigating the analysis of Rnnau et al.[9] that the result of mismatched leaf-nodes in contrast to other algorithms yields many moves (compared to more inserts/deletes in other algorithms). However, moves in Treetank without using hashing and storing the number of descendants of each node is achieved in constant time ($O(1)$) due to the pointer-based approach. Even with hashing enabled which provides inevitable performance-boosts for our internal ID-based diffing algorithm which is used in a subsequent step by our visualizations, only ancestor nodes are affected and thus linear time is needed (however much faster than a deletion and insertion which in addition requires traversing the whole subtree two times).

The runtime complexity is $O(n * e + e^2)$ and the space complexity is $O(n)$ whereas n is the number of unchanged nodes and e is the number of different/changed nodes.

X-Diff[15] operates on unordered, labeled trees. Thus the order of child nodes does not matter. Despite using an unordered tree-model which is not suitable in many cases as for instance document centric XML, updates on element nodes to the best of our knowledge are not defined as the signature for `TextNode`s does not use their values. However updating an internal node is crucial due to otherwise potentially large subtree delete- and insert-operations even though only a single node is updated.

The runtime is defined for three involved steps separately:

1. **Parsing and Hashing**: $O(|T_1| + |T_2|)$
2. **Mapping**: $O(|T_1| \times |T_2|) \times max\{deg(T_1, deg(T_2)\} \times \log_2(max\{deg(T_1), deg(T_2)\}))$
3. **Generating Minimum-Cost Edit Script**: $O(|T_1| + |T_2|)$

Faxma[14] uses fast sequence aligning transforming the parsed documents into sequences of tokens. Subsequently the diff is computed using rolling-hashes with different window-sizes. Moves are handled through the combination of `delete/insert` pairs which is similar to the approach used by *DocTreeDiff*. The delta is a script which includes identifiers to matched nodes with inserted sequences in between. It is thus not defined in terms of an edit script and therefore not directly useful for our purpose.

algorithm	runtime	space	tree model	move support
DeltaXML	not published	not published	not published	not published
XyDiff	$O(n \log n)$	$O(n)$	ordered	yes
FMSE	$O(ne + e^2)$	$O(n)$	ordered	yes
X-Diff	$O(n^2)$	not published	unordered	no
DocTreeDiff	$O(leaves(T)D + n)$	$O(T + D)$	ordered	yes
Faxma	$O(n)$ (average) $O(n^2)$ (worst)	not published	ordered	no

Table 1. Comparsion of tree-to-tree difference algorithms.

Summary The problem in common to all approaches is to efficiently compute a minimum or near minimum edit script in order to transform the first into the second tree. Unfortunately a guaranteed minimum edit script for the the tree-to-tree correction problem is known to be bound in runtime by $O(nm\ min(depth(T1),\ leaves(T1))\ min(depth(T2),\ leaves(T2)))$, with n, m denoting the number of nodes of the trees $T1$, $T2$. Using heuristics speeds up the process but it does in most cases produce non optimal (minimal) edit scripts which might be counterintuitive to humans, because of mismatched nodes which have been changed. Every diff-algorithm has its strength and pitfalls. Depending on the input and expected modification patterns some algorithms provide better results than others. Even though algorithms are compared in [9] we are critical as the size of the delta and the amount of edit-operations might not be the best descriminators. All algorithms work best if leaf nodes can be descriminated very well. Comparing document oriented XML thus usually produces better results in comparison to data centric XML in terms of minimum or near minimum edit-scripts/deltas.

Memory consumption is very important considering large XML instances ranging from 10 MiB to 1 GiB and more. Reducing the cost of computing the LCS which has a large memory footprint might be mandatory but also results in heuristics. A survey of the wide range of algorithms is summarized in [16]. Several algorithms are described and compared according to the attributes memory consumption, time complexity, supported operations and the tree model.

In summary a trade-off between the minimality of edit scripts/operations and the memory consumption as well as the time complexity of the algorithm exists. Furthermore no algorithm exists which outperforms and in respect to the edit-script cost produces always better results than the others while comparing trees of different domains and characteristics. It heavily depends on the change pattern of the input document and parameters of the algorihm if provided.

2.4 Visualization of differences

In the past several visualization techniques have been proposed for hierarichal data ranging from simple node link diagrams, force directed layouts to space filling approaches. Comparison of tree-structures just recently gained momentum.

Treevolution[17] visualizes the evolution of hierarchical data in a radial node-link diagram. Each node might have arbitrary many parent nodes and each ring represents one snapshot of the hierarchy. Inserted nodes are placed on the appropriate ring depending on the time of insertion. However edge crossings occur frequently due to the parent-child relationship whereas often times nodes have more than one parent. A direct consequence is visual clutter which complicates the analysis of the hierarichal relationship between inserted nodes and their parents. Furthermore label overplotting occurs frequently which is a result from drawing the labels in one direction (left to right), however a simple interaction method improves on this by providing rotation mechanisms.

Interactive Visual Comparison of Multiple Trees[18] The authors propose a prototype to compare multiple phylogenetic trees. Several views are available to analyse the trees on different levels of detail. A matrix view for instance displays pairwise tree-similarities based on a similarity score which takes overlapping subtrees into account. The similarity score depends on all nodes in a subtree including inner nodes instead of just determining overlapping leaf nodes. A histogram shows the score distribution among all nodes in all trees. The consensus tree is "a compact form of representing an 1:n comparison" in one tree. The score is "the average of the scores comparing a reference tree node against its best matching unit in all other trees". The last view is a Tree Comparison View which highlights all nodes in the subtree a user marks through a linking and brushing technique in all other trees. It is the only system which is capable of comparing multiple trees on different levels of detail at the same time with linked visualizations. However we assume that the quadratic runtime of comparing all nodes with all other nodes will be restricted to (many) small trees. Furthermore to the best of our knowledge it is not meantioned how nodes are compared, but we assume unique labels or node identifiers are required as the prototype is proposed for phylogenetic trees.

Spiral-Treemap/Contrast-Treemap[19] A Treemap is a space filling approach which maximizes available screen space for the visualization. Most treemap layouts suffer from abrupt significant layout changes even if the underlying data changes were rather small. Tu et al. propose a new layout algorithm called *Spiral Treemap* to improve the layout stability arranging child nodes in spirals which change the orientation by 90 in the corners. Child-nodes are aligned along a spiral in each level beginning at the upper left corner. Therefore edit-operations as for instance inserts and `deletes` only affect local regions. However, we argue that it is not trivial to analyse strutural differences as they are not explicitly visualized in the Contrast Treemap and the texture distortion depends on the layout algorithm, whereas small changes are hardly visible. Furthermore the hierarchy is not easy to recognize in the Spiral Treemap which is a common problem for Treemaps owed to the enclosing nature of child nodes in their parent nodes. Labels however are easily perceived if the rectangles are not too thin which occurs frequently in large trees ranging from about 50000 nodes to a few

hundred or even millions of nodes. Furthermore the nodes degenerate from rectangles to very tiny stripes. Improving the aspect ratio of the rectangles results in Squarified Treemaps[20], which lack the property of ordered siblings. However child-nodes in trees are often ordered which is why Squarified Treemaps in general are only feasible in certain specific cases which do lack a semantic difference in node ordering.

TreeJuxtaposer[21] TreeJuxtaposer is a system designed to support biologists to compare the structures of phylogenetic trees. A new comparsion algorithm to determine matching nodes in near-linear average time is proposed. Perfect matching nodes have the same labels for each of their leaf nodes. Based on a simple similarity measure ($S(A, B)$ between two sets whereas A, B is defined as $\frac{A \cup B}{A \cap B}$) they propose a method to colorize edges of non perfectly matching nodes and a rectangular magnifier to emphasize changed nodes. The visualization itself contains several revisions side by side plotted in node-link diagrams. Selections and rectangular magnifications are synchronized. TreeJuxtaposer uses a node-link algorithm and thus shares the drawbacks of other node-link visualizations such as *Treevolution* and the *Ripple presentation*. In comparison to space filling approaches further attributes as for instance value comparisons, subtreesizes and labels are not visualized or result in visual clutter. Furthermore the fast differencing algorithm to the best of our knowledge relies on unique node labels to support the region query on a two-dimensional plane.

Code Flows: Visualizing Structural Evolution of Source Code[22] Code Flows is proposed for determining and tracking changes in source code between several revisions. It is a space filling approach which uses horizontally mirrored icicles and therefore certain attributes of nodes can be visualized besides highlighting actual tree changes. Labels are readable in smaller trees or when zoomed in because of the rectangular layout of icicle plots. Due to spline-tubes matching, nodes can be tracked very well through different revisions. Even code splits and merges are easily trackable. On the downside small code changes resulting in the addition or deletion of a few nodes might not be visible at first glance. Furthermore the spline connections between matched nodes leads to visual clutter due to overplotting when nodes are moved.

Ripple presentation for tree structures with historical information[23] The Ripple presentation visualizes both evolving hierarchies and categories. Concentric circles are used to indicate an evolving hierarchy through time. Each circle represents one point in time. Nodes are plotted in a special node-link layout. The root node of each subtree is in the focus of the view. Leaf nodes are arranged in ascending order meaning older nodes are drawn on circles further away from the current root of the subtree. The angles of edges are application dependent and facilitate the clustering of categories through time. In their news articles example categories can be extracted from the content. For each child belonging to

the same category the angle of the edge has to be located in between the parent angle. Since the application examples require no diff-calculation and updates as well as deletions of nodes are not considered it is not useful to compare every aspect of changing tree structures. It suffers from a lot of clutter consequent to label overplotting as well. Deletions and updates have not been considered since the example use cases to the best of our knowledge just add nodes and categories. Due to the fact that it is also a node link representation and not a space filling approach attributes of nodes can not be visualized. Thus it is best comparable to Treevolution, but because of the more complex layout algorithm it can group nodes according to categories.

Summary Recently, interactive visualizations of the evolution of tree-structures or different, similar trees have been proposed. Table 2 summarizes the visualizations according to several attributes. The first column denotes the readability of the hierarchy-representation. The second column indicates if a space filling approach is used and to which extend the whole display space is utilized. The third column characterizes the readability of the visualization including the encoding of differences and labels. The last column is the most significant. It determines to which extent changes are visualized. Even deletions are not considered in some cases which might be due to the use cases of the respective visualization. Note that the scala range from "-", not present to "+++" (best). Chapter 6 presents a detailed more comprehensive analysis in comparison to our system.

Most of the visualizations are tailored to specific tasks and thus at best only partielly useful for other applications. In fact besides adapting the diff-algorithm described in [22] none of the proposed systems to the best of our knowledge is able to compare every kind of tree structure due to diff-algorithms which rely on domain characteristics as unique node identifiers/node labels or on change detections which hook into a system. Furthermore we suppose that except TreeJuxtaposer and CodeFlows no other system is able to compare large trees. However, in CodeFlows the filtering of nodes depends on the level of detail (per class-level, function-level...). Thus a global view which filters relevant nodes is not available.

[18] is the only system capable of visualizing differences at various levels of detail due to linked visualizations.

[4] due to the side by side view which needs a lot of space

visualization	hierarchy	space filling	readability	changes
Spiral-/Contrast-Treemap	+	++[4]	++	++
Treevolution	+	-	+	+
Code Flows	+++	++	++	++
Juxtaposer	++	-	++	+++
Ripple Presentation	+	-	+	+
IVCoMT	+++	-	++	++

Table 2. Comparsion of tree-to-tree comparison visualizations. "-" indicates the absence of an attribute, "+" to "+++" implies how well or bad the attribute is supported.

3 Analysis of structural differences

3.1 Introduction

This chapter describes the implementation of an algorithm which does not rely on unique node identifiers (FMSE, described in general in chapter/section (2//2.3)) as well as a new ID-based algorithm which utilizes a preorder traversal on both trees to compare tuples of two nodes each time. The FMSE algorithm facilitates the import of differences between two tree-structures which do not incorporate unique node-IDs in the first place either to compare different, similar tree-structures or the evolution of a tree. Thus, our similarity measures are based on the tree-to-tree correction problem. The visualizations proposed in the next chapter rely on the diff-algorithms described in this chapter which detect edit-operations/diff-types to transform a source tree into a target tree. A reference tree is initially imported in Treetank, our storage backend. Subsequently, changes between this tree and either other trees or the evolution of the reference-tree in terms of edit-operations are stored as subsequent revisions. As described in the last chapter, the Fast Matching Simple Edit-Script algorithm depends on similarity-measures and does not require nor use unique node-IDs in our case. Thus, a minimum edit-sequence usually is not guaranteed if leaf nodes are very similar. While importing the differences through FMSE, the storage system Treetank assignes unique stable node-IDs which are subsequently utilized by our ID-based diffing algorithm to support a fast linear-runtime difference-computation. Fig. 3 describes our approach, the import using FMSE and a subsequent invocation of a fast ID-based diff-algorithm. During the import of collections, that is either multiple revisions of one tree or different similar trees, changes are computed by comparing the latest stored revision in the database backend with either the next tree-revision or (similar) tree to import. Once imported, unique node-IDs and optionally hashes facilitate a new fast diff-algorithm. Diff-tuples are generated by comparing two nodes each time. They include the type of difference, the compared nodes and their depth in the tree and facilitate an aggregated tree-structure made up of both changed- and unchanged-nodes through collecting the diff-tuples in the model of a visualization as described in the next chapter (Chapter 4).

Both algorithms, the FMSE algorithm, which does not require unique node-IDs, used for importing differences and the fast ID-based diff-algorithm are implemented using Treetanks' transaction-based Java-API, the native secure tree-storage system, which is used as an integral part to demonstrate our approach. First, preliminaries such as new edit-operations required for implementing the FMSE-algorithm and a compact, meaningful aggregated tree-structure are described. Note that a rich set of edit-operations facilitates an expressive visualization. It is a lot more intuitive and meaningful to provide atomar `replace`- and `move`-operations to reflect changes between tree-structures than using combinations of `delete`- and `insert`-operations and thus losing any association. Once, having described preliminaries the implementation itself is sketched. A description of our internal ID-based diffing algorithm follows. The chapter concludes by

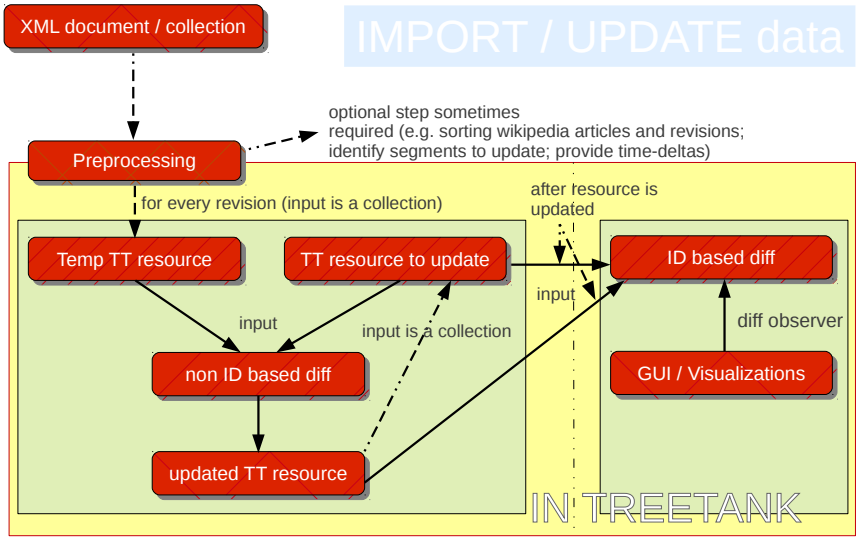


Fig. 3. Importing differences encountered through the FMSE ID-based diffing-algorithm.

analysing the asymptotic bound of the algorithm both in terms of the runtime as well as the space consumption and synthetic, empirical benchmarks which confirm the scalability of our algorithm.

3.2 ID-less diffing (FMSE) / Preprocessing

Preprocessing of raw data is a major task in every data processing pipeline. Besides data specific preprocessing, databases/resources which do not evolve through the Java-API of Treetank have to be imported. Note that it is very common to simply dump full revisions of temporal data, thus most often no direct deltas are provided which just have to be applied to a base revision. Furthermore our prototype must be capable of comparing similar distinct trees. In both cases often times unique node-identifiers are not provided and therefore the trees or revisions thereof must be compared using tree-to-tree comparison heuristics which try to determine and match the most similar nodes/subtrees.

As described in the introduction the Fast Matching Simple EditScript (FMSE) algorithm is implemented. The reasons for choosing FMSE are based on three properties: (1) it is successfully implemented a few times (specifically for XML-documents) [24], [25], (2) it utilizes Treetank's cheap move-operation and (3) supports applying edit-operations/changes on the fly. The move-operation is one of the new edit-operation Treetank supports and very lightweight. It is defined for subtrees. Only local nodes (parent, siblings, first-child) are affected as

well as the ancestor nodes of the node which moves (before and after the move). Ancestor nodes are only modified if hash-values are enabled which represent a fingerprint of the whole subtree and support a faster version of our ID-based diffing algorithm described in the next section.

The Nodes are matched based on a bottom-up traversal searching for the Longest Common Subsequence (LCS) of matching nodes on each level. Predefined functions determine the similarity of nodes/subtrees as described in Chapter 2 which are utilized by the LCS-algorithm to determine matches. Unmatched nodes after determining the LCS are examined for crossmatches (moves). The algorithm not only facilitates the analysis of temporal evolving tree-structures but also the comparison of similar distinct trees. To support the FMSE implementation and expressive visualizations Treetank is enhanced in several ways. The following new operators/methods and components are available:

- `LevelOrderAxis` which incorporates attribute- and namespace-nodes if desired.
- `copy-operation` to copy nodes/subtrees of other *database/resource*-tuples.
- `move-operation` to move nodes/subtrees in the currently opened *resource*.
- `replace-operation` to replace a node and its subtree with another node/-subtree.
- Visitor pattern support for nodes/transactions.
- Merging or avoidance of adjacent text nodes at any time.

The `LevelOrderAxis` and the other operations are described in Appendix A.3 and A.4. The granularity of the operations are important:

- `insert`-operations are defined on leaf nodes. It is not possible to insert inner nodes, however this is only an implementation restriction and simplifies our identifier-based diffing algorithm. Due to the pointer-based encoding of Treetank, however it is possible to include an insert-operation to insert nodes between inner nodes in constant time. Currently, the only workaround is an insert followed by a move.
- `remove`-operations always delete the whole subtree of the node to remove.
- The `update`-operation is defined on inner- as well as on leaf-nodes and updates only a single node.
- The `move`-operation is defined on inner- as well as on leaf-nodes and moves the whole subtree.
- The `replace`-operation is defined on inner- as well as on leaf-nodes whereas the whole subtree of the node to replace is removed and thus replaced by the new node or subtree.
- The `copy`-operation is defined on inner- as well as on leaf-nodes and also copies whole subtrees.

Having described the preliminaries the next section describes the FMSE implementation itself.

FMSE The FMSE implementation first saves node-types and the according node-IDs in two associative arrays during a postorder traversal. Next, the algorithm determines a longest common subsequence of matching nodes. Leaf nodes are compared first, then inner nodes. Thus the inorder-traversal described in [13] must be replaced by a postorder-traversal. Otherwise, some leaf nodes required to determine the similarity of inner nodes are not processed beforehand. The matching of nodes involves two different similarity-metrics as described in the last chapter. However our implementation bears some explanation, as the matching is crucial and the changes applied by FMSE are propagated to a subsequent ID-based diffing-algorithm:

- `TextNode`s are matched based on their String-value. The Levenshtein algorithm is used to compute a similarity measure of the values, which counts update costs of individual characters normalized between 0 and 1. `QName`s of `Namespace`- and `attribute`-nodes are matched first based on equality. In case of attributes afterwards their value is compared yet again using the Levenshtein algorithm in addition to their ancestor-elements.
- `ElementNode`s are compared based on the number of matched nodes in their subtree. Thus not only leaf nodes are compared as suggested in the paper. Recapitulate that all node-types are chained for the fastMatching-algorithm bottom up during a postorder traversal. Empty elements however are compared based on their `QName` similarity, whereas all ancestor nodes are compared, too once more utilizing Levenshtein. This ensures the possibility of matching empty-elements after a deletion or insertion of a subtree. Treating empty nodes as leaf nodes otherwise prohibits matching empty `element`-nodes with other `element`-nodes which include a subtree because leaf nodes and internal nodes are compared in different successive steps and thus are not cross-compared. Matching nodes are stored in a `BiMap` containing forward and backward matchings of nodeKeys.

After storing matching node-IDs, FMSE step one is implemented straight forward. However whenever an `attribute`- or `namespace`-node is determined to be moved it is deleted from the old parent and inserted at the new parent node as moves of these node-types are not permitted by Treetank. Another noteworthy subject regarding moves is, that deleted text nodes in case adjacent nodes are collapsed and must be removed from the mapping as well. Due to adding the consistency constraint that *never*, before and after a commit, duplicate attributes with the same `QName` are permitted, a new attribute value is set in the `WriteTransaction.insertAttribute(QName, String)` method instead of adding a new one if the `QName` of the attribute to insert is identical to another attribute-node with the same parent. This also saves from time overhead due to node-creation. This case occurs whenever the attributes with the same `QName` and parent node are not matched because of very different attribute-values or parent `QName`s. All updated or inserted nodes are added to the matches as described by Chawathe et al. in [13] to prevent them from deletion in the next step.

The second FMSE step, which deletes non matching nodes with their whole subtrees, involves a preorder traversal of the tree. Thus a new `VisitorDescendantAxis` which optionally expects a visitor instance is implemented [5] and detailed in Appendix A.5.

The following cases have to be distinguished. The node to move

1. has no right- and no left-sibling
2. has no right- and no left-sibling but the parent has a right-sibling (the parent must be removed from a stack which is used to save right-siblings for nodes which have a first child.)
3. has a right- and a left-sibling
4. has no right- but a left-sibling
5. has a right- but no left-sibling

Two variants of the third case are depicted in Fig. 4. In case the node to delete has two neighbour nodes which are `TextNode`s (node 4) the right sibling value is appended to the left `TextNode`. The right sibling node is removed from the storage afterwards, too. Next, the transaction is located at the updated left sibling text node. Thus the preorder traversal in the `VisitorDescendantAxis` continues without skipping any nodes. Otherwise if no adjacency text nodes are merged during the remove()-operation (node 9) the transaction is moved to the right-sibling before the operation is finished. Thus, the transaction first has to be moved to the left sibling before the `VisitorDescendantAxis` moves the cursor to the right-sibling. In this case the subtree of node 8 must be skipped as it is processed before.

3.3 ID-based diffing

Once revisioned data is stored in Treetank the main task is to reveal and present structural differences of the tree-structures. Treetank supports collections in form of databases which include one or more resources. Due to stable unique node IDs in each resource every kind of tree-structure is imported updating a single resource with the computed changes using FMSE. Even similar distinct tree-structures are imported updating a single resource. Otherwise, we would not be able to utilize unique node-IDs as different resources do not share unique node-IDs.

A fast linear time diff-algorithm utilizes these unqiue node-IDs and optionally hash-values which represent the content of the entire subtree rooted at a specific node. Note that the algorithm is designed to be able to compare any two revisions and thus not just consecutive revisions. It compares two nodes each time and determines the type of diff.

[5] Visitors are always preferable to other methods if algorithms depend on the specific node-types, due to runtime errors during downcasts or possibly long chains of `instanceof` checks

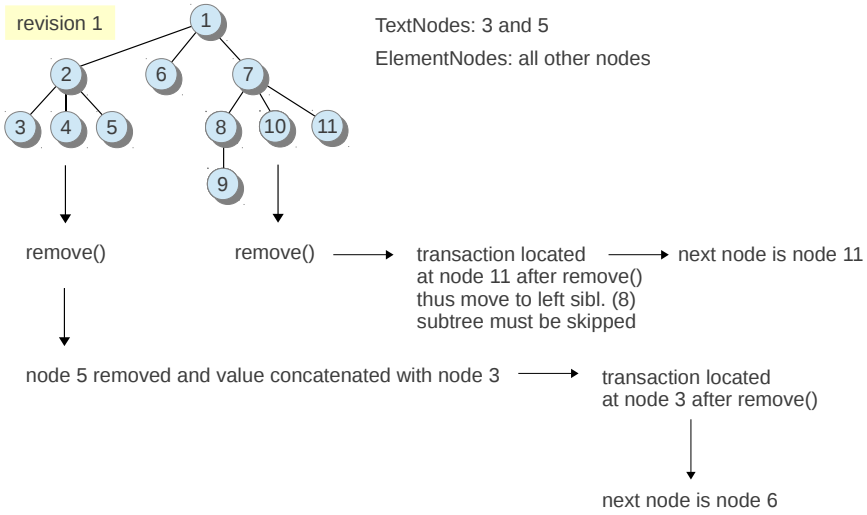


Fig. 4. Deletion visitor; two variants are depicted for the case that the node to removed has a left- and a right-sibling. Either both sibling nodes are `TextNode`s as is the case for node 4 or not (node 10).

Hashes One of our goals is the efficiency of our approach as it is called by our interactive visualizations described in the next chapter. Hashes of all nodes are generated during import in a postorder-traversal and an ancestor-traversal if subtrees are inserted. A new bulk insertion mode in Treetank speeds up the import and reduces the asymptotic bound from $O(n^2+m)$ to $O(n+m)$ whereas n is the number of nodes to import and m is the number of ancestor nodes for which the hash-values have to be adapted. They are build incrementally based on all nodes in a subtree bottom up. Two types of hashes are available, rolling[26]- and postorder[27]- hashes. *Rolling* hashes only affect the inserted or updated nodes on the ancestor axis whereas *postorder* hashes also affect nodes in a postorder traversal starting at the current node. The hashes are optionally used to speed up our algorithm. Whenever identical hashes are determined the nodes are matched and the two transactions opened on the two revisions and used to iterate over both revisions are moved to the next node in document order, which is not a descendant of the current node. Thus the transactions move to the first node in the XPath `following::`-axis. Hence, whole subtrees are skipped from traversal. The hashes include the unique node-IDs as well as node specific content. The hashing-function is designed to be fast and to reduce collisions to a minimum. Even if hash-collisions which are extremely unlikely appear it is not possible to match subtrees with identical hash-values as the node-IDs are also compared which are stable and unique during all revisions. Rolling-hashes are enabled by default during the database/resource creation and optionally used

by our diff-algorithm. It is for instance used by an optional pruning of the tree in a Sunburst-layout to speed up the computation as well as the construction of the visualization. An in depth explanation of this application is provided in Chapter 4. The next subsection briefly describes two modes of the algorithm.

Modes to compare differences Interested observers are notified of the diff between two nodes through registration and the implementation of a special interface method. Currently two modes are available.

- *Structural Diff* calculates changes without comparing attribute and namespace nodes. This implies that whenever the overall structure is crucial this algorithm should be chosen.
- *Full Diff* takes structural nodes as well as attribute and namespace nodes into account. However currently we do not emit non-structural changes. Changes in `namespace`- or `attribute`-nodes result in an `UPDATED` parent-element. This restriction applies as the *SunburstView* which is described in Chapter 4 currently does not include special `namespace`- or `attribute`-items. Instead these are part of the element item and shown on mouseover.

The following diff-types are supported and emmited:

- `INSERTED` denotes that a node is inserted.
- `DELETED` denotes that a node is removed.
- `UPDATED` denotes that a node is updated, that is either the QName of an `element`-node is updated or the value of a `text`-node.
- `SAME` denotes that a node is not changed.
- `REPLACEDOLD` denotes that a node or subtree is replaced (the old node/subtree).
- `REPLACEDNEW` denotes that a node or subtree is replaced (the new node/subtree).
- `SAMEHASH` denotes that a node is not changed and the hashes of the subtrees are identical.

Note that the differentiation between `REPLACEDOLD/REPLACEDNEW` supports an expressive aggregated tree-structure used as an underlying model of the visualizations. However it is only a simple heuristic. Two other diff-types are supported by an optional post-processing step.

- `MOVEDFROM` denotes that a node or whole subtree has been moved from this location to another one.
- `MOVEDTO` denotes that a node or whole subtree has been moved to this location.

The types are splitted, too to indicate the movement of the node, the old place before the move and the new place in the aggregated tree-structure.

The next section describes the preorder-traversal in both revisions which is essential to compare the right nodes each time.

3.4 Traversal of both revisions

The algorithm to traverse the trees and to compute the differences between two nodes in each revision is depicted in algorithm 1. First, the method `treeDeletedOrInserted(IReadTransaction, IReadTransaction)` checks if both transactions opened on each revision can be moved to the start node. If not, either the node is inserted or deleted depending on the transaction which can not be moved.

Let's examine both cases:

- The transaction opened on the older revision can not be moved to the start node. This implies that the tree in the new revision has been inserted.
- The transaction opened on the newer revision can not be moved to the start node. This implies that all nodes in the old transaction have been deleted.

The distinction is used to support the selection of modified nodes in the visualizations which are described in Chapter 4 and only affects subtrees. Otherwise simply put all nodes in the old revision are deleted, whereas all nodes in the new revision are inserted.

If the root-nodes of both revisions are selected by the transactions they move forward in *document order* (depicted in Fig. 5) depending on the last encountered kind of diff between two nodes. Document order is identical to a preorder traversal of a tree. In case of an insert, the transaction opened on the new revision is moved forward, in case a delete is encountered the transaction opened on the old revision is moved forward (the `moveCursor(IReadTransaction, ERevision)`-method). In both cases the whole subtree is emitted (for instance node 10 and its subtree in Fig. 5 is deleted). If a node is updated or is not changed at all both transactions move to the next node in document order. Once the traversal in one of the two revisions reaches the end of the tree, the transaction is located at the document root. The diff-calculation ends if either the transaction on the older revision is located at the document root and the last encountered type of diff was `DELETED` or both transactions are located at their document-root nodes. Note that if the transaction on the newer revision is located at the document root, but the transaction on the old revision is not the following nodes are `DELETED` at the end of the tree and have to be emitted as such (lines 22-28; node 15 in Fig. 5).

3.5 Diff-Computation

Besides moving both transactions forward in document-order depending on the last encountered type of diff the algorithm which determines the type of difference itself is crucial. The computation is the main task of the `diff(INodeReadTransaction, INodeReadTransaction, Depth)`-method outlined in algorithm 2. It is invoked whenever the node-IDs or the `QNames`/`Text`-values of the nodes to compare differ.

First, the depths of the nodes have to be compared. The depth is the sum of nodes in the path up to the root-node. When the depth of the node in the old

Algorithm 1: ID-based diff: traversal

input : HashKind mHashKind, long pOldRevKey, long pNewRevKey, long mOldStartKey, long mNewStartKey, DiffType pDiffType, DiffTypeOptimized mDiffKind, ISession mSession
output: for each node comparsion: DiffType diffType, IStructNode oldNode, IStructNode newNode, Depth depth

```
1  INodeReadTransaction rtxOld ←
   mSession.beginNodeReadTransaction(pOldRevKey);
2  INodeReadTransaction rtxNew ←
   mSession.beginNodeReadTransaction(pNewRevKey);
3  // moveTo(long) returns true in case the transaction could be moved
      to the node or false otherwise.
4  newRtxMoved ← rtxNew.moveTo(mNewStartKey);
5  oldRtxMoved ← rtxOld.moveTo(mOldStartKey);
6  treeDeletedOrInserted(newRtxMoved, oldRtxMoved);
7  DiffType ← null;
8  // Check first node.
9  if mHashKind == HashKind.None OR mDiffKind == DiffTypeOptimized.NO
   then
10 |  diff ← diff(rtxNew, rtxOld, depth);
11 else
12 |  diff ← optimizedDiff(rtxNew, rtxOld, depth);
13 // Iterate over new revision (order of operators significant --
      regarding the OR).
14 if diff != DiffType.SAMEHASH then
15 |  while (rtxOld.getNode().getKind() != ENode.ROOT_KIND AND diff ==
   |  DiffType.DELETED) OR moveCursor(rtxNew, ERevision.NEW) do
16 |  |  if diff != DiffType.INSERTED then
17 |  |  |  moveCursor(rtxOld, ERevision.OLD);
18 |  |  if rtxNew.getNode().getKind() != ENode.ROOT_KIND or
   |  |  rtxOld.getNode().getKind() != ENode.ROOT_KIND then
19 |  |  |  if mHashKind == HashKind.None OR mDiffKind ==
   |  |  |  DiffTypeOptimized.NO then
20 |  |  |  |  diff ← diff(rtxNew, rtxOld, depth);
21 |  |  |  else
22 |  |  |  |  diff ← optimizedDiff(rtxNew, rtxOld, depth);
23 |  // Nodes deleted in old rev at the end of the tree.
24 |  if rtxOld.getNode().getKind() != ENode.ROOT_KIND then
25 |  |  emitOldNodes(rtxNew, rtxOld, depth);
26 done();
```

Algorithm 2: ID-based diff: diff-computation

input : Depth pDepth, INodeReadTrx pOldRtx, INodeReadTrx pNewRtx
output: kind of diff (DiffType enum value)

```
1  DiffType diff ← DiffType.SAME;
2  // Check if node has been deleted.
3  if pDepth.getOldDepth() > pDepth.getNewDepth()) then
4      diff ← DiffType.DELETED;
5      cumulatDiffTypes(diff);
6      if checkReplace(pNewRtx, pOldRtx) then
7          diff ← DiffType.REPLACED;
8  // Check if node has been updated.
9  else if checkUpdate(pNewRtx, pOldRtx) then
10     diff ← DiffType.UPDATED;
11 // Check if node has been replaced.
12 else if checkReplace(pNewRtx, pOldRtx) then
13     diff ← DiffType.REPLACED;
14 else
15     long oldKey ← pOldRtx.getNode().getNodeKey();
16     boolean movedOld ← pOldRtx.moveTo(pNewRtx.getNode().getNodeKey());
17     pOldRtx.moveTo(oldKey);
18     long newKey ← pNewRtx.getNode().getNodeKey();
19     boolean movedNew ←
       pNewRtx.moveTo(pOldRtx.getNode().getNodeKey());
20     pNewRtx.moveTo(newKey);
21     if !movedOld then
22         diff ← DiffType.INSERTED;
23     else if !movedNew then
24         diff ← DiffType.DELETED;
25     else
26         // Determine if one of the right sibling matches.
27         EFoundEqualNode found ← EFoundEqualNode.FALSE;
28         long key ← pOldRtx.getNode().getNodeKey();
29         while pOldRtx.getStructuralNode().hasRightSibling() AND
           pOldRtx.moveToRightSibling() AND found ==
           EFoundEqualNode.FALSE do
30             if checkNodes(pNewRtx, pOldRtx) then
31                 found ← EFoundEqualNode.TRUE;
32                 break;
33         pOldRtx.moveTo(key);
34         diff ← found.kindOfDiff();
35     cumulatDiffTypes(diff);
36 return diff;
```

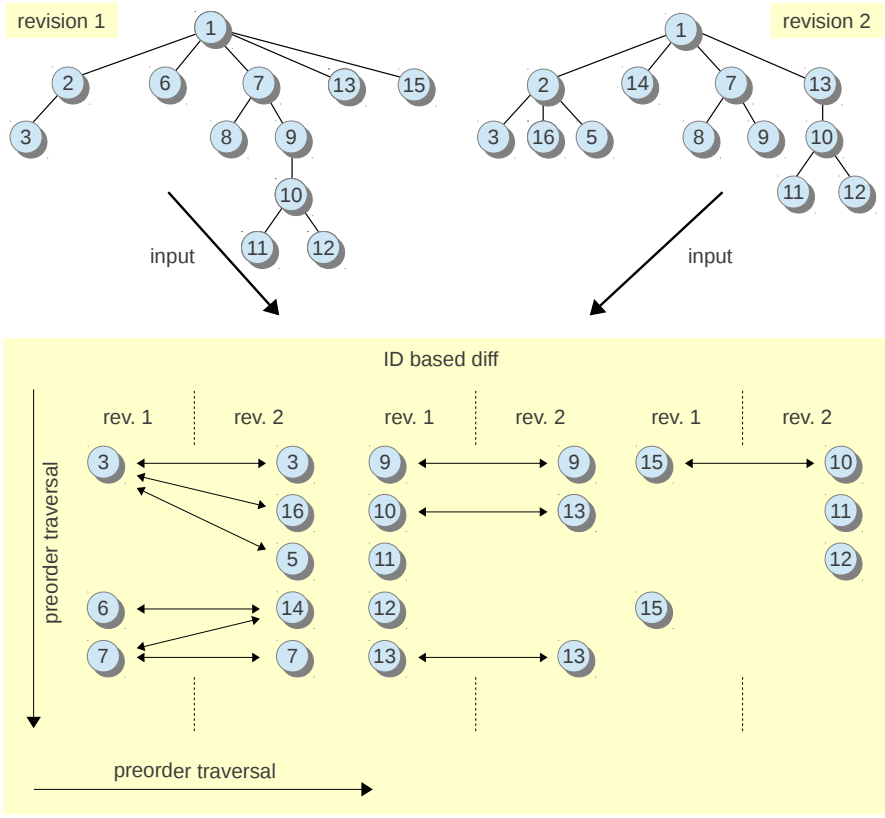


Fig. 5. ID-based diffing.

revision is greater than the depth of the node in the new revision it must have been deleted. Note that the depths are not persisted, thus counters have to keep track of the current depths. All diff-tuples which are of type `DiffType.DELETED` or `DiffType.INSERTED` are saved in a Java `List`. A second datastructure is used to gather the diff types itself, that is a whole subtree which is either inserted or deleted is cumulated by the according diff-type. This datastructure is used to find `INSERTED`, `DELETED` combinations which are instead emitted as of type `DiffType.REPLACEDOLD` (the deleted tuples) and `DiffType.REPLACENEW` (the inserted tuples).

When the depth of the node in the old- and the node in the new-revision instead either is identical or the depth of the node in the more recent revision is greater it is checked whether the node is updated through comparison of the node-IDs and the depths/levels of the nodes. Unless the node is updated the node is examined for replacement. Therefore the datastructure which keeps track of deleted- and inserted-subtrees is reviewed. Consecutive insert/delete-

or insert/insert/delete/delete-tuples are emitted as replaced subtrees. Note that this replace-detection is just a simple heuristic and currently does only detect the aforemeantioned pairs of inserted and deleted nodes.

Assuming a node or subtree is not replaced, it must be decided if either the current node in the new revision is inserted or the current node in the old revision is deleted. First, it is determined if the transaction opened on the old revision is moveable to the current node in the new revision. If not it is immediately obvious that the node is inserted. Otherwise, if the transaction opended on the new revision can *not* be moved to the current node in the old revision it must be a deleted node. For the simple reason that move-operations are supported both checks might succeed. In this case the right siblings of the node in the old revision have to be examined in order to determine the type of diff, until one of them matches the node in the new revision, that is the node-IDs are identical or no more right siblings are available. In the first case the new node is inserted. Otherwise it is deleted.

3.6 Detecting moves

An optional postprocessing step is required to detect moves. The two basic diff-types `MOVEDFROM` and `MOVEDTO` are detected after all operations are emitted. The detection requires three datastructures to store all diff-types, the inserted nodes and the deleted nodes.

Impossibility to detect moves on the fly Note that it is not possible to include the detection of moves in the preorder-traversal of both revisions itself, as it is not known in advance which of the two nodes is the one which is moved and which one is the node which is unchanged. However this is required to determine which of the two transactions must be moved to the next node in document order. All we can argue is, that it is possible to detect a move itself if the transaction on the new revision is moveable to the current node of the transaction in the old revision and vice versa. That is all `DELETED` or `INSERTED` nodes have already been emitted, meaning that one of the two nodes which are currently compared must have been moved and the other must have been unchanged. It is not possible to decide which one of the two nodes stayed the same and which one has been moved. The position in the tree is no implication wheter a node has been moved from or moved to another place, but this is crucial to decide which of the two nodes has not changed and which one actually has been moved. As thus the types have to be matched. Whenever the unique nodeKey of a `DELETED`-node matches the key of an `INSERTED`-node, the corresponding diff-types can be changed into `MOVEDFROM` and `MOVEDTO`. The next subsection details a postprocessing algorithm which is based on this idea.

Detection of moves in a postprocessing step All encountered diff-types are saved in an associative array, a map (index of their encounter ⇔ diff-tuple) [6] in preorder which is the order in which they are observed. Additionally `DELETED` and `INSERTED` nodes are recorded mapping their unique node-ID to the index in the original map with all entries. The algorithm described in 3 expects the three maps. It tries to match `INSERTED` ⇔ `DELETED` pairs and vice versa and checks whether the diffType in the map needs to be adjusted to `MOVEDTO` or `MOVEDFROM` (or not at all). A map which does not contain a value for the specified key returns the special value `null`. First, the old node-ID (might have been deleted) is searched for in the Map containing all inserted tuples. If the key is found (value ! = null) the type is checked. If the current diff-tuple is of type `DELETED` or `MOVEDFROM` the diff type is set to `MOVEDTO`. Note that the check for the `MOVEDFROM` type is necessary as the corresponding `INSERTED` tuple might have been encountered before and thus the type has been changed to `MOVEDFROM` already. The following is the inverse case to set the `MOVEDFROM` type if necessary. Furthermore a link in the form of the index of the matching node with the same node-ID and the corresponding `MOVEDTO` diff-type is additionaly saved. Note that the algorithm does not detect `text`-nodes which are moved to a right sibling of another `text`-node. In this case our implementation of the `moveSubtreeToRightSibling(long)` prepends the value of the current `text`-node to the moved `text`-node and subsequently deletes the current node. This ensures, that no two text-nodes are ever adjacent which otherwise contradicts the XQuery/XPath data model (XDM). In this particular edge case the algorithm determines a `REPLACED` node (a direct consequent from the deletion of the node the transaction resides at and the insertion of the moved node with the prepended text-value) and a `DELETED` node (the node which has been moved).

3.7 Runtime/Space analysis and scalability of the ID-based diffing algorithm

The runtime of the ID-based diff algorithm is $O(n+m)$ whereas n is the number of nodes in the first tree and m the number of nodes in the second tree. It is thus a fast linear diff-computation in comparison to the FMSE-algorithm ($O(n*e+e^2)$, n is the number of unchanged nodes and e the number of changed nodes) which is used in the first place to determine the differences which are imported. The space-consumption is $O(1)$ and in case the replace-operation is enabled at worst $O(k)$. k is the sum of the subtrees which are cached (at most 4 subtrees currently). However the space consumption of an aggregated tree-structure which is going to be described in the next chapter (4) is $O((u+v)*k)$. u is the number of unchanged nodes, v is the number of changed nodes and k is tuple relevant stuff (type of diff, the nodes and the depths in both trees). Thus the asymptotic space consumption is linear depending only on the number of unchanged and changed nodes.

[6] which is used just like a List, to switch between a map implementation based on a persistent BerkeleyDB database depending on a specified threshold value or a simple LinkedHashMap instance

Algorithm 3: ID-based diff: postprocessing to detect moves

input : Map allDiffs, Map inserted, Map deleted
output: none (void)

```
1 // For every diff tuple in the map which saves all encountered diffs
    in document-order
2 for Diff diffTuple : allDiffs.values() do
3     Integer newIndex ← inserted.get(diffTuple.getOldNodeKey());
4     if newIndex != null AND (diffTuple.getDiff() == DiffType.DELETED OR
      diffTuple.getDiff() == DiffType.MOVEDFROM) then
5         allDiffs.get(newIndex).setDiff(DiffType.MOVEDTO);
6     Integer oldIndex ← deleted.get(diffCont.getNewNodeKey());
7     if oldIndex != null AND (diffTuple.getDiff() == DiffType.INSERTED OR
      diffTuple.getDiff() == DiffType.MOVEDPASTE) then
8         allDiffs.get(oldIndex).setDiff(DiffType.MOVEDFROM).setIndex(
          inserted.get(diffTuple.getNewNodeKey()));
```

	1000mods	**5000mods**	**10000mods**
min	36265.14	35717.34	33948.96
max	52459.38	61860.29	44451.06
average	39167.75	37650.67	36579.37

Table 3. Comparsion of different modification-patterns of a 111 MiB XMark instance (update/insert/delete/replace/move every 1000st, 5000st and 10000st node). Runtime in ms.

Our performance evaluation involves measuring the scaling during random modification-patterns (Fig. 6) and different document sizes with scaled modification-patterns (Fig. 8). The modification-patterns are increased in the same scale as the document size such that the documents are modified with approximately the same number of modifications. The hardware used is a common notebook with 4Gb RAM and a Core 2 Duo 2,66 Ghz CPU. All performance-measures are executed 20 times. Fig. 6 shows that the number of modifications minimally affects the runtime. The runtime decreases linear as less modifications between two revisions are encountered. However the linear decrease is minimal due to a few more value and node-ID comparisons.

Fig. 8 (Table 4) depicts the scaling during different document sizes and modification schemes. The scale is logarithmic, thus we are able to identify the linear runtime due to increased document sizes. It therefore emphasizes the asymptotic bound.

Table 5 shows the comparison based on the same documents as in Table 4 however this time utilizing the hashes to skip the traversal of unchanged subtrees (nodes with identical hash-values and node-IDs). The runtime approximately halves in this example. It is obvious that the gain in speed depends

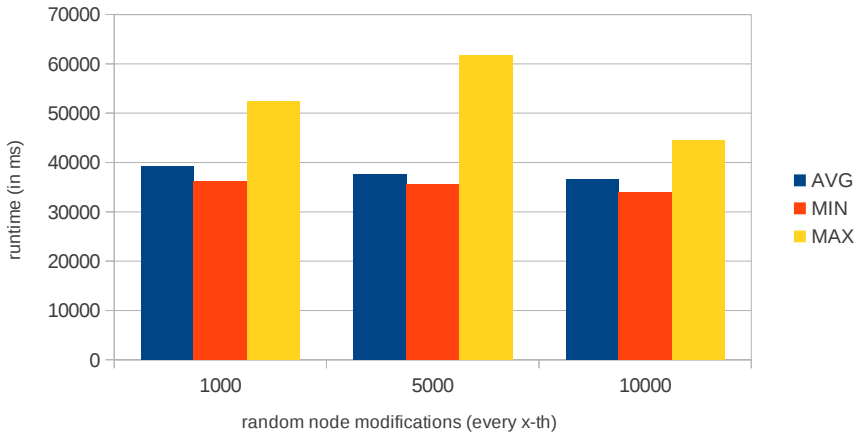


Fig. 6. Scaling during different modification-patterns (update/insert/delete/replace/move every 1000st, 5000st and 10000st node) in a 111 MiB XMark instance.

on the subtree-sizes to skip. The worst case are modifications at leaf nodes in large, deep subtrees. In this case the hash-based optimization does not gain any speedup (in the 11 MiB XMark document with all leaf nodes updated approximately 4062.52ms in the average case compared to 4064.96ms for the hash-based optimization after 20 runs). In the best case modifications occur near the root node of large, deep subtrees (in the 11 MiB XMark document with depth 2 updated the runtime drops from 31979.92ms to 12280.01ms in the averge case). However, the XMark synthetic documents are not very deep, such that usually only a few nodes are skipped for every identical hash-value.

The high max-runtime values are likely to be caused by the JVM warmup during the first or first few runs. This is also confirmed by invoking our Bench-

	11 MiB	**111 MiB**	**1111 MiB**
min	2957.79	33948.96	401878.64
max	5694.65	44451.06	439337.71
average	3323.96	36579.37	413158.13

Table 4. Comparsion of different XMark instances (11 MiB, 111 MiB, 1111 MiB modifying every 1000st, 11000st and 122221st node). Runtime in ms.

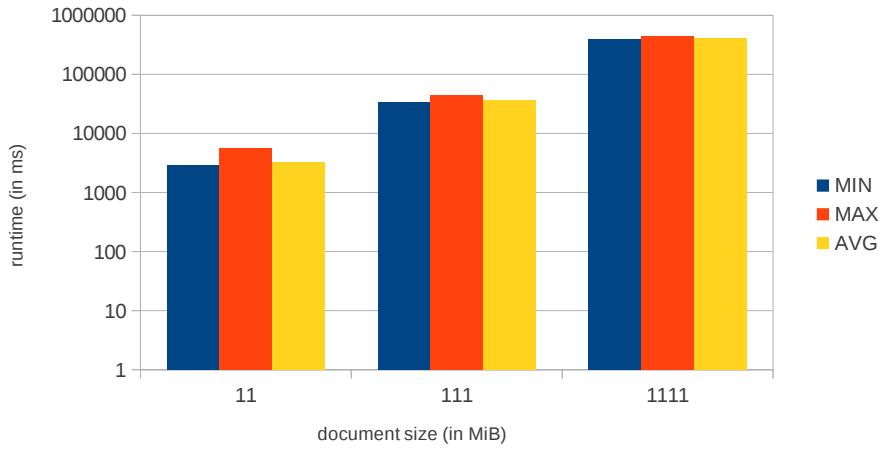


Fig. 7. Different document sizes with modification-count scaled accordingly (11 MiB ⇔ modifiy every 1000th node, 111 MiB ⇔ modify every 11000 th node, 1111 MiB ⇔ modify every 122221th node / Y-axis logarithmic scaled)

	11 MiB	**111 MiB**	**1111 MiB**
min	1239.01	14208.30	197223.12
max	3698.94	19135.01	271794.82
average	1408.74	14596.86	210296.70

Table 5. Comparsion of different XMark instances skipping subtrees of nodes with identical hash-values (11 MiB, 111 MiB, 1111 MiB modifying every 1000st, 11000st and 111000st node). Runtime in ms.

marking Framework Perfidix with repeated invocations of only one run whereas the first invocation always yields the highest runtime by magnitudes. We furthermore encountered a very small linear decrease during the first few runs until a threshold value is reached likely due to filesystem caches.

3.8 Conclusion and Summary

We motivated the import of differences in Treetank between full dumps of temporal tree-structures or different similar trees to subsequently take full advantage of Treetanks' (1) revisioning strategies, (2) unique node-IDs which identify a node through all revisions and (3) hashes of each node, which almost guarantee a unique hash guarding the whole subtree through cryptographic rolling hashes. We implemented the FMSE-algorithm to support the initial import of differences of revisioned data (currently) in the form of XML-documents whereas each document represents one revision, a snapshot at a specific time. Considering no

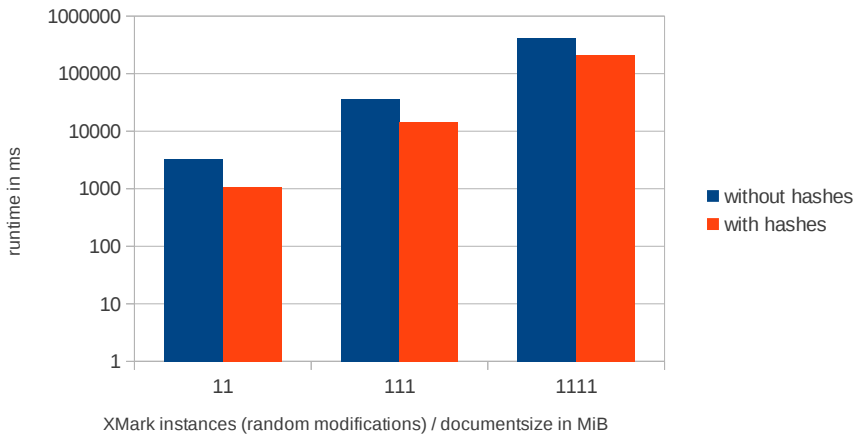


Fig. 8. Different document sizes with modification-count scaled accordingly (11 MiB ⇔ modifiy every 1000th node, 111 MiB ⇔ modify every 11000 th node, 1111 MiB ⇔ modify every 122221th node / Y-axis logarithmic scaled)

such data is available other even more sophisticated preprocessing steps have to be implemented before the FMSE-algorithm on the revisioned data in Treetank is useable. Two of the use cases which are going to be discussed in Chapter 5 require further preprocessing. Note that the FMSE-algorithm does not require unique node-IDs and matches nodes based on a longest common subsequence (LCSS)-computation (which is ambiguous) on leaf- and inner-nodes in a bottom up traversal. Thus it might mismatchs similar nodes eventuating in too many edit-operations and thus only guarantees a minimum edit-script if the leaf node comparison-function is able to match exactly one corresponding node each time. The algorithm is particularly useful to compare similar different tree-structures which do not temporally evolve and usually naturally do not include node-IDs.

Furthermore we implemented many edit-operations which were not available in Treetank to support the implementation of the FMSE-algorithm and an expressive agglomerated tree-structure. We have shown that a subsequent diff-calculation based on IDs and hashes which guard the whole subtree usually is faster than the same algorithm without utilizing hashes. However it depends on the properties of the tree-structures and the modification patterns. Whenever large subtrees are unchanged, thus the subtree-roots have identical hash-values the runtime is reduced drastically.

Besides using hashes optionally to skip whole unchanged subtrees our ID-based diff-algorithm combines `INSERT/DELETE` and `INSERT/INSERT/DELETE/DELETE` sequences of whole subtrees to a single replace-operation. However it is only a simple heuristic.

Moves are optionally detected in a postprocessing-step by searching for `INSERT/DELETE` and `DELETE/INSERT` combinations with the same node-IDs. Move detection is especially useful to support analysts with an expressive visualization. Otherwise, especially in document-centric XML, for instance DocBook[28] documents, it might be impossible to draw conclusions from simple inserts/deletes, whereas an author simply moved a sentence which is obscured due to other inserts/deletions of text and/or markup.

4 Visualizations

4.1 Introduction

The last chapter introduced the first part of our Visual Analytics pipeline, the diff-algorithms in detail. Usually however sophisticated preprocessing-methods are needed, which are explained for a few applications in Chapter 5.

This chapter describes several visualizations which help analysts to gain knowledge. First, an aggregation of the two tree-structures to compare is described. Then the visualizations are detailed. Our visualizations rely on the diff-algorithms described in Chapter 3 and therefore depict the tree-edit distance, that is structural (insert/delete/move/replace) and non-structural (update) operations which transform one tree-structure into the other one. Different similarity measures are used to indicate the similarity of leaf-nodes and internal nodes either based on comparing String-values via the Levenshtein-distance or in the latter case based on overlapping subtree-structures. However the usage of similarity measures is highly modular and thus other measures might be included in the future which either can be switched by user interaction or through heuristics. After briefly describing the *TreeView*, the *TextView* and the *SunburstView* as well as basics of our specialized Sunburst layout, an explanation of filtering technique follows which together with the ID-based diffing-algorithm [7] facilitates the analysis of large tree-structures ranging from about 100MB to even GBs of data. The key assumption underlying this efficient diff-algorithm/visualization is that similar trees are compared and therefore only a small fraction of a tree-structure has to be transformed to derive the other tree-sturcture. Querying capabilities, similarity measures and the visualization of moves are described subsequently. Next, small multiple display variants are detailed which facilitate the comparison of several tree-structures. An asymptotic runtime- and space-analysis as well as a short performance study follow. The chapter concludes with a summary.

4.2 Aggregation

An aggretion of two tree-structures is illustrated in Fig. 9. The top half depicts the two tree-structures (revision 1 and revision 2) to compare whereas the bottom displays the aggregation or fusion of the trees based on diff-tuples observed from our ID-based diffing-algorithm. The two trees are input to the ID-based diff-algorithm which in turn fires diff-tuples. These tuples form the basis of the agglomerated tree-structure. A straight forward approach which we followed is to store the tuples in a simple List datastructure [8]. The colors of the nodes in the agglomeration denote if and what change is made. Deletions for instance are marked in red, whereas insertions are colored blue. Updates are not only supported for leaf nodes, as in the ContrastTreemap approach described in Chapter 2 but also for internal nodes. Furthermore the replace-operation as well as moves are supported.

[7] usually the hash-based version comparing the hash-values of the nodes first

[8] in our case a Map which is used like a List to exchange a Java core collection map implementation with a persistent BerkeleyDB map implementation

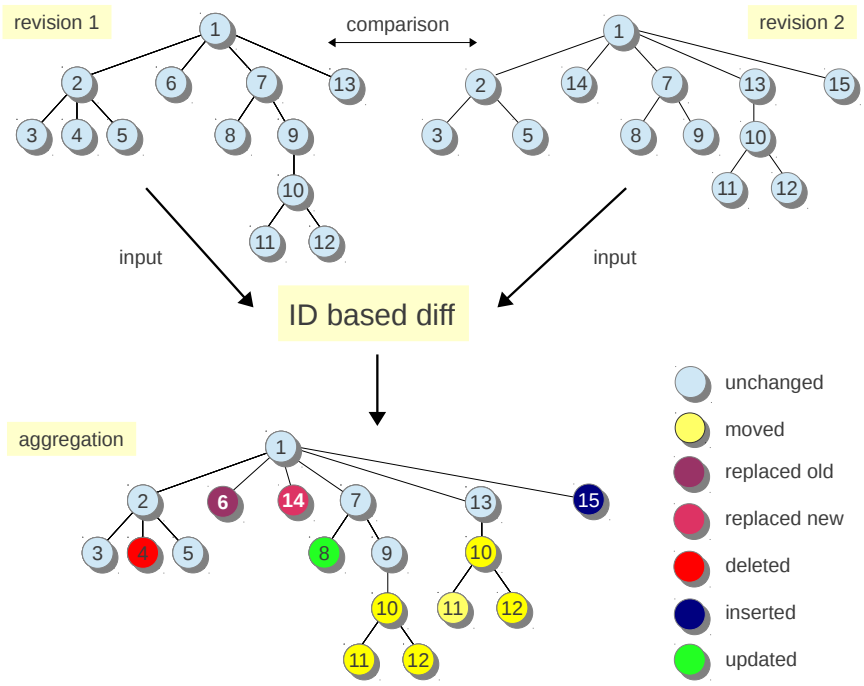


Fig. 9. Two tree-structures aggregated. The numbers denote unique node-IDs. Both revisions are input to the ID-based diff-algorithm. The output represents diff-tuples including the node-IDs from both nodes which are compared in each step, the type of diff and the depths of both nodes. Storing the observed diff-tuples in an ordered data-structure forms a simple tree-aggregation.

4.3 Visualizations

As described in the motivation humans are best in interpreting visual content. Therefore visualizations are developed which facilitate humans in gaining new insights and quickly detecting differences in tree-structures. Next, all available views are described in detail.

- First, a *TreeView* displays nodes in a tree structure just like visual frontends for filesystems as illustrated on the left side in Fig. 10. Nodes either can be expanded to show all child nodes which are inside the current viewport or collapsed to hide children. The subtree of a selected node is marked with a background color. Currently the view is not able to incorporate the aggregation and thus is not further described.
- The *TextView* displays serialized XML-documents or fragments and supports syntax highlighting. Moreover it just serializes the part of data which

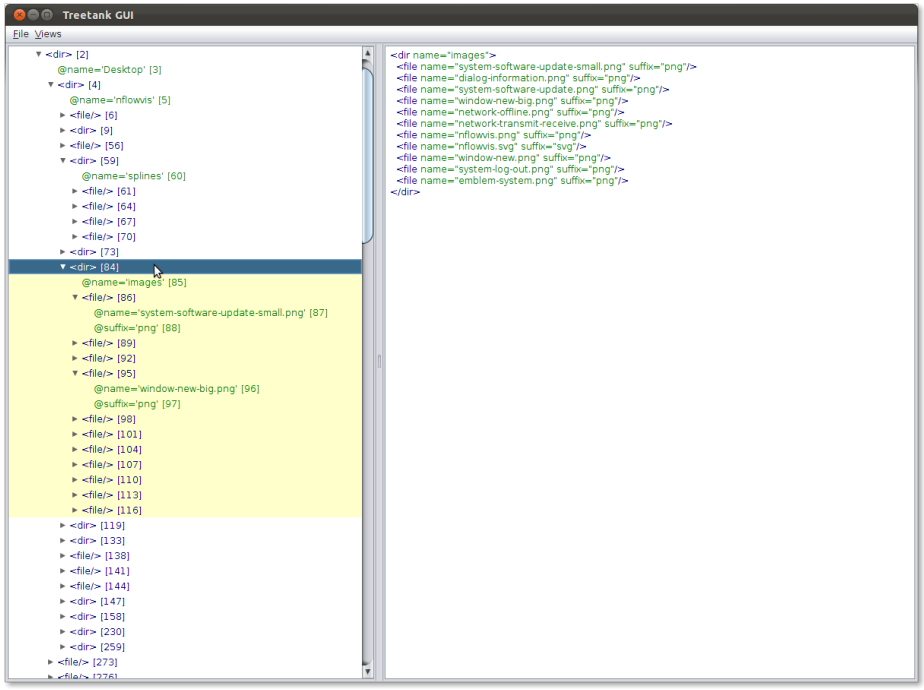


Fig. 10. TreeView and TextView side-by-side

is currently viewable and an additional small overhead of pixels to enable scrolling. Other data is serialized and appended while scrolling down. A reusable pull-based StAX-parser supports the syntax highlighting, serialization of end-tags as well as the append-approach.

Fig. 10 displays the *TreeView* and the *TextView* side by side. Note that the two views are kept in synchronization.

In order to support an analyst with the task of analysing differences in tree-structures the view supports another mode which incorporates the aggregation of the two tree-structures to compare described in section 4.2. Based on this aggregation a second pull-based parser is developed. The depths of the diff-tuples in the aggregated structure are used to determine when to emit end-tags. A dedicated background-color is used to mark the type of diff. In case an updated node is encountered the old value and the new value is emitted. The view in contrast to the *SunburstView* which is described next is not able to visualize connections between the old and new location of moved nodes.

A legend which describes the color $\leftrightarrow$ change mapping is currently only available from within the *SunburstView*. However this is only an implementation detail and a help-dialog will be added in future releases. Exemplary a side-by-side view with the *SunburstView* is depicted in Fig. 11 whereas the

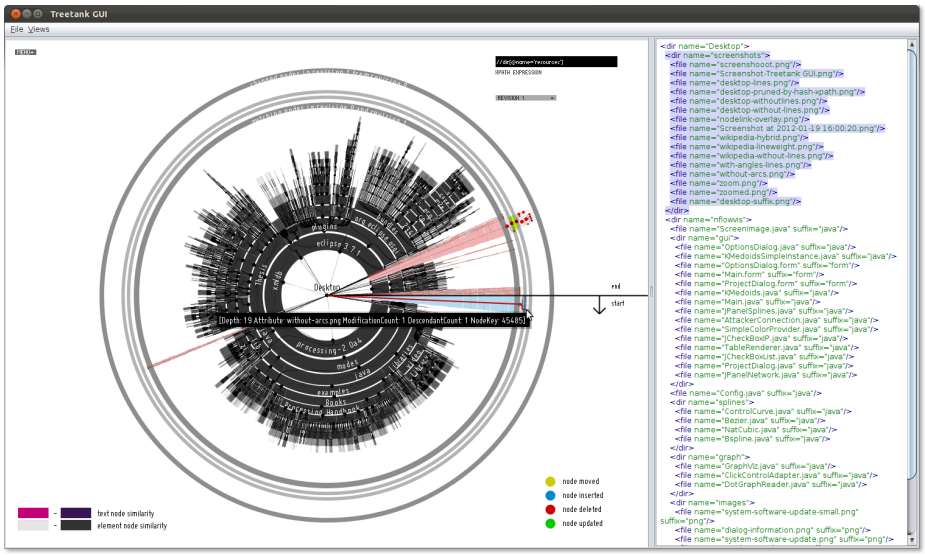


Fig. 11. SunburstView and TextView side-by-side.

first inserted subtree is also visible in the *TextView* area, marked by a blue background-color.

While the *SunburstView* as we will shortly see provides a great overview about the whole tree-structure and subtrees, the *TextView* provides a better detailed view on selected subtrees. Other deficiencies meantioned in the introduction regarding the boundary of nodes and XML-specific details do not apply as we compare the tree-structure with the ID-based diffing-algorithm in the first place instead of comparing single characters line by line. Besides the lack of an appropriate overview, which is one of the advantages of the *SunburstView*, the *TextView* is an ideal partner to the *SunburstView* as the XML text-serialization is better readable than radial Sunburst labels.

The diff-algorithm is only ever called once for every visible view[9]. The diff-tuples are then broadcasted to all other views which are capable of displaying the aggregated tree-structure.

- The *SunburstView* displays a tree structure in a radial layout (Fig. 12). We first describe the basics to display one tree-structure and extend the approach to include the aggregated tree-structure to display the differences between two tree-structures. The Sunburst-layout is a space filling approach, thus aiming for a maximized usage of available screen space for the hierarchical visualization. Furthermore it is an adjacency based approach, drawing child nodes next to their parent node. In contrast, Treemaps enclose child-nodes within parent nodes. Thus a Treemap utilizes available screen space to the

[9] the only exception are small multiple displays which represents changes among several tree-structures

full extend as the root node occupies the whole available screen space recursively embedding descendants as rectangles. In contrast, in the Sunburst method corners are left empty due to the radial representation. While this alone on first glance might be a great drawback in addition to circular segments which are more difficult to read regarding node labels and comparisons of item-sizes, the layout is stable even if a lot of changes have to be displayed and the hierarchical structure is much better readable. The Spiral Treemap layout which has been described in Chapter 2 is relatively stable but it is still very difficult to track changes which might be scattered through 90 degree changes in direction as well as the complicacy to follow nested rectangles which are arranged in spirals in comparison to the simplicity of a Sunburst layout.
The root node of a tree-structure in a Sunburst-layout is plotted in the middle of the screen depicted as a circle. Child-nodes of the root node are drawn in circular segments next to their parent. The radius depends on the depth. It shrinks with increasing depth such that the area of circular-segments between two levels does not change. Otherwise items toward the edges occupy more space. However this behaviour can be changed to further visually emphasize changes which will be visualized along the edges (section 4.4). The arc of an item, which depicts one node, in the Sunburst layout depends on one or more node-attributes. A relative measure in regard to the other children is used.
In our case the subtree-sizes are mapped onto the extend of each item such that nodes having more descendants occupy more space. The formular is straight forward:

$$extend = \begin{cases} 2 \cdot \pi & \text{if } node\ is\ root\ node \\ parentExtend \cdot descs/(parentDescs - 1) & \text{otherwise} \end{cases} \tag{3}$$

Note that we recently added the number of descendants of each structural node in Treetank to maximally speed up the creation of the visualization.
The color of each item in case of internal nodes (element nodes) is mapped to the subtree-size of a node. `TextNode`s are colored according to their text-length.
A node-link diagram is drawn on top of the *SunburstView* to further emphasize the hierarchical structure. Dots representing the node in addition to the SunburstItem-segment are depicted in the center of the item whereas either bezier curves or straight lines denote a child/parent-relationship between the nodes.
In order to support large tree-structures the generated Sunburst-items are drawn into an offscreen buffer, whereas the items are only used to implement a mouseover effect and to support XPath-queries with subsequent highlighting of the resulting nodes/items.

Interaction The view is highly customizable. Checkboxes enable or disable plotting the node-link overlay and/or the Sunburst-arcs to either em-

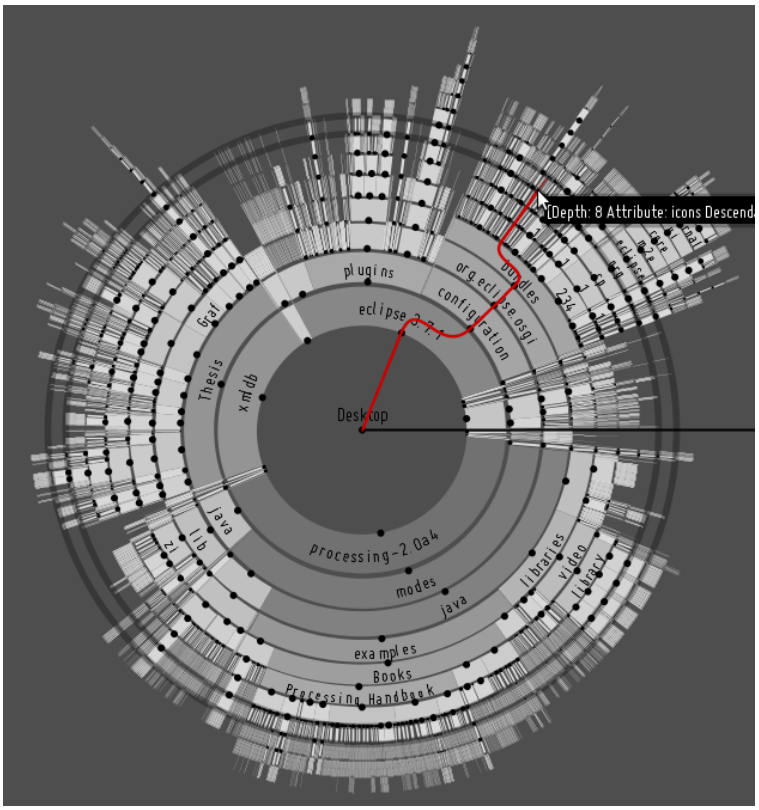


Fig. 12. SunburstView depicting the structure of the authors filesystem directory "/home/johannes/Desktop".

phasize the parent/child relationship or subtree-sizes[10]. Furthermore the line/curve-thickness denoting parent/child relationships and the dot-sizes are adjustable.

To support different mappings from node-attributes to the color of Sunburst-items, a term which is used interchangeably in the following sections, a linear-, squareroot- and logarithmic-normalization is available.

Crucial to the interaction and the value of the visualization itself is the possibility to drill down into the tree. Clicking an item results in drawing the selected node with its subtree in a new Sunburst-diagram whereas the selected node becomes the new root. Furthermore a simple, fast undo-operation is supported as we keep track of offscreen-buffers and the items.

A well known technique to enlarge small regions is to use transformations of the screen-space, as for instance a fisheye lense to select very tiny items. The enlargement of small items via a fisheye lense is depicted in Fig. 13(a). Zoom-

[10] subtree-similarities in the comparison mode

ing and panning is also incorporated allowing affine transformations of the screen (Fig. 13(b)) to analyse important regions. Note that the mouseover-effect displaying additional information about the node itself as well as the legends are not affected by the transformation. As the background-buffer cannot be used in this case zooming is restricted to smaller trees with an upper bound of about 10_000 to 15_000 nodes.
In order to manipulate Treetank resources it is even possible to insert XML fragments as right-siblings or first-childs as well as to delete nodes.

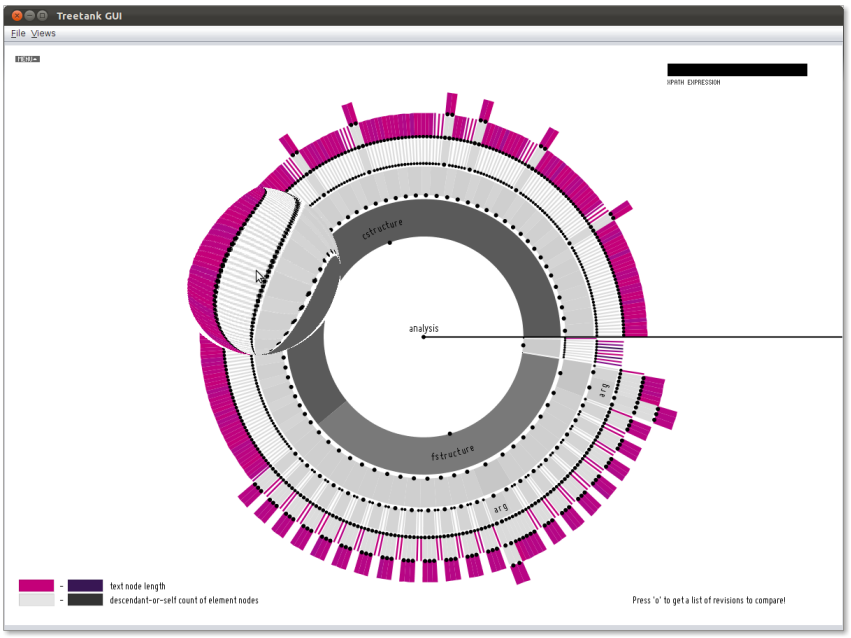
Querying XPath is usable to query the tree-structure for specific nodes. Result sequences are highlighted in a light green. Fig. 14 displays the result of a simple `//*[text()='var:0']` query to highlight all nodes which have a text-node child with the value "var:0".

Labels Whenever the Sunburst items sufficiently large and an adaptable scale to draw the arcs for each depth is greater than a predefined value, labels are drawn. Labels in the top half of the visualization, that is if the center of the item is greater than π, are drawn beginning at the start-angle in clockwise direction, otherwise they are drawn starting from the end-angle in counter clockwise direction. Furthermore the font-size is limited to a range between two values, whereas the size is decreased with an increased depth.

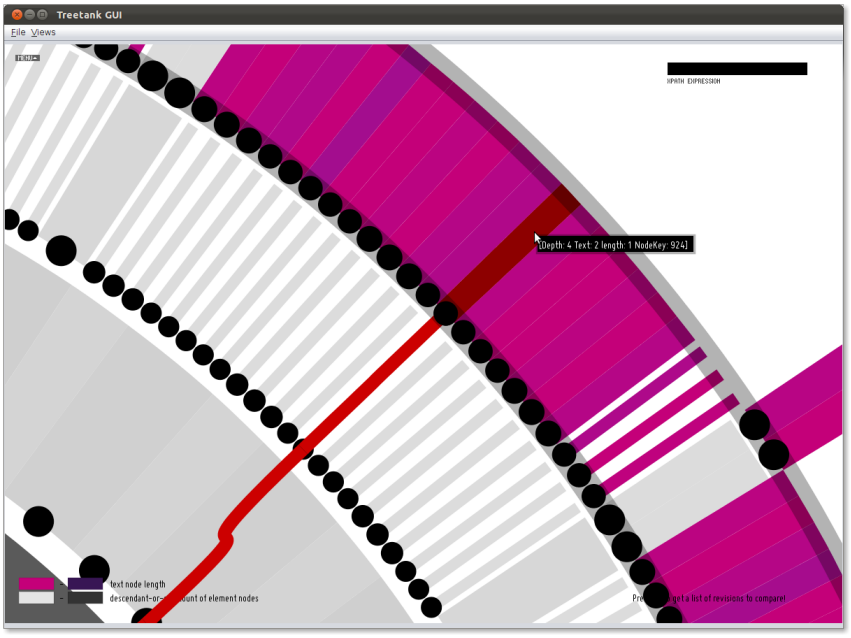
Filtering/Pruning The standard *SunburstView* optionally filters the tree by level. While this filtering is not perfect in circumstances where the fanout is very large, it works very well to keep the number of generated Sunburst items small. Furthermore the view currently is used as an entry point to the comparsion view which is also based on the Sunburst-layout whereas it is planned to backport the filtering by itemsize.
Whereas it is sufficient to use an XPath-query as for instance `//*[count(ancestor::*)<3]` to get a sequence including all nodes between level 0 and 3 we opted for a tree-traversal implementation, as the XPath query has to touch all ancestor nodes in the current Treetank implementation due to our encoding which does not utilize hierarchical node-IDs as for instance the ORDPATH/Dewey-IDs where it is usually trivial to compute such queries on the ID itself in an in-memory B*-tree or another datastructure.

4.4 Comparsion using a new Sunburst-layout algorithm

The standard *SunburstView* includes a comparison-mode. Once a base revision is opened and the *SunburtView* is enabled an analyst is able to choose another revision from a dropdown menu for comparison. Note that all interaction capabilities described earlier are also available in the comparison mode. Differences and additional capabilities are described in the following sections. In order to



(a) Fisheye transformation



(b) Zooming into the visualization

Fig. 13. Techniques to enlarge regions of interest.

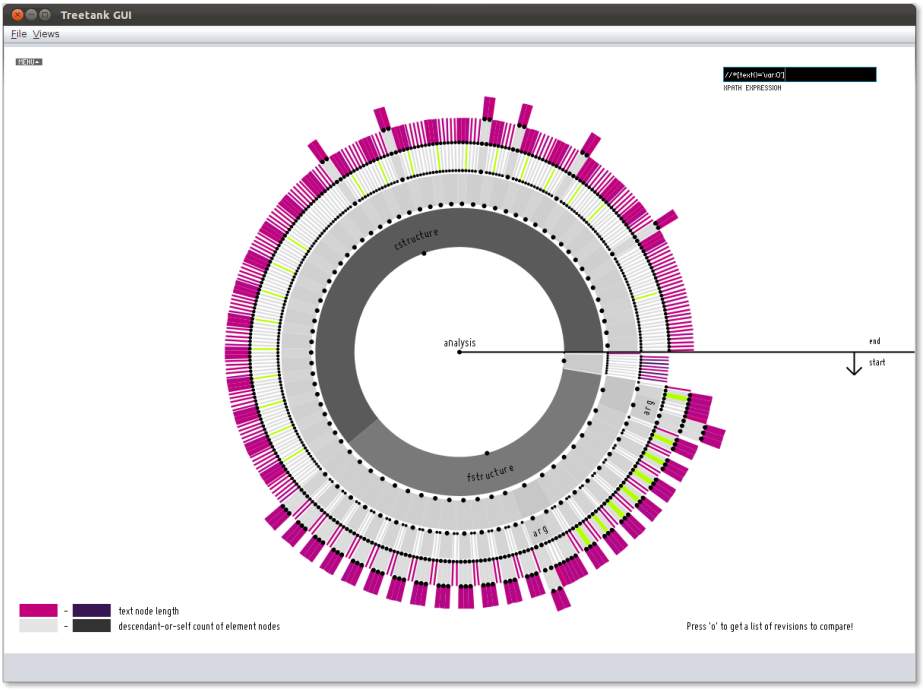


Fig. 14. XPath query results displayed in light green

compare tree-structures in a radial arrangement similar to the described SunburstView which facilitates exploring a single revision a new layout-algorithm is developed. Next, we first describe the new layout.

Sunburst Comparsion-Layout The Sunburst comparison layout is illustrated in Fig. 15. Nodes are colored as depicted in the color legend in the bottom right corner. All nodes which are unchanged during the comparison of revision 0 and 1 are plotted inside the inner circle which is labeled "matching nodes in revision 0 and revision 1". The circle itself is drawn between the maximum level of the unchanged nodes plus one and maximum level plus two. Changed nodes are zoomed out from their original place and drawn between the two dark circles labeled "changed nodes in revision 0 and revision 1" and "matching nodes between revision 0 and 1". Fig. 15 depicts the area with hatches. Similarly the arrows emphasize the direction in which changed subtrees are zoomed/dislocated. Both, the hatches and the arrows are only drawn to stress our design decisions and the semantic zoom which serves a double purpose. First, the visualization adheres to a Tree-Ring metapher depicting the evolution of a tree. Just like the age of a tree in the nature is deducable by analysing rings in a cross-cut of the stem whereas the rings denote the age and each ring represents one year starting from the center to the outside, our prototype aims at representing the changes

between two rings. In our visulization the unchanged nodes form the center of the *SunburstView* whereby changed nodes are zoomed to the border between the inner and outer ring which is representing the growth of a tree in case of analysing temporal tree-structures. Additionally, this transformation displaces changed subtrees to a prominent place. Thus, small changed subtrees or even single node-changes are much better noticeable as they are not surrounded by unchanged subtrees which might even be deeper. To depict changes between multiple trees or several revisions of a tree first considerations involved the addition of changes from a sequence of sorted revisions in further rings. The center thus represents unchanged nodes between *all* compared revisions whereas changes between selected or consecutive revisions are drawn between new appended rings. According to the Tree-Ring metapher each comparison between a pair of trees appends a new ring denoting the changes. Therefore two rings denote the boundaries between the changes of comparing two revisions. However, this affects the whole layout each time. The idea proved to be not viable because of complexity issues. To name a few

- The center must keep space between unchanged nodes/subtrees for all upcoming changes (between all compared trees/revisions).
- Consecutive calling the diff-algorithm and merging diffs into a single aggregated tree-structure.
- Keeping track of all opened transactions on each revision and resetting the transaction appropriately depending on the revision in which a change has occured to derive node-labels.
- Similarly the depth for items in the tree will change very often which involves further state and it is almost not possible to denote the current depth during a preorder traversal of the aggregated tree-structure (depending on the tree-structure itself).
- The `subtree-size` and `modification`-count of each nodes' subtree will be cumbersome to calculate.

These and similar complexity considerations formed the idea of introducing small multiple displays instead, which are described in section 4.7.

Short animation In order to clearly demonstrate the semantic zoom which dislocates items a short animation is implemented as test persons usually do not grasp the meaning without further explanation. Thus the transformation of changed subtrees is shown which dislocates the items to their dedicated positions along the arrows in Fig. 15 [11]. However the animation depends on the number of Sunburst items which are created and thus is skipped depending on a threshold to avoid a reduction of the framerate (frames per second). Similarly zooming/panning is not allowed if too many items are generated.

[11] remember, the arrows are not drawn in the visualization, they are just added to the screenshot to emphasize the transformation

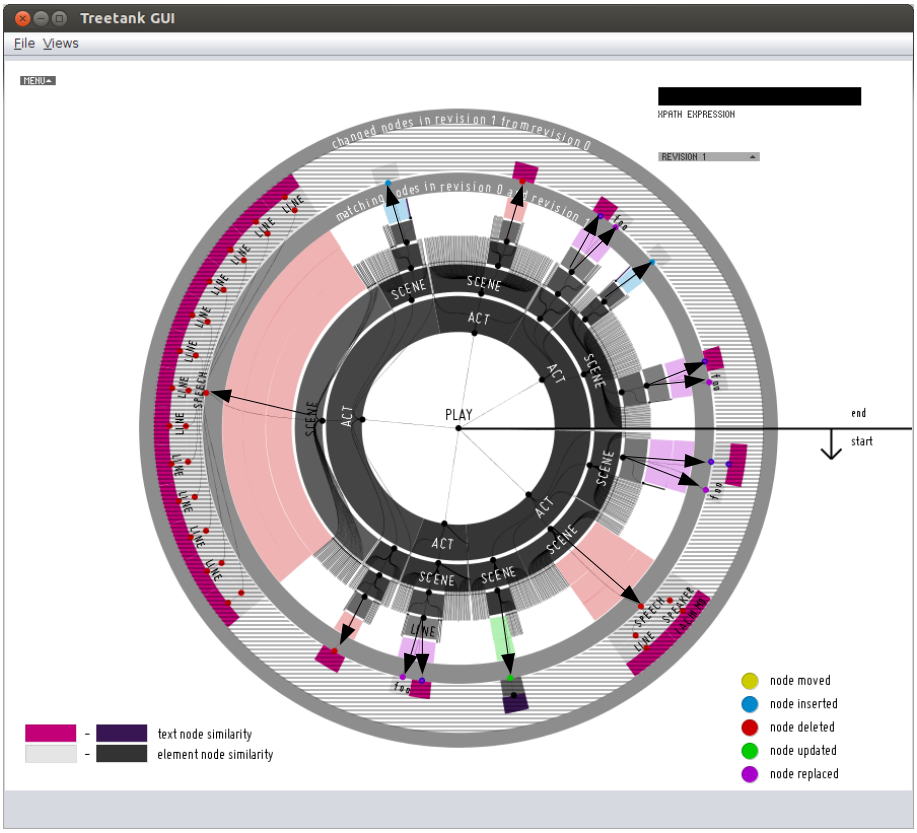


Fig. 15. SunburstView - comparison mode.

Layout algorithm After invoking the ID-based diffing-algorithm and collecting observed changes, the new Sunburst-layout is drawn. First of all, based on the aggregation of the compared trees, Sunburst items are created. The diff-tuple which denotes a node in the aggregation is of the following form: key in first revision / key in second revision / depth in first revision / depth in second revision / type of diff. The Sunburst items are created during a preorder traversal of the diff-tuples. Initially we developed an axis based on the key idea to traverse the tree-structure of the newer revision and change a transaction cursor to the old revision whenever a deleted, moved (the old place) or replaced (the old) node is encountered. The advantage is that node-pointers are usable as a guidance to traverse the aggregation in preorder. Note, that the tree-aggregation in contrast is based on the diff-tuples and thus does not include direct pointers to follow. Therefore we first opted for the pointer-based to follow node-pointers with a read-transaction. The current-transaction is temporally replaced by a transaction opened on the old or back to the one opened on the new revision depending on

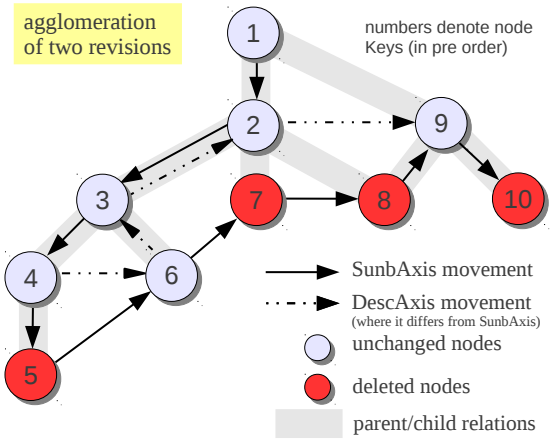


Fig. 16. SunburstCompare-Axis based on node-pointers.

the current type of diff. However the pointer-based preorder-traversal beares a lot of complexity as the node-ID of the next node to traverse has to be set in advance and the movement of the transaction in a lot of cases cannot be immediately reflected by adjusting datastructures which are required to determine the start-angle, end-angle, subtree-size and other attributes of a Sunburst item. Fig. 16 demonstrates a lot of this complexity on a simple tree-aggregation. Note that the terms "aggregation" and "agglomeration" are used interchangeably in this thesis.

If the transaction is located at the node with the node-ID/nodeKey 4 the next node returned following pointers[12] is the node denoted by node-ID 6 because node 5 is deleted and thus not referenced and not available in the newer revision. As such it is not sufficient to follow only child-node pointers. Thus additionally the depth of the next diff-tuple must be compared to the current depth. Furthermore the kind of movement must be tracked to adjust datastructures used to determine all Sunburst item attributes required (start-angle, end-angle, number of subtree modifications...).

Another example of the complexity is the movement from node 6 to node 9. Usually the next node will be node 9, however in our case the next node must be the deleted node 7. Thus internal stacks keeping track of attributes which are required for new sunburst items in many cases must be adapted differently than during a simple preorder traversal of a single revision.

The last pitfall in Fig. 16 occurs after traversing the node denoted by node-ID 9. Usually the traversal is finished but in this case the deleted node "10" follows.

[12] for instance in the DescendantAxis

Algorithm 4: Diff-Axis hasNext()-skeleton

input : (instance variables) INodeReadTrx mOldRtx, INodeReadTrx mNewRtx, boolean mHasNext, int mDepth, int mNextDepth
output: true, if more diffs are in the diff list, false if index == size

```
1  // Fail if there is no node anymore.
2  if !mHasNext then
3      return false;
4  // Setup everything.
5  setupDiffs();
6  if mHasMoreDiffKinds == true then
7      if mDiff == DiffKind.UPDATED then
8          mOldRtx.moveTo(mDiffCont.getOldNodeKey());
9      // Always follow first child if there is one.
10     if mNextDepth > mDepth then
11         return processFirstChild();
12     // Then follow right sibling if there is one.
13     if mDepth == mNextDepth then
14         return processRightSibling();
15     // Then follow next following node.
16     if mNextDepth < mDepth then
17         return processNextFollowing();
18 // Then end.
19 processLastItem();
20 mHasNext ← false;
21 return true;
```

We observe that following node-pointers to traverse the aggregation is very complex in terms of a lot of special cases have to be handled and thus is a performance issue as well due to a lot more comparsions and instructions.

The complete algorithm therefore is omitted as we developed a second, in comparsion lightweight algorithm. Instead of using a pointer based traversal it became apparent that it is easier to directly use the diff-tuple and move either the transaction opened on the old revision or the transaction on the new revision depending on the diff-type.

The outline of *hasNext*(), a method which returns `true` if the preorder traversal is not finished or `false` otherwise, is described in algorithm 4.

The method `setupDiffs` takes care of setting the current depth which depends on the diff-type, the upcoming next depth, if the list has more diff-elements as well as setting the current working-transaction. Either the one opened on the old revision or the one on the new revision is used depending on the current diff-type. The following three `if`-clauses ensure the preorder traversal. Instead of following pointers the depth of the next element in the aggregation must be compared with the current depth. This is a direct consequence of not generat-

ing an intermediate more sophisticated tree-structure of the agglomeration. The type of movement (firstChild, rightSibling or followingNode) is used to determine how to adjust datastructures which are required to create a Sunburst item such as the start-angle, end-angle and parent-index. Details are omitted for brevity.

The following section describes implementation details of the semantic zoom.

Semantic zoom implementation The implementation of the *semantic zoom* with changes highlighted in a dedicated place requires adapting the depths to dislocate changed-nodes to their dedicated position between the two rings. Three cases have to be distinguished in which the depth must be adapted (Fig 17).

1. Transition from an unchanged node to the first changed node in a subtree (1. in Fig. 17).
2. Transition from a changed node back to an unchanged node in case the unchanged node is not in the changed nodes' subtree. An unchanged node might only be in a changed nodes' subtree if the changed node has been updated (2. in Fig. 17).
3. Transition to another changed node once a (changed) subtree has been traversed and a node on the XPath `following::`-axis follows whereas its original depth would be less than the depth of the inner ring ($pMaxDepth + 2$) which only ever includes unchanged nodes (3. in Fig. 17).

Algorithm 5 determines if and how the depth must be adjusted. The first and second case is handled by the first `if`-clause, whereas the transition back to the original depth is handled by the `elseif`-clause. An instance variable *mTempKey* is used to determine when to switch back. It is the node-ID of the first node in the XPath `following::`-axis. *mTempKey* is adapted in case 3. such that it eventually contains the node-ID of an unchanged node. An `UPDATED` node most probably incorporates unchanged nodes if it is an internal `ElementNode`. In these cases the depth of the parent node plus one is used instead of the original depth such that the updated node *and* its subtree are dislocated.

Besides adjusting the depth the semantic zoom requires some form of highlighting. In our case we opted for a global "distortion" which enlarges modified subtrees and shrinks subtrees of unchanged, potentially uninteresting nodes. Thus, the arc of a Sunburst item depends on two variables, the `subtree-size` and the number of `modifications` in a nodes' subtree.

$$ext = \begin{cases} 2 \cdot \pi & \text{if } node\ is\ root\ node \\ parExt \cdot ((1-\alpha) \cdot descs/(parDescs - 1) & \\ \quad +\alpha \cdot mods/(parMods - 1)) & \text{otherwise} \end{cases} \tag{4}$$

The number of modifications in the forumla is derived from the `subtree-size` added to the number of `modifications` and multiplied by a constant. The addition of the `subtree-size` is needed to handle unchanged nodes, which do not contain any modifications in its subtree. In order to further emphasize and

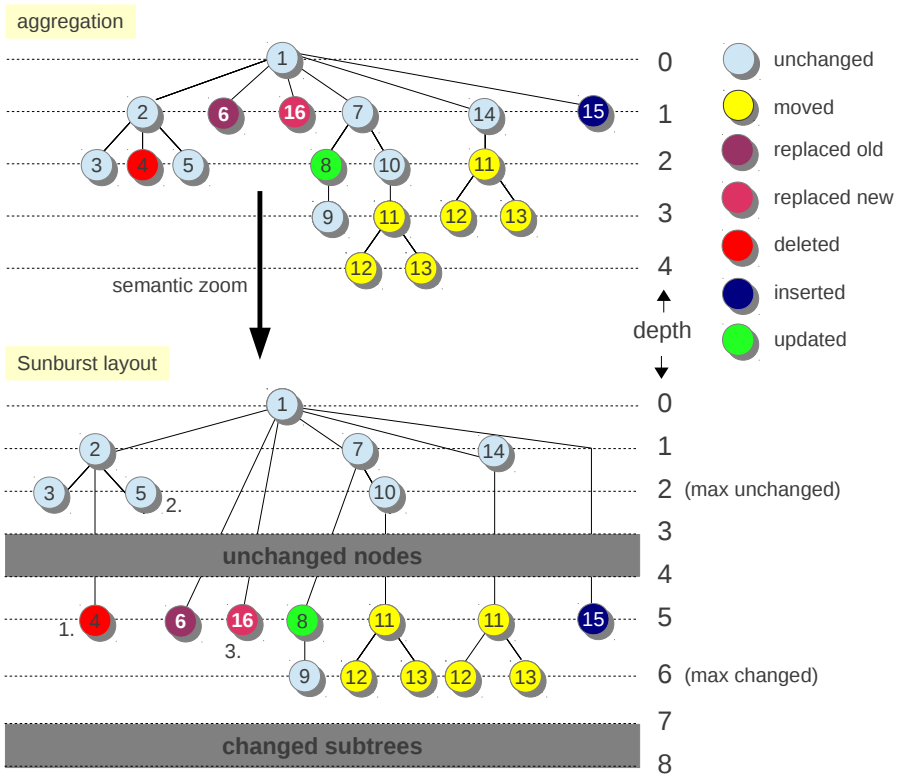


Fig. 17. Sunburst-layout depicting changes in the depth. All nodes above the grey rectangle labeled "unchanged nodes" are unchanged whereas the area between the rectangle named "changed subtrees" and "unchanged nodes" includes all changed subtrees. However it also includes changed nodes below an updated node as for instance node 9.

enlarge subtrees with a small number of modifications a constant is multiplied which has proven useful in empirical studies (Chapter 5). Furthermore note that if the parent node is modified the constant must be subtracted from the parent `modification`-count in advance.

The two variables are computed in parallel to the preorder traversal in the `Diff-Axis`. The results are appended to a thread safe queue designed for producer/consumer relationships. Modifications for the root node are gathered while observing diff-tuples, thus the number of modifcations does not need to be computed afterwards as for the other nodes in the agglomerated tree-structure. A simple heuristic determines depending on a depth-threshold if tasks are executed in the calling thread instead of another thread, as context switches for very small subtrees are too costly. Observe that the new descendant-count of each node in

Algorithm 5: Calculate depth

```
input  : int pDepth, int pMaxDepth, DiffType pDiff, long mTempKey, long
         mInitDepth, int mPrunedNodes
output: new depth
1  int depth ← pDepth;
2  if pDiff != DiffType.SAME AND pDiff != DiffType.SAMEHASH AND pDepth
   ≤ pMaxDepth + 2 then
3      // Case 1 and 3.
4      depth ← pMaxDepth + 2;
5      int index ← mIndex + mPrunedNodes + mDescendantCount;
6      if index < mDiffs.size() then
7          DiffTuple nextDiffTuple ← mDiffs.get(index);
8          DiffType nextDiff ← nextDiffTuple.getDiff();
9          boolean nextIsOldTransaction ← isOldTransaction(nextDiff);
10         mTempKey ← 0;
11         if nextIsOldTransaction == true then
12             mTempKey ← nextDiffTuple.getOldNodeKey();
13         else
14             mTempKey ← nextDiffTuple.getNewNodeKey();
15 else if (pDiff == DiffType.SAME OR pDiff == DiffType.SAMEHASH) AND
   pDiffCont.getNewNodeKey() == mTempKey then
16     // Case 2.
17     depth ← pDiffCont.getDepth().getNewDepth() - mInitDepth;
```

Treetank can not be used, because the aggregated tree-structure is traversed which incorporates deleted, replaced and moved nodes depending on settings (move- and replace-detection). Algorithm 6 depicts how the two variables are derived by traversing the agglomeration at a specified index until the depth of a diff-tuple either is less than or equal to the start depth or no more diff-tuples are following thus forming a subtree-traversal. Note that the depth depends on the diff-type. In case of a `DELETED`, `MOVEDFROM` or `REPLACEDOLD` diff-type the depth of the node in the older revision is used, otherwise the depth of the node in the new revision. Furthermore recapitulate that the depths are computed by our ID-based diffing-algorithm instead of persistently stored as an attribute of the node by Treetank.

Filtering/Pruning Providing an initial Sunburst overview of huge tree-structures in reasonable time, ranging from a few seconds to a few minutes, requires pruning techniques to filter nodes of no or least interest. Therefore three types of filtering are provided. Changes in a nodes' subtree are always guaranteed to be visible. The following screenshots in this section are related to Fig. 18 which depicts the same tree comparison in the SunburstView without filtering nodes.

Algorithm 6: Computes the `subtree-size` of a node as well as the number of `modifications` in the nodes' subtree.

input : int pIndex, List pDiffs
output: CONSTANT_FACTOR * diffs, descendants, subtract

```
1  int index ← pIndex;
2  DiffTuple diffTuple ← pDiffs.get(index);
3  DiffType diff ← diffCont.getDiff();
4  int rootDepth ← getDept(diff);
5  int diffs ← 0;
6  if diff != DiffType.SAME AND diff != DiffType.SAMEHASH then
7      diffs ← 1;
8  int descendants ← 1;
9  boolean subtract ← false;
10 index ← index + 1;
11 if diffCounts == 1 AND index < pDiffs.size() AND hasFirstChild(pDiffs) then
12     // Current node is modified and has at least one child.
13     subtract ← true;
14 boolean done ← false;
15 while !done AND index < pDiffs.size() do
16     diffTuple ← pDiffs.get(index);
17     diff ← diffTuple.getDiff();
18     int depth ← getDept(diff);
19     if depth ≤ rootDepth then
20         done ← true;
21     else
22         descendants ← descendants + 1;
23         if currDiff != DiffKind.SAME AND currDiff != DiffKind.SAMEHASH
           then
24             diffs ← diffs + 1;
25         index ← index + 1;
```

- *by itemsize* Sunburst items which have no changes in its subtree and are too thin to be perceived individually or too thin to be selected even with the fisheye transformantion are pruned based on a predefined threshold-value. An example is depicted in Fig. 19. This type of filtering is useful wherever nodes which do not differ are of value but depicting the whole tree will not add any significant value. It considerably speeds up the generation of the Sunburst items in large tree-structures, however it does not affect the diff-calculation. Furthermore, if a new node is selected to drill down into the tree the items have to be rebuild, using the `Diff`-axis. Thus the optimization to create new items based on the initial set with adjusted angles is not usable. The subtree-size of each node as well as the number of `modifications` must be recalculated as well.

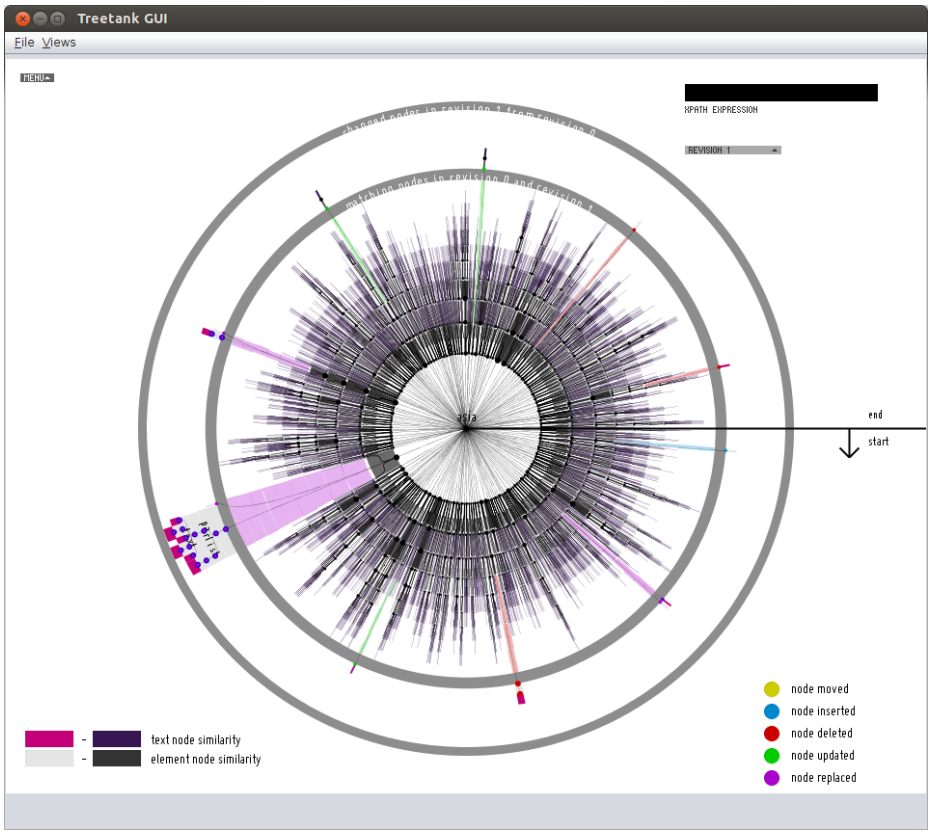


Fig. 18. Comparison without pruning.

- *by hash-based diff-algorithm* The diff-algorithm is invoked with the option to utilize persisted hashes which are created for every resource based on a database-configuration parameter. Per default a fast rolling-hash approach is used. All edit-operations thus only trigger a recomputation of the hashes of the ancestor-nodes based on the hash-value of the edited node. As described in Chapter 3 everytime the hash values of the compared nodes are identical the traversal of both subtrees is skipped. Thus, items are only created for nodes, which include changed nodes in their subtree as well as for nodes with identical hash-values (Fig. 20). This type of pruning is especially useful for large tree-structures, whereas in contrast to the pruning by itemsize it speeds up the diff-computation as well as the item creation, as in both cases subtrees of nodes with identical hash-values are skipped. However, in comparison with the itemsize-based approach sometimes more items have to be created as nodes having identical hash-values are always included.
- *by hash-based diff-algorithm without nodes which have the same hash* This type of pruning is related to the method described above. In addition to

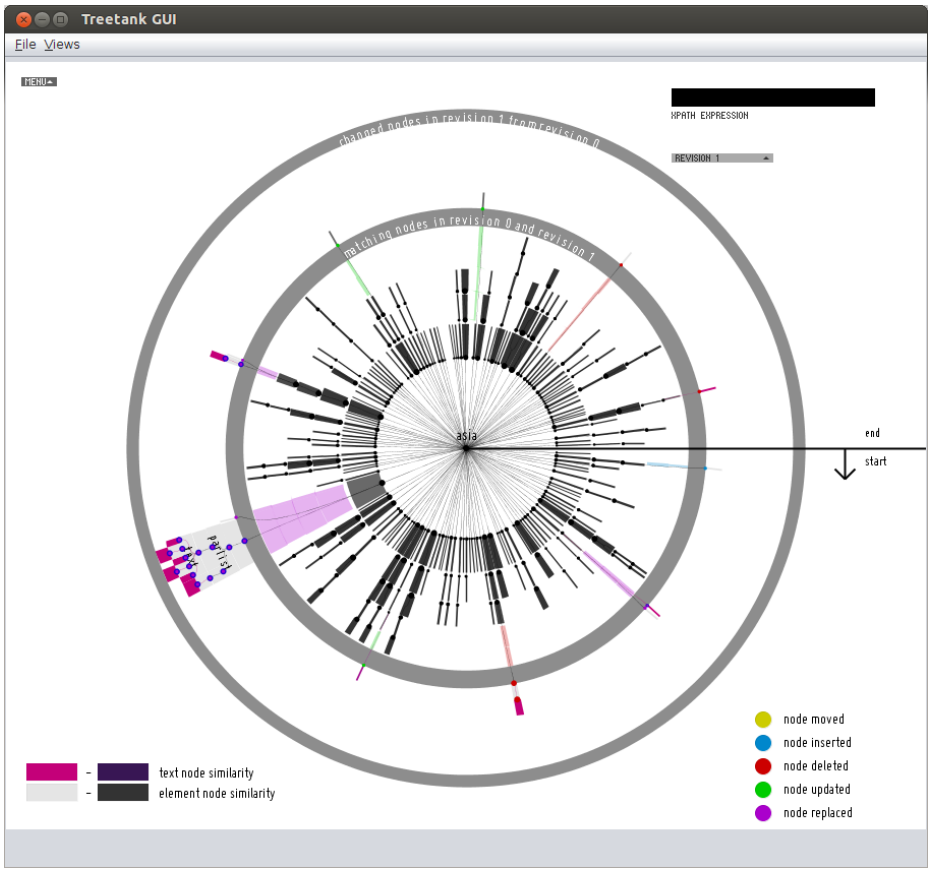


Fig. 19. Pruned by itemsize.

pruning nodes in the subtree of nodes with identical hash-values, additionally these subtree-roots are pruned, too. As a direct consequence the visualization is much better readability in case of many consecutive nodes with identical hash-values (Fig. 21).

Usually the maximum depth of unchanged nodes in the aggregated tree-structure is computed in a pipeline using a concurrent queue if more than two cores are available. Thus the computation is effectively parallelized with the diff-calculation. However, all above filtering types require postprocessing as maximum depth nodes might either be within the subtree of an unchanged node or in case of the itemsize-based filtering might be too thin regarding a threshold value und thus are pruned. Therefore, the maximum depth must be recomputed and the depth of changed nodes must be adapted accordingly. Note, that otherwise too much screen-space is wasted as the subtree-root of changed nodes is plotted

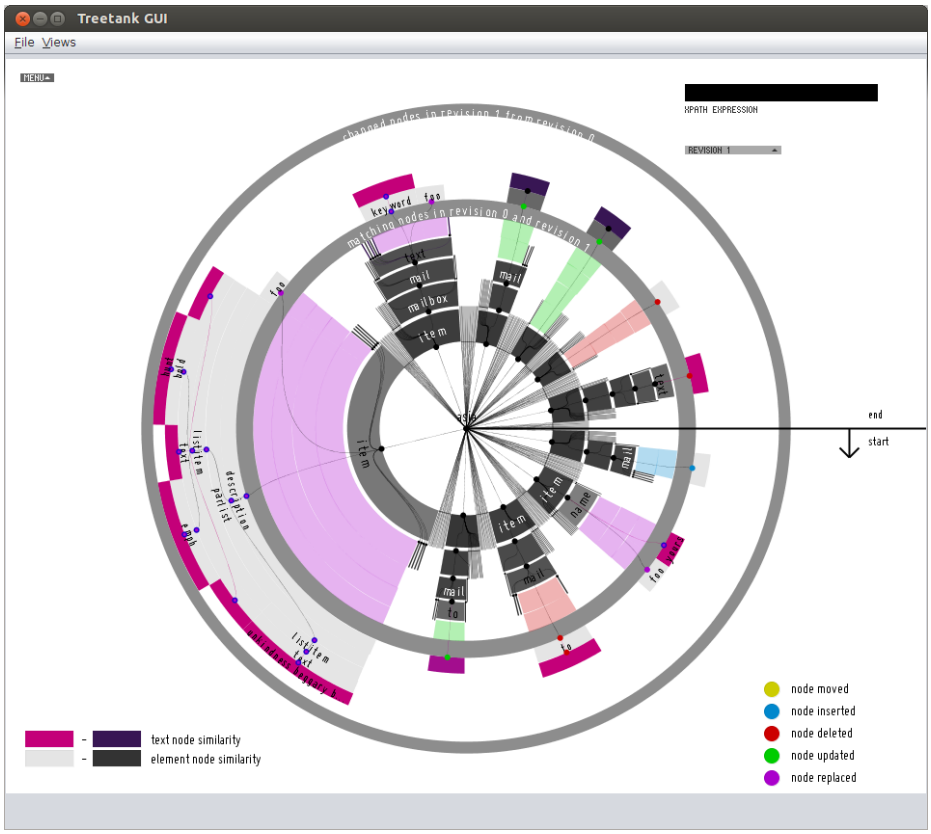


Fig. 20. Pruned by identical hash-values.

between the maximum-depth of unchanged nodes plus two and the maximum depth plus three.

Having described several pruning-techniques and briefly meantioned a general zooming/panning technique based on affine transformations which is inherited from the *SunburstView* layout depicting one revision the next section provides insight on how the selection of a new root-node to dig deeper into the tree-structure is handled. Note, that it is of the utmost importance to enlarge specific subtrees in the layout itself by user interaction.

Zooming / Details on demand Supporting analysts with the ability to dig deeper into the tree-structure for most but the tiniest tree-structures is crucial. Thus we support the selection of a new root node through a mouse-click on the desired item. In our first prototype this triggered the recalculation of the diffs for the nodes' subtree as well as the maximum depth of unchanged nodes, the number of modifications and the subtree-size for each node in the subtree to

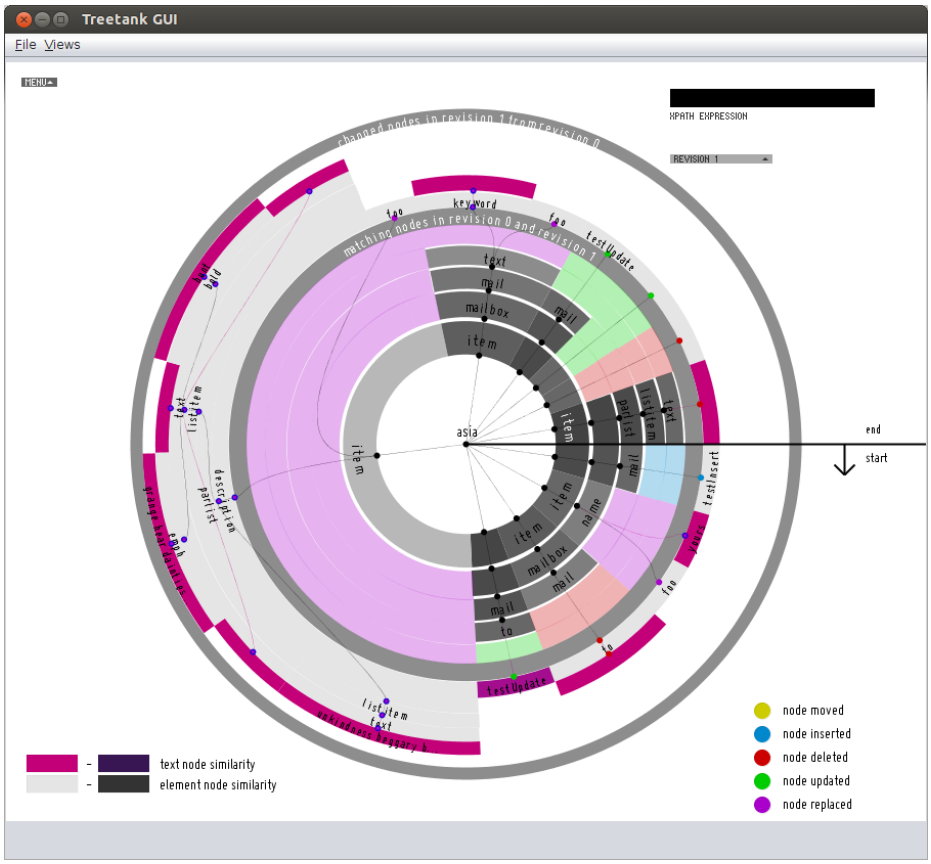


Fig. 21. Pruned by identical hash-values without building items for nodes with identical hash-values.

build new enlarged items. Thus, a simple subtree-selection triggered the whole visualization-pipeline.

However due to unnecessary recalculations despite in case of the item-based pruning our optimized approach just recalculates the maximum depth of the unchanged nodes in parallel utilizing all available cores and subsequently builds new items based on angle-upscaling.

Having described the new Sunburst-layout tailored to comparison of tree-structures with all available pruning methods and the ability to select a new root-node in detail the next section describes how a similarity score between different types of nodes is measured, which is used for color-encoding of the items.

Similarity measures Similarities for `Element`- and `Text`-nodes are measured differently. All non-leaf nodes of an XML-document are element nodes. However, if an element is a leaf node it is an *empty element*.

`Element`-node similarity is measured based on overlapping subtrees. Consider the comparison of tree-structures T1 and T2. The similarity score for an element in the aggregated tree-structure T_{aggr} is defined as

$$Sim(node_{T_{aggr}}) = \frac{descs(node_{T_{aggr}}) - mods(node_{T_{aggr}})}{descs(node_{T_{aggr}})} \tag{5}$$

The similarity score is normalized between $[0, 1]$. The nodes of type `INSERTED/DELETED/REPLACED` always are scored 0 as they do not have any equivalent in the other tree. In case of `SAME`-nodes the similarity depends on overlapping subtree-structures. If no modifications are in the subtree the score is 1. `UPDATED` nodes are modified and therefore add to the dissimilarity.

`Text`-nodes are leaf nodes which therefore have no child. We thus use a different similarity score defined as

$$Sim(node_{T_{aggr}}) = \begin{cases} Levenshtein(node_{T_1}, node_{T_2}) & \text{if } node \text{ } is \texttt{ UPDATED} \\ 0 & \text{if } node_{T_{aggr}} \text{ } is\texttt{INSERTED} \\ & \texttt{/DELETED/REPLACED} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Note that the similarity score for updated nodes is computed based on the diff-tuple which incorporates both node-IDs. The Levenshtein algorithm is used, which defines a similarity score based on per-character edit-operations to change one string-value into the other one and is normalized between 0 (no similarity) and 1 (same string-value).

In order to improve the usefulness of our visualizations we provide the ability to specify queries for specific nodes.

4.5 Querying

XPath 2.0 queries are currently partly supported by our XPath 2.0 engine. We also examined our Saxon[29] binding, which is surprisingly rather slow. However we did not measure time differences which is out of scope of this thesis. Future work most probabliy will include a Brackit[30] binding which is a "flexible XQuery-based query engine" developed at the TU Kaiserslautern. In contrast to Saxon it is specifically designed to work on top of databases and adding specific indexes for instance will be easy. However this feature is currently reevaluated.

We provide query capabilities on top of the agglomerated tree-structure and therefore query both compared revisions in parallel. The items are sorted by node-ID in parallel. Once all query results have been collected they are also sorted (the result is a sequence of node-IDs). A subsequent traversal of the items highlights query results.

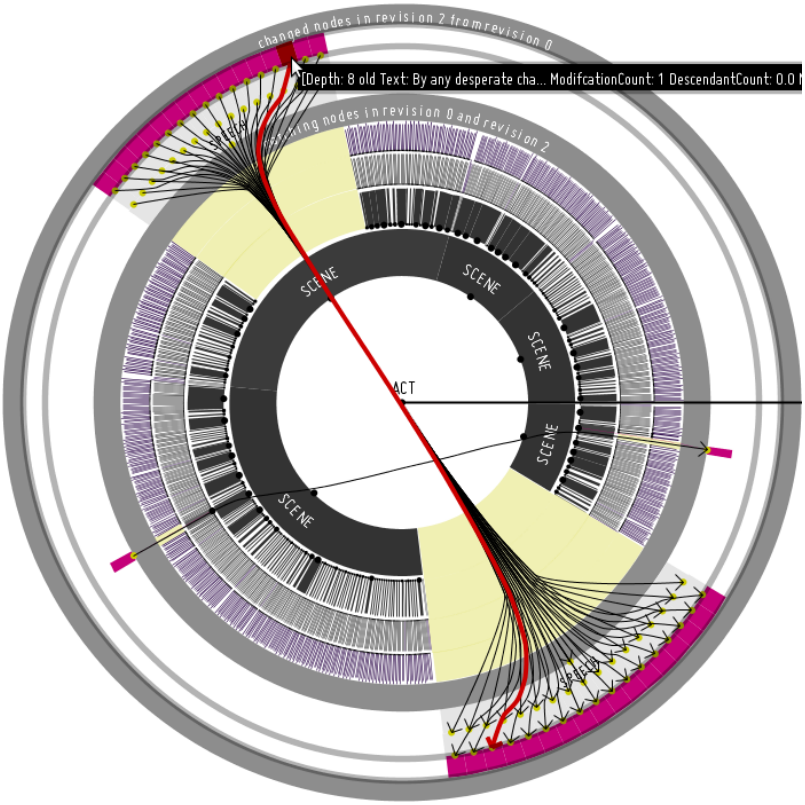


Fig. 22. Moves visualized using hierarchical edge bundles.

4.6 Visualization of moves

Hierarchical Edge Bundles[31] avoid visual clutter of subtree moves. The technique creates a path up to but usually not including the Lowest Common Ancestor (LCA) of a source-node down to the destination-node. The path is used to define control points for plotting a curve.

The LCA is defined as:

$$lca(a,b) = min\{c|a \prec c, b \prec c\} \tag{7}$$

A simple algorithm computes the LCA through adding elements on top of two stacks following the first node (moved from) and the second node (moved to) up to the root-node. In a next step the two stacks are processed in a loop removing the top element as long as identical node-IDs are found. The LCA is the last node-pair for which the node-IDs do match.

Fig. 22 illustrates two move-operations on a subtree in a shakespear XML-document.

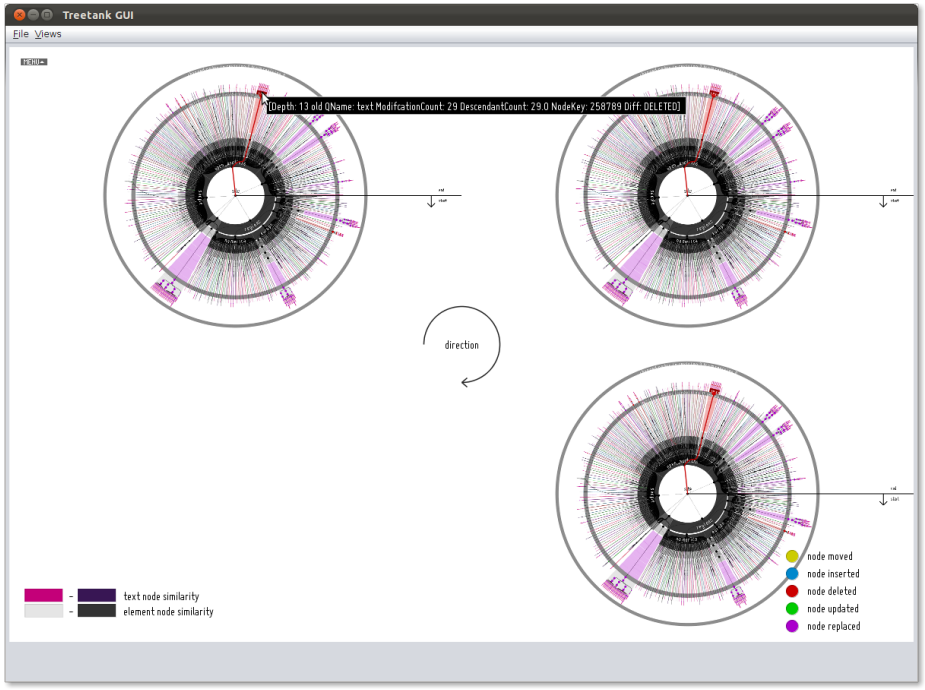


Fig. 23. Small multiple - differential variant.

Despite comparing two tree-structures we furthermore aim to support a broad overview about the changes (and similarities) of multiple trees.

4.7 Small multiple displays

Small multiple displays of the *SunburstView* provide an overview about the changes between several tree-structures. Two variants based on same-titled well known revisioning strategies as well as a hybrid variant are described next.

- *differential* The differential variant displays changes related to a base revision. This is especially useful if several tree-structures have to be compared to a common base version. The direction is clockwise. That is the upper left corner displays a SunburstView comparison between revision 0 and 1 (if zero is the base revision which is loaded), the right upper corner displays changes between revision 0 and revision 2 et cetera (Fig. 23).
- *incremental* The incremental variant displays changes related to the last revision in increasing order. Suppose we have loaded revision 2, then in the upper left corner revision 2 and 3 is compared, the upper right corner contains the comparison between revision 3 and 4 et cetera (Fig. 24).
- *hybrid* The hybrid variant displays changes just like in the incremental variant. However, a diff between the first revision and the last one to display is

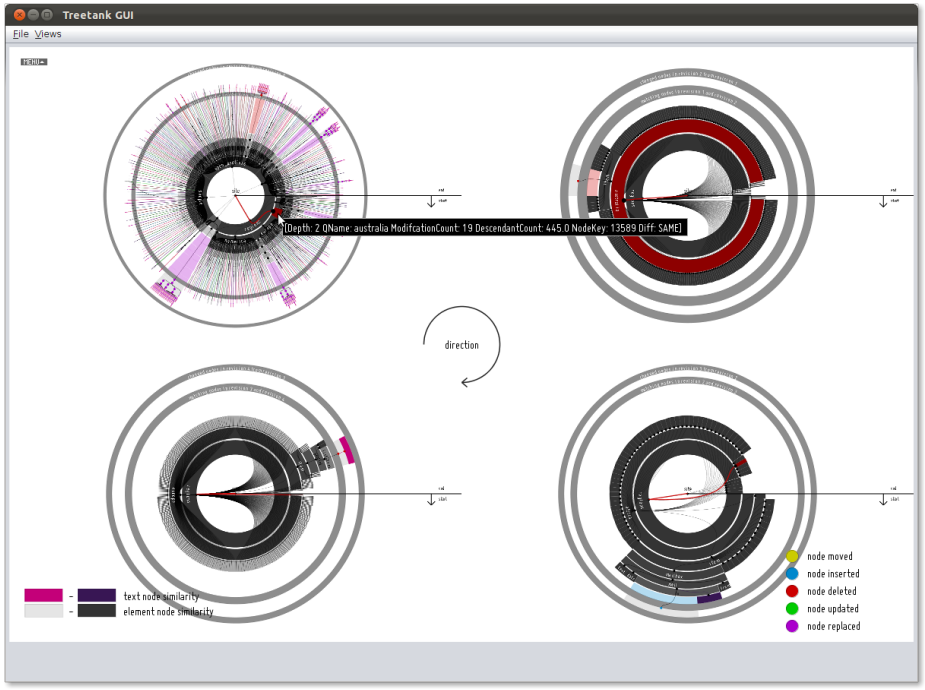


Fig. 24. Small multiple - incremental variant.

issued in the first place to get approximately the whole aggregated tree-structure. Upcoming changes in subsequent incremental comparisons are blackened. Furthermore, the colors denoting the changes lighten up in subsequent comparisons such that it is clear in which comparison the nodes are changed. During testing this view proved to be not as useful as we hoped. Nodes which are added in one revision and subsequently removed in another do not appear. In addition the colors denoting the type of change and during which comparison the change occured are too hard to distinguish in most cases.

The implementation parallizes the computation of the small multiple displays and stores each Sunburst-visualization in an offscreen image, which is appended to a list in the view. This list must be sorted by revision-number after the items for all visualizations have been created. The more recent revision-number is saved along with the buffered offscreen image to support the clockwise order of comparisons. The items of each visualization are saved in a simple datastructure in the model to support highlighting of items on mouse over through brushing and linking. The technique is illustrated in Fig. 23 and 24. Items are highlighted in red just like in the SunburstView but the same item based on a node-ID equivalence relation is linked and highlighted in all small visualizations. Note that we aim to highlight whole subtrees on hovered nodes in a future version

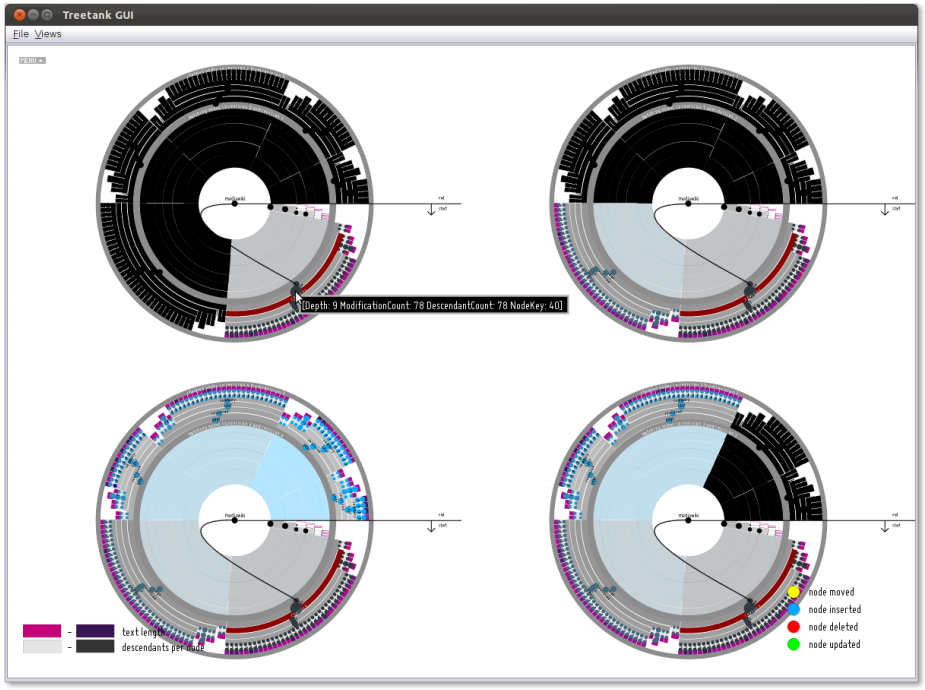


Fig. 25. Small multiple - hybrid variant.

such that moves in the subtree during upcoming comparisons in the incremental variant will be easily visible resulting in forests.

The two variants support all filtering methods described earlier. Next we provide a short asymptotic runtime- and space-analysis backed by performance measures.

4.8 Runtime/Space analysis and scalability of the ID-based diff

The runtime of the algorithm currently is bound by determining the subtree-size and modifications in the agglomerated tree-structure for each node. The runtime complexity thus is $O(n^2)$ whereas n is the sum of changed and unchanged nodes between two trees $T1$ and $T2$. However by storing the diff-tuples which only include the depth of the two compared nodes to form a very simple tree-structure we are limited to a preorder traversal. Building a more sophisticated tree-structure based on pointers or sequences denoting child nodes for instance in an in-memory Treetank structure will support a subsequent postorder traversal to determine the subtree-size and modifications for each node reducing the runtime to $O(n)$. Due to using Java7 which is not available for OS X we are not able to measure the performance on computers with four or more cores. However, counting the subtree-size and modifications is started in parallel to

building Sunburst items which consumes these through a Java `BlockingQueue`. Thus we assume adding more cores will speed up the creation of the items considerably as tree-structures often times are rather flat and have a large fan-out instead of being deep with high average levels/depths of the nodes, especially considering document-centric XML [9]. The `subtree-size` of the root-item is the size of the accumulated List or Map of diff-tuples observed from the ID-based diffing algorithm and thus does not need to be computed. The number of `modifications` of the root-item is also determined on the fly while observing diff-tuples. Fig. 26 depicts performance measures, the average of ten runs of the Sunburst-visualization on an 11MiB-document and an 111MiB-document of the XMark-benchmark. The documents are identical to the documents used in Chapter 3 for benchmarking. Thus we randomly modified the instances after every 1000st node in the 11Mib-document and after every 10000st node in the 111MiB-document. The hardware used is also identical (Core 2 Duo 2666Mhz, 4Gb RAM). It is obvious that the exponential growth in case no pruning is enabled is unacceptable. Showing the 111MiB-document without pruning lasted too long (¿ 15min for each run) such that we aborted the execution. By reducing the number of Sunburst-items which have to be created considerably each one of the pruning-mechanisms reduces the runtime tremendously. Usually the number of Sunburst-items to create is reduced so much that the exponential time to compute the modifications in each nodes' subtree plus the subtree-size itself is not measurable and the runtime becomes linear. Furthermore as we are currently not able to measure the impact of parallelizing this task with four and more cores we can not measure the impact of parallelization which might also considerably speed up the computation without pruning. We assume that the context switches with only one or two cores in fact slow down the computation and thus introduced a simple threshold value to .

Fig 27 shows benchmarking results comparing the runtime of the fastest pruning, pruning-by-hashes without creating items for identical hashes with move-detection enabled and disabled. We are able to determine that the move-detection usually is fast. Asymptotically it is bound by the size of the agglomerated tree-structure n, thus $O(n)$.

In summary we are able to conclude that the hash-based-pruning without creating Sunburst items is by far the fastest and that the move-detection usually does not inhibit the runtime of our algorithms considerably. Furthermore it is required to use pruning whenever the exponential time required to compute subtree-sizes and modifications therein significantly inreases due to a large aggregated tree. Adding more CPUs however should decrease the runtime as the computation of subtreesizes and modifications therein are computed in parallel.

4.9 Conclusion and Summary

We have introduced a multitude of visualizations ranging from a simple *TextView* displaying syntax highlighted serialized XML in the viewport appending new text during scrolling to a *SunburstView* and several *Smallmultiple* display variants

Fig. 26. GUI-performance.

	1000mods	**5000mods**	**10000mods**
min	36265.14	35717.34	33948.96
max	52459.38	61860.29	44451.06
average	39167.75	37650.67	36579.37

Table 6. Comparsion of different modification-schemas of a 111 MiB XMark instance (change every 1000st, 5000st and 10000st node).

based on the *SunburstView*. The *TextView* supports the visualization of structural differences based on the aggregated tree structure and through a color-coded background depending on the diff-type. However non-structural changes, for instance the similarity between updated String values can not be visualized. Our main contribution is a new Sunburst layout based on the idea of a semantic zoom, which places all changed nodes prominently between two rings and facilitates a global distortion to enlarge changed-subtrees and thus to shrink unchanged subtrees accordingly which are potientially uninteresting. Structural changes are visualized through color coding of the node in the overlaying node-link diagram. The type of diff is indicated through a color-encoding. Furthermore the similarity of `TextNode` values and `ElementNode`s are depicted based on predefined functions. Large matched subtrees result in a higher similarity score for element nodes, few character replacements/inserts/deletes increases the similarity score for text nodes.

Moreover three filtering techniques facilitate the analysis of large tree-structures. Pruning by itemsize is useful if changed *and* and unchanged nodes are of importance speeding up the creation of items considerably. However it does not affect the diff-computation. Thus, we also provide hash-based filtering tech-

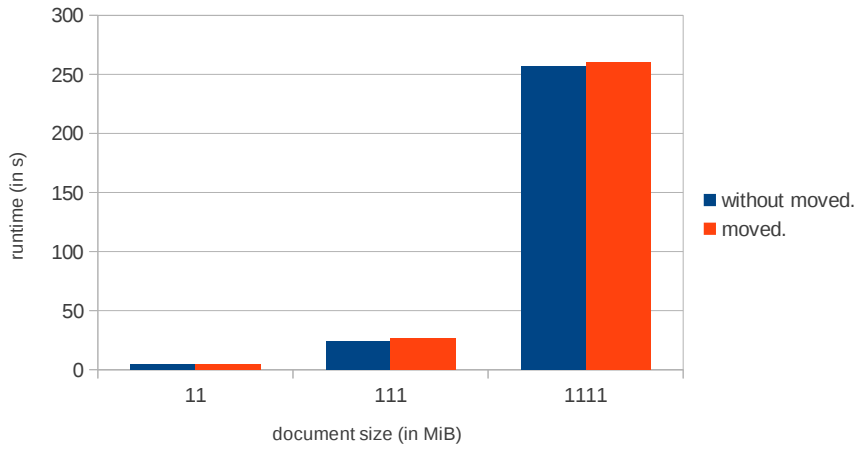


Fig. 27. GUI-performance using hash-based pruning without adding identical hash-values and move-detection enabled/disabled.

niques, which utilize the diff-algorithm with the optimization to skip the traversal of subtrees of nodes with identical hash-values. These filtering types usually reduce the number of items even more and accelerate the diff-computation. Note that the hashes include the unique node identifiers and other node-specific content, thus by using an exchangeable cryptographic hash-function th.

Move-detection is enabled on demand. Furthermore inserted/deleted subtrees in a row are summarized as replace-operations.

In addition to the *TextView* and *SunburstView* small multiple variants support the comparison of multiple trees (> 2). The number of comparsions is only limited by our implementation (which is only an implementation-detail) and the available screen space. Note, that the small multiple displays currently use too much unused screen space due to storing the whole visualization except the legends and menus in an offscreen image which are downscaled afterwards. As the extends of the main GUI-window usually are not squarified the space usually used for drawing menu-components and legends in the *SunburstView* is multiplied and wasted. However this is merely an implementation detail which will be fixed in a future version. The filtering techniques are also available in the small multiple variants.

5 Applications

5.1 Introduction

The last chapters described in detail the different components involved in our Visual Analytics approach for comparing tree-structures. This chapter exemplary studies three usage scenarios and describes additional preprocessing steps if necessary. To demonstrate the feasability of our approach with real world data the following three applications are studied:

- Import of Lexical Functional Grammar (LFG) XML exported files.
- Import of sorted Wikipedia articles by revision-timestamps.
- A Directory recursively watched for changes within a filesystem. The XML representation thereof is based on FSML[32].

5.2 LFG

Linguists often face the problem of having to compare Abstract Syntax Trees (ASTs). The LFG (Lexical functional grammar) in particular differenciates between two structures:

1. Constituent structures (c-structures) which "have the form of context-free phrase structure trees".
2. Functional structures (f-structures) "are sets of pairs of attributes and values; attributes may be features, such as tense and gender, or functions, such as subject and object."[33]

We obtained an XML-export of a collection of different versions of a c-structure/f-structure combination. To import differences between these XML documents in Treetank the FMSE-algorithm described in Chapter 2 and 3 is used for the simple reason that the XML-export does not include unique node identifiers. Thus we can not rely on node-IDs and have to figure out differences using our similarity metrics defined for leaf- and inner-nodes. Fig. 28 illustrates the visualization of a diff between two revisions.

It is immediately obvious by investigating the original XML-documents, that the FMSE-algorithm mismatched nodes in the first place which in effect causes a lot of edit-operations. Nodes are touched which are not changed and vice versa[13]. Thus, the FMSE-algorithm does not generate a minimum edit-script and executes too many edit-operations especially in case of the deletion and reinsertion of the old- respectively the new-`cstructure` subtree. In doing so, a subsequent visualization is impractical, for the simple reason that it almost mirrors the update-operations issued by the FMSE-algorithm for consecutive versions. As stated in Chapter 2 ID-less algorithms work best if leaf nodes and therefore especially text-nodes are very well distinguishable. That is the initial

[13] note, that the final outcome is nontheless correct in terms of the first tree-structure is correctly edited to represent the second

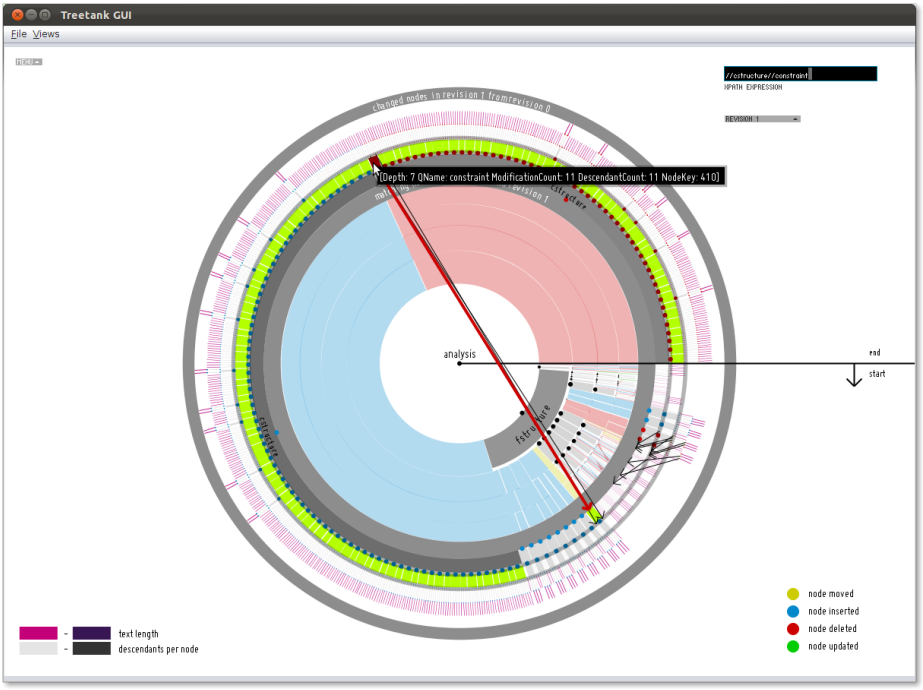


Fig. 28. LFG comparsion

matching of leaf nodes in most if not all cases really matches unchanged nodes, which is the desired behavior.

Having a close look at the input documents reveals a lot of changed or updated text-values (Listing 5.2). Our threshold value for matching subtrees in case of inner nodes is 0.5 (has to be > 0.5 for matching nodes) and thus the arg-elements are not matched.

For that reason we changed the leaf-node comparison slightly. Instead of just relying on the Levenshtein-distance for text-node values we also try to match all ancestor `QNames` [14] if the returned normalized value is $<=$ the threshold value defined for the similarity of leaf nodes. Note that only leaf nodes are considered to match if the threshold is $>$ than the predefined threshold. Hence, the leaf nodes are matched and "updated" by FMSE instead of deleting/inserting the whole subtree.

The result is depicted in Fig. 29.

We observe that still many nodes are changed, but by looking at the original XML-documents we observe that indeed a lot of nodes have been updated/deleted/inserted or replaced. For instance by scrolling both XML-documents at the end of the fstructure-subtree a lot of subtrees must have been inserted.

[14] ancestors are always element nodes, the document-root node is not visited

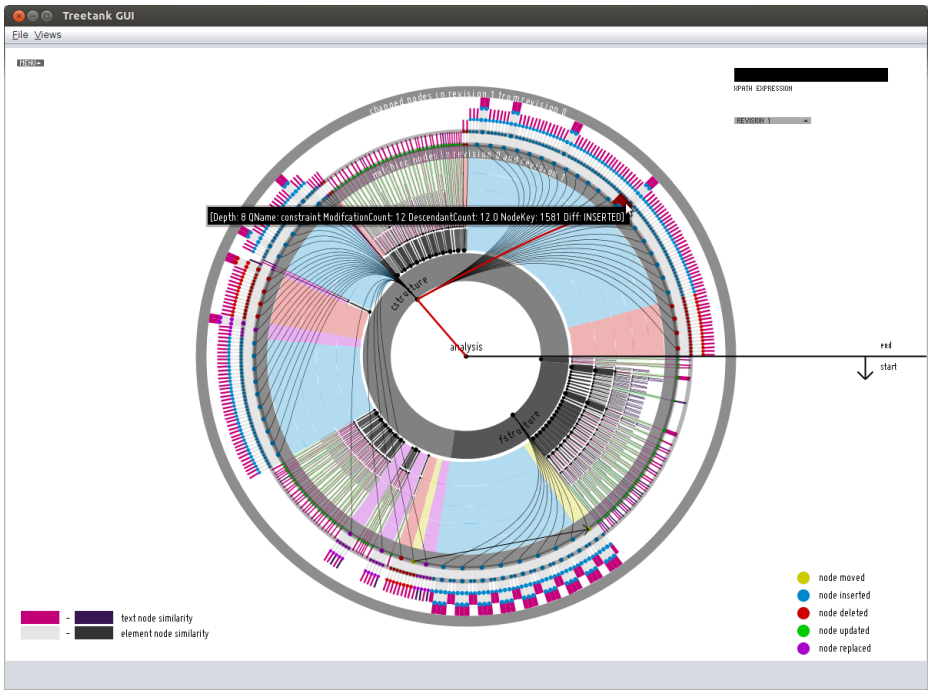


Fig. 29. LFG comparsion revised

Listing 1. CStructure/FStructure comparsion

```
<cstructure>                          <cstructure>
  <constraint>                          <constraint>
    <label>subtree</label>                <label>subtree</label>
    <arg no="1">102</arg>                 <arg no="1">163</arg>
    <arg no="2">ROOT</arg>                <arg no="2">ROOT</arg>
    <arg no="3">-</arg>                   <arg no="3">-</arg>
    <arg no="4">83</arg>                  <arg no="4">145</arg>
  </constraint>                         </constraint>
  <constraint>                          <constraint>
    <label>phi</label>                    <label>phi</label>
    <arg no="1">102</arg>                 <arg no="1">163</arg>
    <arg no="2">var:0</arg>               <arg no="2">var:0</arg>
  </constraint>                         </constraint>
  ...                                   ...
</cstructure>                         </cstructure>
```

However as a direct consequence of this short evaluation it is inevitable to rely on unique node identifiers if we want to visualize minimal edit-scripts in case of updated leaf node values which are very distinct to the other trees'

leaf node values or worse the leaf nodes are very similar. Comparisons based on similarity-metrics are always based on heuristics. Assuming the export of the XML-documents includes unique identifiers of the nodes, importing requires utilization of the internal diff-algorithm to simply store incremental changes, the changes from the old- to the new-revision. The following steps have to be implemented:

1. Import of the initial XML-document.
2. Import the first updated XML-document to a temporary resource.
3. Utilize the internal Treetank diff-algorithm to determine changes based on unique node-IDs. Whenever an edit-operation is encountered the appropriate transaction-method has to be executed.

Furthermore this approach applies to every XML-document which incorporates unique node identifiers.

Next, we study the import of several articles from Wikipedia ordered by timestamp of their revisions.

5.3 Wikipedia

Wikipedia is studied as an application to demonstrate the feasability of our approach on a large text corpora. However several issues have to be solved during preprocessing.

- Wikipedia is dumped as a very large XML-file. "'The XML itself contains complete, raw text of every revision"'[34] and thus is a full dump instead of an incremental- or differential-dump describing the changes an author performed. Appendix C depicts a small example of the structure of an Wikipedia XML-dump. Moreover WikiText, which is a proprietary markup format is stored as plain text and not replaced by XML-markup. Thus, the content of an article in each revision is a huge `TextNode` which includes the full text instead of just denoting the changes and its context in some form. Differences between the actual content of several revisions of an article on a node-granular level can not be found with a state of the art native XML database system, which supports revisioning of the data, as a direct consequence. The database systems' best bet is to generate a character based delta between the two large text-nodes.
- Furthermore due to the fact that XPath 2.0 should be usable without requiring an XQuery/XPath Full Text 1.0 implementation and our overall goal is to analyse the temporal evolution of the stored data, WikiText markup has to be converted into semistructured XML fragments. To accomplish this the Wiki2XML parser from the MODIS team [35] is used.
- Another issue arises because the Wikipedia dump is not sorted and we want to analyse and visualize the temporal evolution during several snapshots. Neither articles are sorted by date/time nor revisions of the articles are. Thus for the subsequent revisioned import the revisions have to be sorted

with their associated article metadata (page id, author...). For the simple reason that we have to deal with large amounts of data an external sorting algorithm has to be used. Instead of implementing an own approach, Hadoop is a natural choice. Its *MapReduce*[36] framework is a "programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes". The overall process of the programming model is divided into two functions.

1. `Map` is a function that splits the problem into subproblems are distributed among worker nodes in a cluster. Logically it is a function of the form $Map(k1, v1) \rightarrow list(k2, v2)$ where $k1$ and $v1$ is an input *key* and *value* and the output from the function is a list of *key*/*value* pairs (`list(k2, v2)`). After that the MapReduce framework groups the values according to the keys.
2. `Reduce` is a function of the form: $Reduce(k2, list(v2)) \rightarrow list(v3)$. It receives a *key* and a list of grouped *values* and performs any computation which might be feasable and returns another list of *values*.

The `map`-function receives input data splitted into records. Most MapReduce frameworks rely on a programmable mechanism to do this. In Hadoop it is the responsibility of the `InputStream` in conjunction with a `RecordReader`. The following steps describe the implementation of sorting Wikipedia by Hadoop.

1. `XMLInputStream` in conjunction with an `XMLRecordReader` splits records on *revision-elements* with the *page*-metadata prepended. The **timestamp** of each revision denotes the key, the whole subtree of each *revision*-element is saved as the value for the `map` input. The algorithm is straight forward. A `SAX-Parser` is used to determine starting and ending of each record.
   The implementation utilizes a configurable record identifier such that it is adjustable to schema changes in the Wikipedia dump.
2. `XMLMap` just forwards the received key/value pairs.
3. `XMLReduce` finally merges consecutive revisions with the same *page-id* and **timestamp** by means of Saxon, an XSLT/XQuery-processor. Our XSLT stylesheet to achieve the concatenation is small yet maybe not straight forward (Appendix B).

Once the dump is sorted with MapReduce it has to be imported into Treetank. First of all, as we have splitted the data on `revision`-elements and prepended the `page`-metadata we have to construct a new root node, which is simply done by prepending a `mediawiki` start-tag, then writing the result of the Hadoop-run and appending a corresponding end-tag to a new file. Otherwise it is no valid XML-document and can not be parsed by XML-parsers.

Next, it is the responsibility of a Wikipedia-Importer to import the revisioned file which is now in ascending timestamp order of all revisions from all pages/articles. It utilizes the FSME-implementation described in detail in Chapter 2 and 3 to just import incremental changes between the latest stored revision in Treetank and a shredded List of `XMLEvent`s in a temporary database/resource.

Therefore as data is read-in from the XML document with a StAX-Parser the page-metadata as well as the whole revision-subtree is saved in a simple main-memory collection, a `List`, assuming the XML-content of an article can be put in main memory. The assumption holds true as every revision can be parsed and rendered in a web-browser. Everytime a `page` end-tag is encountered the page-id is searched in the already shreddered revision. If it is found the FMSE algorithm is called for the found `page`-element, such that the encountered differences are shreddered subsequently. In case the XPath expression returns no results, that is the page-ID is not found a new page is going to be appended as a right sibling to the last `page`-element.

The Importer for Wikipedia allows the usage of a timespan (hourly, daily, monthly, yearly) which is used for the revisioning. Whenever a new timestamp is encountered, which differs from the current timestamp depending on the chosen timespan the transaction is commited.

Fig. 30 demonstrates the result of importing 50 articles of Wikipedia sorted by article-revisions and an hourly commit [15]. Revisions 70 and 71 are compared. The *edit-distance* used by our FSME-implementation to determine and update the stored Treetank-data with the encountered differences in the first place works reasonably well, as most text-nodes are distinguishable very well. We used the hash-based pruning to enlarge regions of interest. The *TextView* on the right side is particular useful to quickly reveal several changes in particular updated text whereas a label in the *SunburstView* only depicts the new updated text content in case of `TextNode` updates and usually the content is too big to be displayed within the arc, especially if the content contains whole paragraphs as is usually the case for Wikipedia. Both text values, the old- and the new-value is displayed on mouseover. We quickly detect that a lot of paragraphs have changed by looking at the *SunburstView*. Details are depicted by either hovering an item or scrolling down in the *TextView*.

Another visualization (Fig. 31) depicts the situation without pruning and without including the number of modifications of a node in its subtree to scale the Sunburst items, the segments for each node.

The changes in the last page/article are hardly visible. Most items denote unchanged nodes without any modifications in their subtree and thus do not carry any useful information, if we just want to quickly determine changes. This and the fact that we do not include the modification weight to scale items, that is only the subtree size of each node matters, results in considerably reduced itemsizes of changed items.

Despite viewing changes between subsequent revisions which in case of importing only 50 articles most often results in appending or updating one article it is possible to view changes between arbitrary revisions. Fig. 32 reveals changes between revision 90 and 106. Thus seven articles have changed and it is very easy to determine regions of interest and to further drill down into the tree through selecting a new root node in the *SunburstView*. The page/article with the name "ArgumentForms" has been changed considerably. The page-nodes which are

[15] only hours in which changes occured are reflected

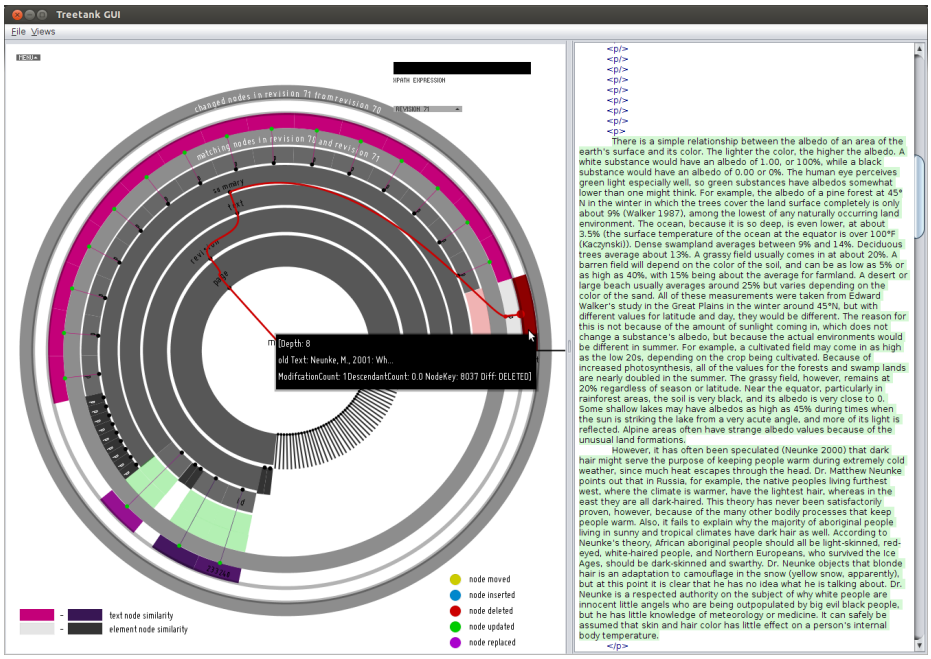


Fig. 30. Wikipedia comparsion.

containers of the whole article do not have enough matching subtree-nodes in common between the two revisions, such that they are not matched, too. The FMSE-algorithm thus deletes the whole article and inserts the article as a new first child of the "wikimedia" root-element. The changes in the other articles mainly either include single paragraphs which are updated or larger subtrees which are replaced. Detailed changes are depicted on demand, that is if we drill down into the tree and/or scroll down in the *TreeView*. The transparent red square in the *SunburstView* in Fig. 32 illustrates the article/page to which we have scrolled in the *TextView*.

Besides comparing only two revisions small multiple display variants facilitate the comparison between several revisions (currently at most five revisions). Fig. 33 illustrates changes between revisions 70,71 (upper left), 71,72 (upper right), 72,73 (bottom right) and 73,74 (bottom left). We are quickly able to determine that except in the SunburstView comparing revisions 73 and 74 all other small multiples change or delete/insert the same article. The comparison between revision 70,71 and 72,73 reveal a lot of updated paragraphs. Between revisions 71 and 72 the same article has been deleted and inserted. We are quickly able to determine that a lot of paragraphs have been added as the inserted article includes more descendants than the article in the old revision. Thus the FMSE-algorithm does not match the page-nodes due to very different subtrees. Repacitulate that in order to match inner nodes, more than at least half of the nodes in both sub-

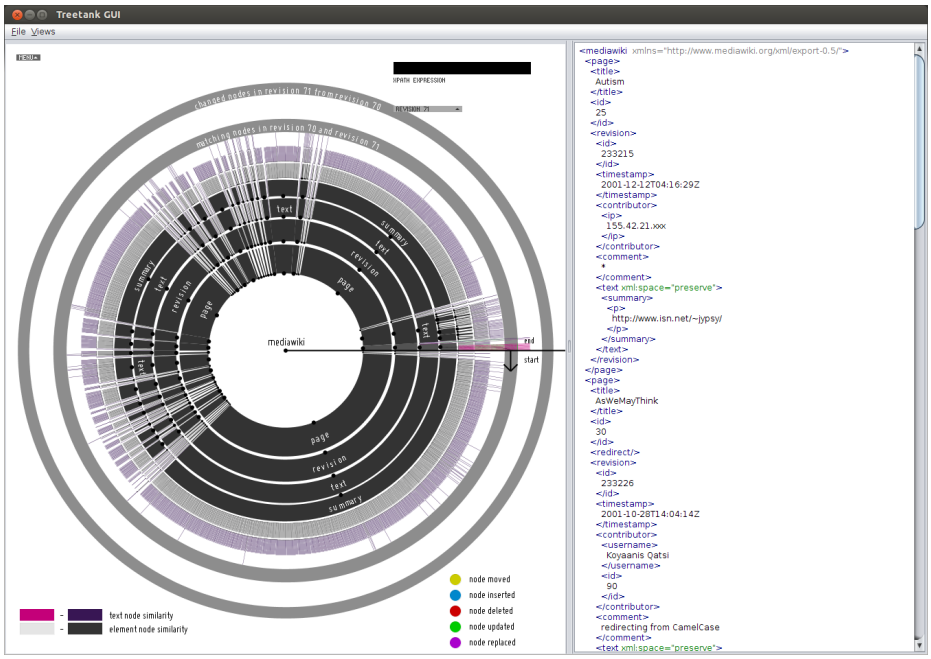


Fig. 31. Wikipedia comparsion without pruning and including the modification weight to determine the size of a SunburstItem

trees have to be matched. To gain a better understanding about the differences between revision 73 and 74 we switch to the *SunburstView* in conjunction with the *TextView*. We are able to quickly determine that the deleted and inserted page denotes the same article (autism) once more (Fig. 34). The article changed from a single paragraph which just included a URL to a slightly more profound description of autism.

Fig. 35 finally reveals that between revision 10 and 13 articles are added which is expected. Articles are most probably going to be altered in later revisions once most or all 50 articles have been prepended. However revision 14 updates an article added in one of the first 10 revisions.

5.4 Import of Filesystem-based tree-structure

Filesystems usually organize data in a tree-structured hierarchy whereas a unique *Path* denotes the location of a file. As a direct consequence it is an optimal use case to demonstrate the feasability of our proposed approach as the tree-structure can not be neglected if we quickly want to detect differences in the folder/file-structure based on snapshots.

Use cases are manyfold, ranging from monitoring software-evolution based on the package/class hierarchy to monitoring if employees stick to organizational policies.

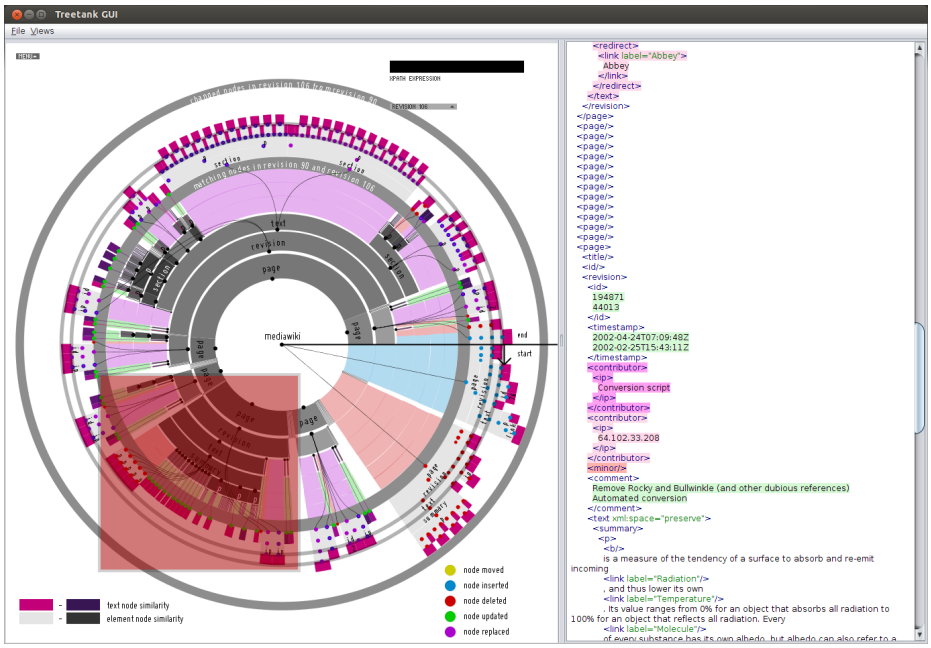


Fig. 32. Wikipedia comparsion pruned by itemsize

In order to take snapshots of a directory with all subdirectories we study two approachs:

1. FSML representation based on a python script [32], which is executed several times to obtain snapshots based on an XML-representation which are afterwards imported in Treetank. The differences between the snapshots are calculated based on the FMSE-algorithm described in Chapter 2 and 3.
2. FSML representation based on an initial import of a directory with all its subdirectories. Changes in that directory and all subdirectories are afterwards monitored.

The File System Markup Language[32] is an XML-dialect developed by Alexander Holupirek to represent the tree-structure of filesystems by folders and files as well as metadata of certain file-types.

The first approach is considerably simpler than the import of Wikipedia due to the snapshot-creation which results in different files. Therefore we do not have to reorder whole subtrees according to timestamps in the first place to obtain a subsequent import ordered by time. Instead applying the FMSE-algorithm on the last imported revision and the new XML document is sufficient.

Fig. 36 reveals the differences between revision 0 and 4 using the hash-based pruning on manually taken snapshots of the src-folder of our GUI project (slightly outdated) with a Python-script, developed by Alexander Holupirek, too.

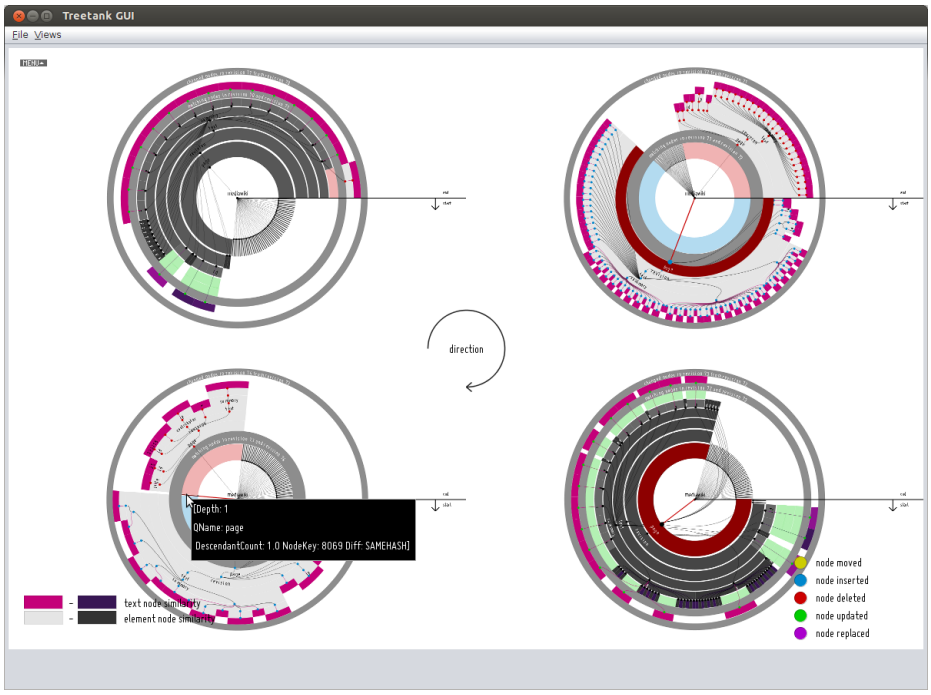


Fig. 33. Wikipedia comparsion - depicting differences through the incremental smallmulltiple variant

We are able to detect possible hotspots of development on the package/class-level. It is immediately obvious that a ForkJoin-Quicksort implementation has been deleted recently. Another interesting observation is the recent work on BSplines which have been added to facilitate the Hierarchical Edge Bundling technique described in Chapter 4. Furthermore we are able to spot recent work on a new *Treemap*-view. The XPath-query `//element()[@st_mode="0100664"]` highlights nodes whith the appropriate mode.

The alert reader might think that some of the nodes for instance the subtree of the "sunburst"-subtree which are plotted should not have been processed by the ID-based diff algorithm because we are utilizing the hash-based diffing algorithm, but by close inspection it is revealed that these nodes have different hash values (on mouseover the details are revealed, that is type `DiffType.SAME` instead of `DiffType.SAMEHASH`). We currently do not color-encode nodes which have identical hash-values differently to nodes with identical node-IDs but different hash-values. Moreover up until now we have always used the ID-based diffing mode which does not include namespace/attribute comparisons. However, once including checks for namespace/attribute equivalence we are able to detect updates of certain nodes (Fig. 37) due to last access time of a file/directory which is denoted through the `st_atime`-attribute. Each time either a namespace- or an

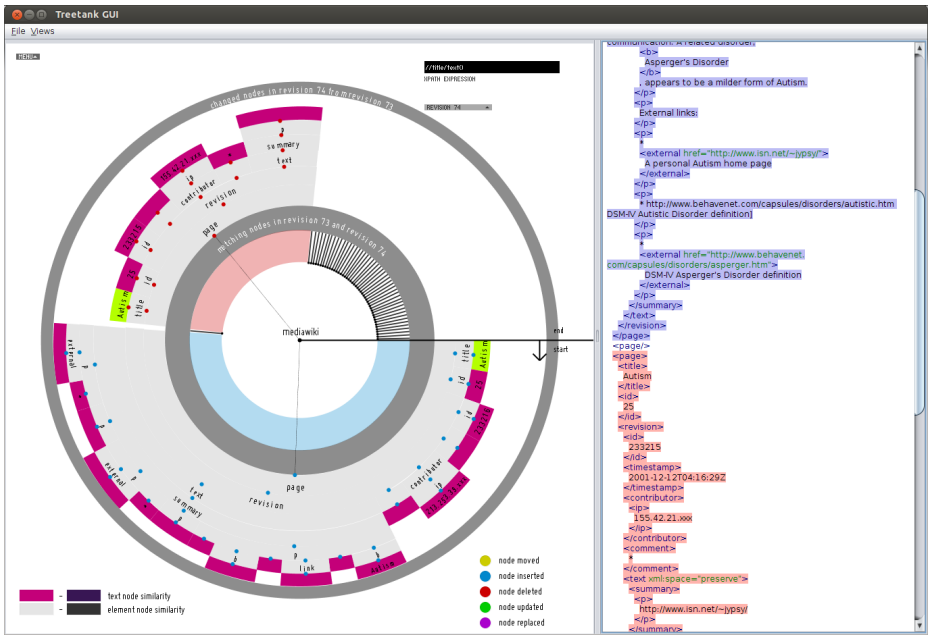


Fig. 34. Wikipedia comparsion - depicting differences between revision 73 and 74

attribute-node differs the parent `ElementNode` is emitted as being updated once the full diff mode is enabled.

We conclude, that the preprocessing step of matching nodes in the FMSE-algorithm works reasonably well on FSML-data as no similar `TextNode`s are included. However we aim to support the extraction of metadata for several kinds of files which might introduce the possibility of mismatches in the future.

In order to avoid any mismatches and therefore executing too many update-operations on nodes which have not changed at all as well as to avoid the costly execution of the FMSE-algorithm in the first place the capabilities of current filesystems to register for modification-events are additionally used. The steps are as follows:

1. To obtain the hierarchical structure of filesystems the Java7 Filesystem-Walker API is used instead of the Python-Script. A new database is created in Treetank with a standard resource named "fsml" whereas the hierarchical structure is mapped to the resource while traversing a directory.
2. The new `WatchService` is used to detect subsequent changes in a watched directory. In order to support watching all subdirectories as well a suitable datastructure as for instance an associative array has to be used in the first place. Java does not permit recursive watching of all subdirectories, as it is not supported by some filesystems.

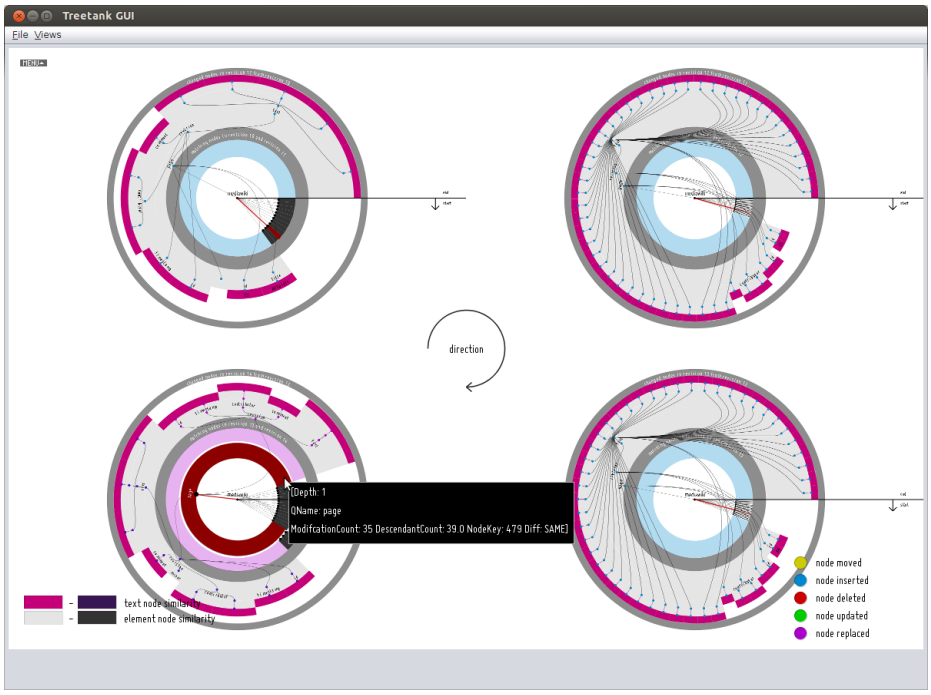


Fig. 35. Wikipedia comparsion - incremental smallmultiple variant depicting changes between revisions 10,11,12,13 and 14 from the upper left to the bottom left in clockwise order.

The `WatchService` detects the following events:

- `ENTRY_CREATE`: File or directory has been created.
- `ENTRY_DELETE`: File or directory has been deleted.
- `ENTRY_MODIFY`: File has been modified.

Therefore moves and renames are not supported out of the box. The path to the directories and files is translated to an XPath query, which locates the appropriate node in the database/resource. In case a new file or directory has been created a new `ElementNode` is prepended as a first child of the parent node as filesystem-trees are unordered. In this case the XPath-query translates the parent-path into the appropriate XPath-query, that is it is of the form `/dir[@name='pathComp1Name']/dir|file[@name='pathComp2Name']`. The path component usually denotes a directory whereas the last component might be a directory or file in case of an `ENTRY_DELETE` event. Deleted directories have to be removed from a WatchService/Path-mapping in an associative array. Furthermore all paths have to be saved in another datastructure to denote if the deleted path pointed to either a file or directory. Thus another associative array is used to save a Path $\Leftrightarrow$ EPath mapping for inserted nodes. EPath is a Java enum to determine the type (file or directory).

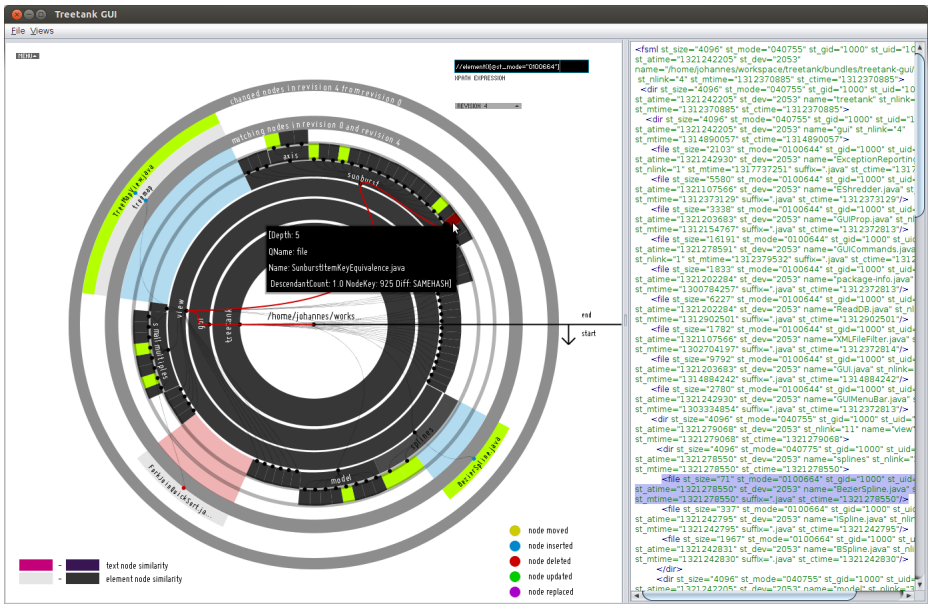


Fig. 36. FSML comparsion on the GUI src-folder

Listing 2. FSML structure

```
<fsml>
  <dir name="Desktop">
    <dir name="Lichtenberger">
      <dir name="Bachelor">
        ...
      </dir>
      <dir name="Master">
        <dir name="Thesis">
          <dir name="figures">
            <file name="fsml-incremental.png" suffix=".png"/>
            ...
          </dir>
          <file name="thesis.tex" suffix=".text"/>
          ...
        </dir>
        ...
      </dir>
    </dir>
    ...
  </dir>
</fsml>
```

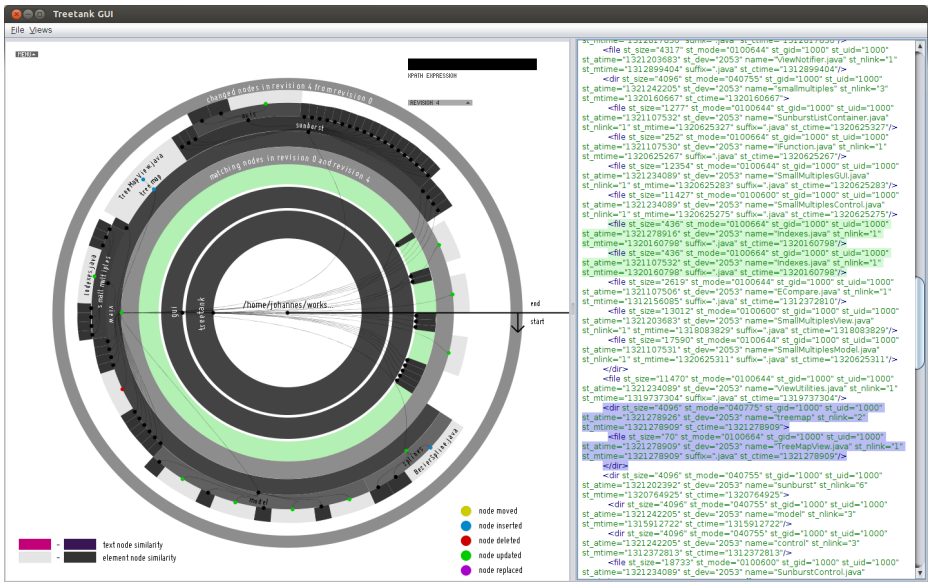


Fig. 37. FSML comparsion on the GUI src-folder using a full diff including namespaces and attributes

The FSML-subdialect currently used is very simple. Listing 5.4 provides an example of the simple structure. Directories are mapped to `dir`-elements, whereas files are mapped to `file`-elements. Note that the names can not be used without preprocessing to denote the elements, as for instance whitespaces are not permitted in QNames. Thus, either whitespace characters and other not allowed characters are escaped or as in our case saved as attribute-values in `name=""`-attributes. The labels in the visualizations are optionally based on this special attribute.

While this representation currently does not incorporate most of the strengths FSML usually provides it is easy to add metadata about files and to incorporate text-files. Optionally instances of custom classes are pluggable to provide any type of extensibility. Thus it is possible to provide extractors for certain types of files and to incorporate text-files into the FSML-representation itself, possibly by adding a link to another resource in the fsml-database.

Fig. 38 is an example of mapping the author's Desktop-folder to a database in Treetank (move- and replace-detection is disabled). Subsequent revisions are commited every five minutes.

Most files and directories are unchanged. Usually subtrees of changed items (besides the processing-subfolder which has been deleted) are rather small compared to most unchanged nodes. Thus, we study the effect of changing the filtering method. Pruning by same hashes instead of the itemsize is displayed in Fig. 39. Sunburst items for the large subtrees of the unchanged nodes are not

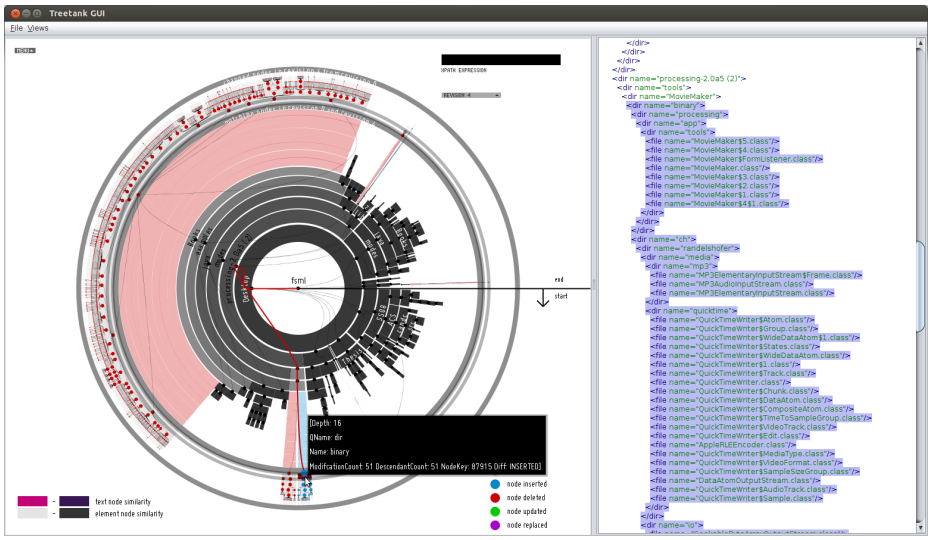


Fig. 38. FSML comparsion of "/home/johannes/Desktop"

generated. As a direct consequence items of interest are enlarged and thus much better visible.

A sequence of comparisons between several revisions is viewable with the incremental small multiple display variant (Fig. 40). We used the hash-based pruning without generating items for nodes with identical hash-values, to keep the number of items to a minimum. By hovering the items and comparing the values as well as subtree sizes we are able to determine a directory-rename between revision 2 and 3 in the bottom right view. A `rename`-operation however is not supported by the Java `WatchService` such that usually, depending on the filesystem a delete- and an insert-event in either order is received.

In general we are able to answer questions such as

- Auditing: Which files are updated/inserted/deleted or not during a sequence of snapshots?
- Does an employee adhere to company instructions regarding installed software? (by file-extensions or analysing the binaries and storing an additional attribute with "true" or "false" values)

5.5 Summary

This chapter introduced a few use cases for comparing tree-structures. Depending on the activated views different characteristics are revealed.

The smallmultiple differential-variant should be used to compare similar, different tree structures. Choosing the incremental variant reveals changes in temporal evolving trees. In case the tree is rather small (for instance less than

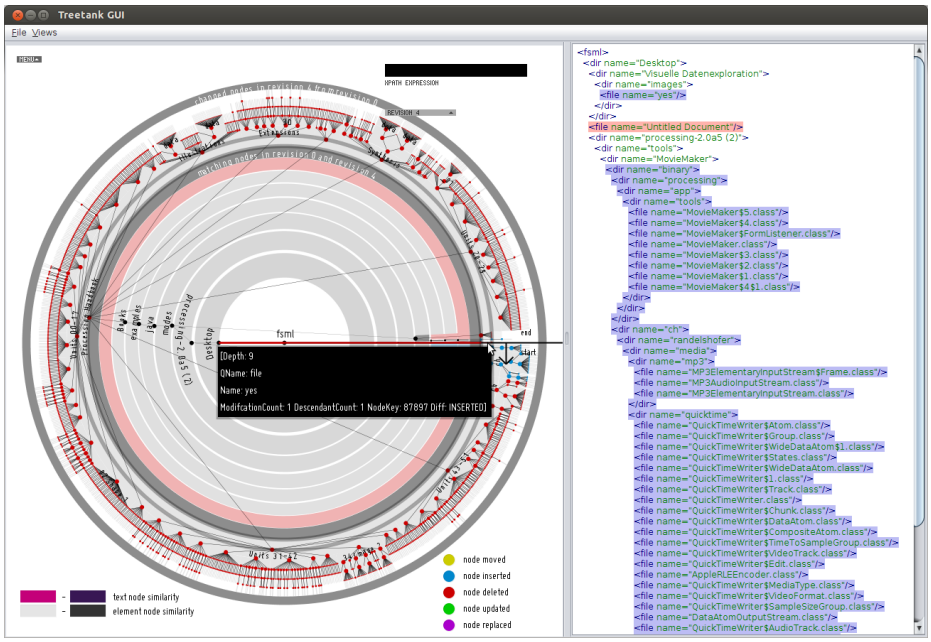


Fig. 39. FSML comparsion of "/home/johannes/Desktop" pruned by same hashes

5000 nodes) it is arguable that the views can be used without filtering. However, once more items have to be created it is necessary to choose one of the filtering techniques to keep the numer of items small and to prune nodes/subtrees of no interest. Hovering through linking and brushing facilitates keeping track of changed nodes. When one of the pruning techniques is enabled, inserted nodes might not be visible in upcoming comparisons. However, this indicates that the items have not been changed within later revisions.

The standard *SunburstView* reveals all kinds of differences and furthermore allows two zooming operations. The zooming/panning which transforms the viewing-coordinates through affine transformations may be used if the number of items is small and therefore animations are possible whereas a left-mouseclick on an item transforms the item into a new root-node. As already explained in Chapter 4 this involves either a transformation based on node-copies and adjusting angles or the full pipline has to be passed in case of the itemsize-pruning.

We have merely scratched the surface of the possibilities of our current approach. The small multiple displays will greatly benefit from a temporal XPath-extension. Some proposed axis as `next::`, `previous::`, `future::`, `past::`, `revision::` already have been implemented but have to be incorporated in an XPath/XQuery-engine. Simple queries as for instance
`//dir[@name='wiki-sorted']/future::node()` will reveal nodes with the at-

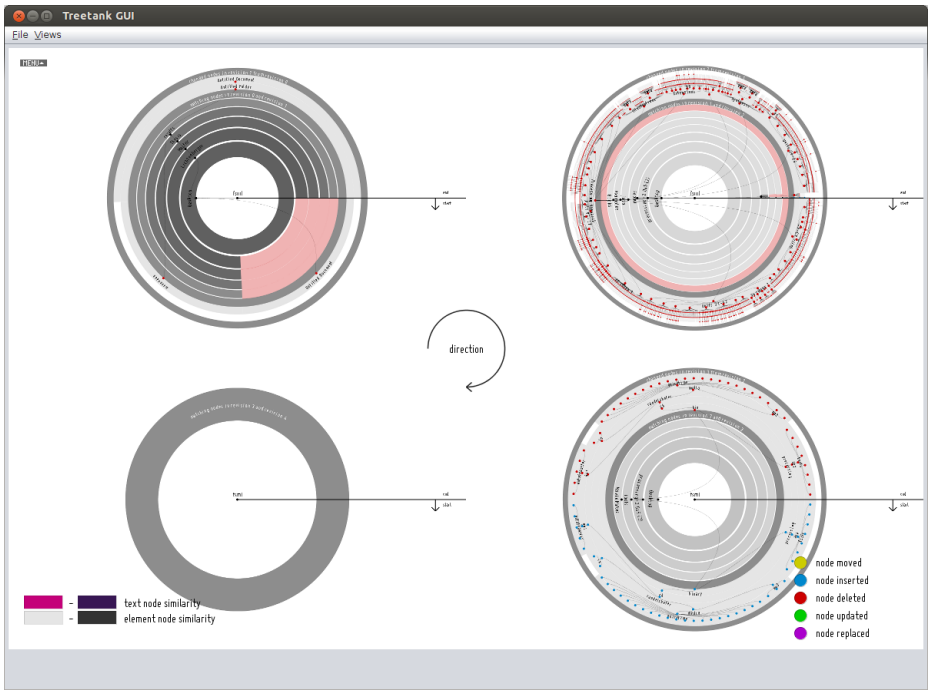


Fig. 40. FSML comparsion of "/home/johannes/Desktop" (small multiple displays – incremental view)

tribute `name='wiki-sorted'` in all future revisions. The start-context will be based on the currently opened base-revision.

6 Summary, Conclusion and Future Research

6.1 Summary

A full pipeline has been implemented backed by our secure tree-storage system Treetank ranging from (extensive) preprocessing, ID-less and ID-based diffing-algorithms to new layout algorithms developed for comparing tree-structures.

Almost all data is subject to preprocessing. Wikipedia has been used to demonstrate the feasability of our approach for comparing the temporal evolution of several articles by their inherent tree-structure. The Wikimedia Foundation provides only full Wikipedia dumps which are not sorted by revision-timestamps. Therefore it must be sorted at first using Hadoop. Importing structural changes which should be revealed by the visualizations afterwards requires a diff-algorithm which does not expect unique node IDs. Thus the FMSE-algorithm described in Chapter 2 has been implemented (Chapter 3) as well as several missing edit-operations in Treetank (`copy`, `move`, `replace`), whereas others have been extended to adhere to the XPath/XQuery Data Model (XDM) to simplify the internal ID-based diff-algorithm and to add to the visualizations' expressiveness[16]. The FMSE-algorithm is furthermore useful to compare multiple similar trees. In both cases either the evolution of a tree or different trees are imported as snapshots/revisions in a single Treetank resource. In the latter case changes between a base tree and several other trees are imported. Leveraging the ID-based diff-algorithm the small multiple display differential variant visualizes the comparison between two revisions/trees in currently four regions of the screen (top left, top right, bottom right, bottom left). Other studied applications include the comparison of snapshots of a directory in a filesystem and two revisions of XML-exported LFG-grammars.

An ID-based diff-algorithm has been developed to compare imported data based on snapshots/revisions, whereas the comparison is *not* restricted to both subsequent revisions and whole documents. It supports two modes. Either only structural nodes which denote the structure of a tree (currently the subset of XML nodes `DocumentNode`, `ElementNode` and `TextNode`s) are compared or structural and non structural nodes (adding `NamespaceNode`s and `AttributeNode`s). A fast version utilizes hash-values of each node generated during the import. These are derived from the nodes' descendants and used to reduce the runtime of the algorithm. The whole subtree of nodes with identical hash-values is skipped.

All visualizations capable of visualizing an aggregated tree-structure trigger the ID-based diff-algorithm if needed. A specialized *SunburstView* visualizes the aggregated tree-structure which is build through collecting diff-tuples observed from the diff-algorithm. A selection of a new root-node togehter with a simple undo-operation allows to drill down into the tree as known from other Sunburst-visualizations. However the costly recalculation of diff-tuples, modifications and subtree-sizes of each node is omitted in almost all cases. Instead we recalculate

[16] for instance no two adjacent text-nodes are ever created

the maximum depth of unchanged nodes and scale a new list of items to fit the new boundaries. A notable exception is the itemsize-based pruning as the diff-tuples are usually not saved to avoid their constant in-memory space consumption. Instead they are subject to garbage collection. The type of diff however is saved in the created Sunburst item and used by the visualization through color coding. A new Sunburst layout algorithm has been designed to specifically support the analysis of structural changes, which are featured prominently between two rings whereas the ring which is closer to the root node encloses almost all unchanged nodes except those who are in subtrees of updated nodes. The second ring denotes the maximum depth of changed nodes. Thus the area between the two rings encloses all changed nodes as well as unchanged nodes in the subtree of updated nodes. The first changed node in a subtree is dislocated from its original place to reside between the two rings supporting a tree-ring metapher based on the analogy of an aging tree in the nature which adds growth rings every year. Note, that the analogy is especially appropriate for temporal tree-structures from which our approach emerged.

The *SunburstView* in comparison mode furthermore facilitates the comparison of large trees by utilizing one of three available pruning techniques. First, an itemsize based approach to filter nodes based on their extend is provided. It speeds up the creation of Sunburst items, but does not affect the runtime of the diff-algorithm used in the first place. Second, a method utilizing the hash-based diff-option skips whole subtrees of unchanged nodes with identical hash-values and thus builds solely interesting items to the task at hand (which is analysing structural and non-structural changes). A third variant even skips the creation of items for nodes with unchanged content and identical hash-values altogether. This variant is especially interesting for very large tree-structures to minimize the runtime as well as the memory consumption and to provide maximum space for potentially interesting subtrees with changed nodes.

Besides, sometimes it is interesting to view the whole aggregated tree in an overview. To emphasize structural changes in addition to our semantic zoom the arc of an item is based on the `subtree-size` of a node as well as of the number of modifications in its subtree. A slider facilitates the setting of how much impact the number of modifications has. Note that the number of modifications is multiplied by a constant factor as modifications are usually small compared to the whole tree. Furthermore the `subtree-size` must be added as a fallback which is needed to cover the case of zero modifications in the subtree.

In addition to a novel Sunburst layout we provide three different small multiple display variants. Two of them, the incremental- and the differential-view compute the diffs and the subsequent visualizations in each of the four screen regions (top-left, top-right, bottom-right, bottom-left assuming five subsequent revisions of the tree-structure to compare exist) in parallel. The incremental variant displays changes related to a sliding window of two subsequent revisions. The differential version displays changes related to the opened base revision, that is either a reference tree if multiple similar trees are compared or a snapshot of a temporal tree-structure. On mouse-over items are highlighted in each

region in which they are present. Compared to an icicle-view which connects unchanged items in each revision by splines our approach features changes much more prominently through the semantic zoom/the tree-ring metapher as well as the inclusion of the number of modifications to determine the start- and end-angle of a Sunburst item. A hybrid variant currently suffers from a lack of visually distinguishable colors encoding the type of change of a node. Furthermore nodes added and removed or vice versa during the comparison of multiple revisions are not visible as the overall agglomeration which is visualized in each small multiple are build through comparing the opened base-revision and the last revision. However to provide an expressive view adding diff-tuples during subsequent comparisons of all revisions to an agglomeration is necessary. Yet, a first implementation proved to be too slow in informal tests.

6.2 Conclusion

We presented several visualizations and diffing algorithms tailored to comparing tree-structures. In retrospective the goals described in the introduction have been achieved. The GUI-frontend embedding the visualizations aids analysts in comparing tree-structures ranging from temporal- to similar-trees.

Our approach based on a tight storage integration integrates several edit-operations utilizing an expressive aggregated tree-structure build through observing diffs from the ID-based diff algorithm. Unlike the straight forward approach of building two in memory datastructures with nodes of both revisions which are going to be our algorithm uses less space and thus is much more useful. Furthermore it supports replace-, update- and move-operations and corresponding diff-types and is faster in comparison to . In contrast to other visual tree-comparison tools our prototype incorporates several edit-operations and diff-types including a moved and replaced-nodes which is rarely seen in other visualizations. Filtering-mechanisms facilitate the comparison of very large tree-structures. Note that besides CodeFlows[22] and Treevolution[17] no other approach described in Chapter 2 to the best of our knowledge incorporates such filters and therefore most likely are not useful to inspect large datasets. Two filtering techniques depend on hashes which are used to compare the whole subtree of a node including itself. Subtrees are skipped from comparison if the hashes are identical. Thus they are neither traversed nor items are build subsequently. Besides building an aggregated tree-structure from observing changes by the ID-based diff-algorithm no state is involved.

To the best of our knowledge it is the only approach besides the work presented in [] capable of displaying comparisons through various linked visualizations depicting the tree-structures at various levels of detail. Small multiple displays provide a high level view of the differences between multiple tree-structures. Linking and brushing techniques highlight nodes.

Unlike state-of-the-art visualizations of tree comparisons described in Chapter 2 our approach is entirely database driven. The DBMS is tailored to temporal tree-structures which are stored as snapshots. Each node in the database/resource is unique and remains stable throughout all revisions. As most trees do not

inhibit unique node-IDs we implemented an ID-less diff-algorithm called FMSE based on inner-node/leaf-node similarity measures to import differences, which eventuates in a new revision. A smilarity-measure for `element`-nodes depends on the amount of overlapping nodes in its subtree. In contrast `text`-nodes are compared based on the Levenshtein-algorithm as `text`-nodes are always leaf-nodes in a tree. Other algorithms might be easily included for instance to compare numerical values. Once the tree-structures or revisions thereof are imported, we are able to use a fast diff-algorithm ($O(n+m)$) based on the generated node-IDs. Furthermore we are able to utilize hashes to further speed up the algorithm.

The next section evaluates our approach similar to Chapter 2 but in more detail according to several attributes.

Evaluation Criteria

- *SpaceFilling* Space filling techniques try to maximize the usage of available screen space and thus facilitate a higher information density. Sunburst layouts are generally space filling, but in contrast to Treemaps lack space filling properties in the corners. However in our case the corners are used to display GUI components and legends. Our small multiple variants currently require too much unused space as we trigger a normal sunburst visualization each time occupying the whole viewport which is afterwards downscaled. However usually the screen-viewport is rectangular instead of squarified whereas the width is greater then the height depending on the extends of the main window. Thus, space is unused at the left- and right-border of the visualization. In a future version we will use a squarified clip of the view depending on the maximum depth to generate minimized offscreen buffers. Other space filling approaches include Icicle plots (Code Flows) which are comparable in space consumption as always the two complete tree-structures to compare are plottet. In our case unchanged nodes are not plotted twice. A special stable Spiral-Treemap layout has been proposed which utilizes all available screen space and remains relatively stable after changes. Usual Treemap layouts suffer from abrupt layout changes during modifications of the underlying data.
- *Hierarchy* The Hierarchy in Sunburst layouts is very well depicted due to the adjacency based layout whereas it is not as obvious in Treemaps which encapsulate children in parent-rectangles. Cushion Treemaps have been developed for better readability of hierarchical relationships using shading. However, compared to adjacency based layouts they still suffer from worse readability. Code flows utilizes Icicle plots which are rectangular views of the radial Sunburst layout, thus adjacency based and very well readable. Our approach optionally plots a node-link diagram on top of the Sunburst layout to further illustrate the relationships such that the hierarchy is at least as well depicted as in other node-link diagrams (for instance in TreeJuxtaposer and Treevolution). To draw a node-link diagram on top of a Sunburst layout facilitates a higher information density. Individual nodes in the node-link overlay are color-coded based on their type of difference. Furthermore their

child/parent relationship, the links between the nodes are displayed through connections (straight lines or bezier curves). Thus we are able to map attributes to the arc/extension of the Sunburst items, the color of the arcs and the color of the dots/nodes in the node-link diagram. The radius of the nodes in the node-link diagram however currently is not used to encode another attribute. To maximize the information density theoretically it is possible to use histograms in the Sunburst items instead of just using one color for the whole item to visualize multiple attributes instead of one. However we are confident that the items usually are too small, such that a whole histogram in usually small items are certainly not be readable.

- *Labels* Labels in a Sunburst-view are usually plotted in a radial layout, otherwise overplotting occurs. To support better readability of item-labels our visualization is able to be rotated. However, node labels of Treemaps and Icicle plots usually are better readable due to their rectangular display in comparison to circular plotted labels. Thus we decided to extend our *TextView* to take the agglomerated tree-structure into account. It is an ideal partner of the *SunburstView* as it provides better readability of small subtrees but lacks an overview about the entire tree-structure and all differences which are depicted by the *SunburstView*. Furthermore moves are not visualizable.
- *Similarity of ID-less tree-structures* Some proposed tree-to-tree comparsion visualizations depend on node-IDs (Spiral-/Contrast-Treemap[19]) or the comparison technique has not been meantioned. Others depend on domain characteristics (Treejuxtaposer[21], Code Flows[22]). Juxtaposer seems to rely on unique leaf node labels. Otherwise it is not obvious how to map node labels to their postorder rank on a region plane. In contrast to Treejuxtaposer and other proposed systems our prototype is able to compare every kind of tree-structure.
- *Structural changes* Our visualizations, in particular the *SunburstView* and the small multiple display variants support the highlighting of all kind of structural-changes (inserts, deletes, updates, replace and move-operations) through color-coded nodes. In case of moves- links, currently depicted as arrows, denote the movement from their original- to their target-place using hierarchical edge bundling to avoid or at least reduce visual clutter due to overlapping lines or curves. Furthermore the extend of the sunburst items is based on the nodes' subtree size *and* the number of modifications in the subtree. Most other visualizations do provide a global distortion to further emphasize changes except Treejuxtaposer to the best of our knowledge.
- *Non structural changes* are attribute-value changes of nodes. Changes in `TextNode`s are color coded to denote that the value is `UPDATED` through coloring the node in the overlapping node-link diagram accordingly. Furthermore the color of the Sunburst item reflects their similarity using the Levenshtein distance.
- *Filtering* is one of our primariy concerns. The focus of this thesis evolved around the idea of comparing Treetank-resources by interactive visualizations. As Treetank is a secure storage system it naturally often times stores very large tree-structures. Thus, our first consideration was to prune items

which cannot be visually perceived, that is Sunburst items which are too small are not created. However this only affects the creation of Sunburst items and does not improve the ID-based diff-algorithm. To speed up both, the ID-based diff algorithm as well as the construction of Sunburst items we developed a diff-algorithm variant which skips subtrees of nodes with identical hash-values and thus moves both transactions to the next node in the XPath `following::`-axis. Accordingly, in case large subtrees are skippable the diff-algorithm using hash-comparisons is much faster. Besides even less items usually have to be created and in case of a subtree-selection only the itemsizes have to be recalculated instead of reinvoking the diff-algorithm, recalculate the descendant- and modification-count for each node, and the preorder traversal of the agglomerated tree-structure with all stack-adjustements is unnecessary. Pushing the idea of filtering by hashvalues to its limits involves to avoid the creation of items for the nodes with identical hash-values. These filtering techniques furthermore enlarge changed subtrees naturally as the arc of the items is based on the *subtree-size* of each node and the number of *modifications* therein. TreeJuxtaposer once more along with Code Flows to the best of our knowledge seems to be the only system which is able to handle large tree-structures but the diff-algorithm of TreeJuxtaposer relies on unique node labels. Code Flows does not provide a global filtering method such that their filtering is comparable with our method to drill down into the tree. However, due to filtering by queries the filtering is mightier. Implementing such a behavior in our prototype will be straight forward allowing the user to specify an XPath-query and invoke the diff-algorithm for each result on both revisions. Our current approach is able to filter differences and related context nodes on a global basis and on subtrees once a user drilled down into the Sunburst visualization with a guaranteed visibility of modifications much like in TreeJuxtaposer.

Table 7 summarizes all visualization-approaches according to the evaluation criterias specified above.

Most proposed visualizations rely on stable unique IDs (the Contrast Treemap[19], Code Flows[22], TreeJuxtaposer[21] (unique node labels)) and do not include the detection of `replace`- and `move`-operations. However Code Flows visualizes moves due to spline connections between matching nodes with the same ID in two adjacent icicle plots. Due to the reliability on unique node IDs these visualizations do not cover trees which do not include unique node-IDs or labels. In contrast our prototype is able to determine and visualize these tree-structures utilizing the FMSE-algorithm to import differences through similarity metrics for leaf- and inner-nodes and a bottom-up LCS computation.

Interactive Visual Comparison of Multiple Trees(IVCoMT[18]) provides the ability to compare multiple trees through various linked views allowing the comparison on different levels. However we assume that the similarity metrics are de-

[19] the labels are better readable in Treemaps and Icicle plots due to their rectangular nature instead of drawing the label on an arc

[19] rotating however might prevent some labels from overplotting

visualization approaches	space filling	hierarchy	labels	ID-less comparison	structural changes	non structural changes	filtering
Sunburst	o	+	+[17]	+	+	+	+
Spiral-/Contrast-Treemap	+	o	+	-	o	+	-
Treevolution	-	o	o[18]	-	o	-	-
Code Flows	+	+	+	-	o	-	+
Juxtaposer	-	+	o	-	+	-	o
Ripple Presentation	-	+	o	-	+	-	-
IVCoMT	+[19]	+	+	-	+	o	-

Table 7. Comparsion of tree-to-tree comparison visualizations. Appendix D provides a detailed legend.

fined for phylogenetic trees which have unique labels and that the visualizations are tailored to many small trees in comparison to a few very large tree-structures or a temporal evolving tree as in our case.

6.3 Future Research

Many topics are subject to further research.

- First of all we want to incorporate already developed temporal XPath axis in an XQuery processor, most probably Brackit[30], which is important to further analyse temporal evolving tree-structures. Furthermore in order to support fast query-response times it is inevitable to provide indexes.
- A recently developed path summary is usable to significantly speed up the comparison of very large tree-structures if isomorphism is not an issue, for instance by providing a high level matrix overview about the changes.
- Extend the Hierarchical Edge Bundles to use a gradient color to indicate the direction of moves instead of arrows.
- Building an index structure for visualizations of consecutive revisions (the small multiple incremental-variant and the SunburstView comparing consecutive revisions).
- Evaluation and integration/implementation of various other tree-similarity measures.

A Treetank

A.1 General persistent storage enhancements

While not exclusively developed for our tree-to-tree comparsion for completeness we want to meantion several techniques which have recently been developed to support the persistent storage of large tree-structures efficiently (with a minimum space overhead).

- Values of `TextNodes` are compressed by using the Deflate-algorithm which combines the LZ77 algorithm and Huffman coding. Decompressed values are cached in memory once they are requested. Furthermore only values which are greater than a certain length-threshold are compressed.
- Pointers to neighbour nodes, the first child and the parent node are persisted as ranges.
- As ranges are persisted node-IDs are efficiently compressed.
- Whole `NodePage`s are compressed using the very fast Snappy-algorithms due to a lot of compression/ decompression in case of many modifications of the same NodePage in the BerkeleyDB log.
- In case of the `DocumentNode` only the firstChild-pointer and descendant-count has to be persisted. Similar the serialization of `TextNode`s doesn't include the childCount and descendantCount as well as the firstChild-pointer.
- The `NamePage` which is used to store String-names which are commonly repeated in XML-documents now contains index-mappings whereas we opted for different indexes based on an element-QName index, an attribute-QName index and a namespace/URI-index. This allows an XPath- or XQuery- Optimizer to use the index for queries as `count(//@attribute='foo')` or `count(//element()='bar')`. We furthermore aim to provide backreferences to the nodes as we encountered far too long response-times on larger tree-structures. A path-like index therefore might be inevitable in the long term.
- Deletion of `attribute`- and `namespace`-nodes in subtrees.

A.2 ACID properties

Consistency rules have been enhanced while developing our prototype. The following provides a brief overview about the ACID-properties of Treetank.

1. *Atomicity* is ensured through the transaction layer and the interchangeable backend (currently BerkeleyDB). As such atomicity is even guaranteed in case of power failures, errors and crashes.
2. *Consistency* as of now involves the checking of `QName`s for validity. At all times no adjacent text nodes are created which is consistent with the XPath/X-Query Data Model (XDM). We also introduced XML entity encoding for the XML characters serialization (in the `XMLSerializer` as well as the new `StAXSerializer` and `SAXSerializer`). Furthermore we check attribute `QName`s for duplicates and throw an appropriate runtime exception if a new

attribute insertion would yield duplicates. Similarly insertion of duplicate namespace prefixes for namespaces of the same parent element-node are prohibited.

3. *Isolation* is guaranteed through *Snapshot-Isolation* which is based on the transaction-, page- and I/O-layer through versioning. Furthermore currently only one write transaction is allowed per resource. To maximize the properties of tree-structures the implementation of concurrent write-transactions on different subtrees with appropriate locking is in development.
4. *Durability* is guaranteed through the backend. The transaction log created by Treetank's BerkeleyDB-binding implemented as a cache to store all changed pages and nodes is written and flushed on transaction commit.

A.3 Axis

The axis in Treetank have been changed to adhere to the `hasNext()` and `next()` specifications of the `Iterable` interface. The check if `getNext()` is true is added to all axis such that `hasNext()` is idempotent. It simply checks a flag which is set in `resetToLastKey()` to make sure the transaction points to the node after the last call to `hasNext()` without changing the node-ID to which to move in the next call to `next()`. Furthermore the transaction now is not moved forward in `hasNext()` anymore which is done only when calling `next()`. Instead a variable denoting the next node-ID is set which is used by the `next()` implementation to move to the next node. Furthermore `next()` now also is idempotent, simply checking if it has been called before. When true and `hasNext()` has not been called immediately before it is first called by `next()`.

Levelorder-Axis The `LevelOrderAxis` is described in algorithm 7. Just like other axis to traverse certain regions or the whole tree-structure in Treetank it is based on the `Iterator/Iterable` Java interfaces to support the `foreach`-loop and iterator-based iteration. All other axis are changed as described in Appendix A. *mFirstChilds* is a double ended queue to remember all first childs of each node for a subsequent new depth ($depth + 1$). `processElement()` is invoked to add non structural nodes, that is `attributes` and `namespaces` to the queue. After initialization the queue is empty and *mNextKey* is initialized to either the current key (if self is included), the right sibling node key if there is one or the first child node key. The NULL_NODE_KEY is a special node key to denote that the traversal is done.

A.4 Edit operations

The `copy`-operation adds the capability to add whole subtrees of another resource or revision to the currently opened *resource/revision*. Actually three `copy`-operations exist. Either the subtree is inserted as a `firstChild`, `rightSibling` or `leftSibling` of the currently selected node. The node to copy must be a

Algorithm 7: LevelOrderAxis (hasNext())

input : boolean mFirst, Deque mFirstChilds, long mKey
output: node key of next node

```
1  if getNext() then
2      return true;
3  resetToLastKey();
4  // Setup.
5  INodeReadTrx rtx ← getTransaction();
6  IStructNode node ← rtx.getStructuralNode();
7  // Determines if it is the first call to hasNext().
8  if mFirst == true then
9      mFirst ← false;
10     return processFirstCall();
11 // Follow right sibling if there is one.
12 if node.hasRightSibling() then
13     processElement();
14     // Add first child to queue.
15     if node.hasFirstChild() then
16         mFirstChilds.add(node.getFirstChildKey());
17     mKey ← node.getRightSiblingKey();
18     return true;
19 // Iterate over non structural nodes (attributes/namespaces).
20 if mInclude == EInclude.NONSTRUCTURAL then
21     processElement();
22 // Add first child to queue.
23 if node.hasFirstChild() then
24     mFirstChilds.add(node.getFirstChildKey());
25 // Then follow first child on stack.
26 if !mFirstChilds.isEmpty() then
27     mKey ← mFirstChilds.removeFirst();
28     return true;
29 // Then follow first child if there is one.
30 if node.hasFirstChild() then
31     mKey ← node.getFirstChildKey();
32     return true;
33 // Then end.
34 resetToStartKey();
35 return false;
```

structural node, that is either an `ElementNode` or a `TextNode`. In case the transaction is located at a `DocumentRootNode` which is a special document node, which can not be deleted and exists in every revision the read transaction has to move to the first child in the first place.

The `move-` operation just like the `copy-` operation is implemented in three different versions, `moveSubtreeToFirstChild(long)`, `moveSubtreeToRightSibling(long)` and `moveSubtreeToLeftSibling(long)`. Details are omitted, however the new constraint that at no time no adjacency text nodes are allowed as well as keeping the child-count of parent nodes and the descendant-count of ancestor nodes before and after a move consistent adds a lot of complexity.

The implementation of the replace-operation is straigth forward using a delete- followed by an insert-operation chaining the two to provide an atomic operation.

Besides, the `remove`-operation has been modified to merge adjacent `TextNode`s if the deleted node has two neighbour text nodes to adhere to the XPath/XQuery Data Model (XDM) and to provide meaningful visualizations. `insert`-operations as of now check if the new `TextNode` would yield two consecutive `TextNode`s and, if so, update the value of the existing node with a concatenation of the old- and new-values instead. Therefore the storage consistently avoids consecutive `TextNode`s.

A.5 Visitor

A special `VisitorDescendantAxis` executes a visitor specific implementation for each visited node before moving to the next node in preorder. The return value of the methods a visitor has to implement (a visitor specific implementation for each node-type) guides the traversal in the axis. The following result types are currently available:

- `EVisitResult.TERMINATE`, terminates the traversal of the (sub)tree immediately and returns false for upcoming `hasNext()` calls.
- `EVisitResult.CONTINUE`, continues preorder traversal.
- `EVisitResult.SKIPSUBTREE`, signales that the axis skips traversing the subtree of the current node.
- `EVisitResult.SKIPSIBLINGS`, signales that the axis should move to the next following node in document order which is not a right-sibling of the current node.
- `EVisitResult.POPSKIPSIBLINGS`, is a special type which signals that the element on top of the internal right sibling stack must be popped, which is needed to implement for instance the deletion-visitor for the second FMSE-step.

A.6 DeletionVisitor core

Algorithm 8: FMSEDeleteVisitor: delete(pWtx)

input : NodeWriteTrx pWtx
output: EVisitResult type

```
1  long nodeKey ← pWtx.getNode().getNodeKey();
2  // Case: Has no right and no left sibl. but the parent has a right
       sibl.
3  pWtx.moveToParent();
4  IStructNode node = pWtx.getStructuralNode();
5  if node.getChildCount() == 1 AND node.hasRightSibling() then
6  |   pWtx.moveTo(nodeKey);
7  |   pWtx.remove();
8  |   return EVisitResult.SKIPSUBTREEPOPSTACK;
9  pWtx.moveTo(nodeKey);
10 // Case: Has left sibl. but no right sibl.
11 if !pWtx.getStructuralNode().hasRightSibling() AND
   pWtx.getStructuralNode().hasLeftSibling() then
12 |   pWtx.remove();
13 |   return EVisitResult.CONTINUE;
14 // Case: Has right sibl. and left sibl.
15 if pWtx.getStructuralNode().hasRightSibling() AND
   pWtx.getStructuralNode().hasLeftSibling() then
16 |   boolean removeTextNode ← checkIfTextNodeRemove();
17 |   if removeTextNode then
18 |   |   pWtx.remove();
19 |   |   return EVisitResult.CONTINUE;
20 |   else
21 |   |   pWtx.remove();
22 |   |   pWtx.moveToLeftSibling();
23 |   |   return EVisitResult.SKIPSUBTREE;
24 // Case: Has right sibl. but no left sibl.
25 // similar to above case (omitted)
26 // Case: Has no right and no left sibl.
27 mWtx.remove();
28 return EVisitResult.CONTINUE;
```

B XSLT stylesheet to combine consecutive pages/revisions

Listing 3. XSLT stylesheet to combine consecutive pages/revisions

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">

  <xsl:output method="xml" indent="no"
    omit-xml-declaration="yes" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <xsl:copy>
      <xsl:for-each-group
        select="descendant-or-self::page"
        group-by="concat(id, revision/timestamp)">
        <page>
          <xsl:copy-of select="* except revision" />
          <xsl:for-each-group
            select="current-group()/revision"
            group-by="xs:dateTime(timestamp)">
            <xsl:apply-templates
              select="current-group()" />
          </xsl:for-each-group>
        </page>
      </xsl:for-each-group>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="revision">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

C Wikipedia Dump - XML structure

Listing 4. Wikipedia Dump - XML representation

```
<mediawiki xml:lang="en">
  <page>
    <title>Page title</title>
    <id>67365</id>
    <restrictions>edit=sysop:move=sysop</restrictions>
    <revision>
      <timestamp>2001-01-15T13:15:00Z</timestamp>
      <contributor>
        <username>Foobar</username>
      </contributor>
      <comment>I have just one thing to say!</comment>
      <text>A bunch of [[text]] here.</text>
      <minor />
    </revision>
    <revision>
      <timestamp>2001-01-15T13:10:27Z</timestamp>
      <contributor>
        <ip>10.0.0.2</ip>
      </contributor>
      <comment>new!</comment>
      <text>An earlier [[revision]].</text>
    </revision>
  </page>

  <page>
    <title>Talk:Page title</title>
    <id>127455</id>
    <revision>
      <timestamp>2001-01-15T14:03:00Z</timestamp>
      <contributor>
        <ip>10.0.0.2</ip>
      </contributor>
      <comment>hey</comment>
      <text>
        WHYD YOU LOCK PAGE??!!! i was editing that jerk
      </text>
    </revision>
  </page>
</mediawiki>
```

D Visualizations - Evaluation legend

- space filling:
  - -: The visualization is not space filling.
  - o: The visualization is considered to be space filling. However the visualization lacks space filling properties in the corners of the screen (for instance the SunburstView) or in other areas (Icicle-plots which are a rectangular variant of the SunburstView).
  - +: The visualization occupies the whole screen space.
- hierarchy:
  - -: The hierarchy is not displayed via encapsulation of child nodes in parent nodes, through adjancency or through connecting nodes via curves/direct lines (for instance in a matrix).
  - o: The hierarchy is visualized but edge crossings are possible which results in visual clutter (certain node-link diagrams) or they are visualized through encapsulation in a parent node (Treemaps). In the case of Treemaps it is usually not easy to visually perceive the nesting even if cushing Treemaps reduce the issue.
  - +: The hierarchy is displayed without any visual clutter and parent/child relationships are easily perceivable.
- labels:
  - -: The labels are not present.
  - o: Labels are plotted, however visual clutter resulting from label overplotting occurs frequently.
  - +: Labels are plotted without the possibility of label overplotting.
- ID-less comparison:
  - -: The visualization is not capable of visualizing tree-structures which do not incorporate unique node-identifiers.
  - +: The visualization is capable of visualizing tree-structures which do not incorporate unique node-identifiers.
- structural changes:
  - -: The visualization is not capable of visualizing structural changes (should never happen).
  - o: The visualization visualizes unchanged, inserted and deleted nodes. Either no other operations as for instance moves are displayed or all operations are displayed through connecting unchanged nodes via splines. Thus, all changes are only visible through close inspection, especially in case of single node modifications or small subtree changes.
  - +: The visualization visualizes all update primitives including updates and moves through visual encoding (for instance usually color coding).
- non structural changes:
  - -: The visualization is not capable of visualizing non-structural changes.
  - o: The visualization visualizes non structural changes, that is updated values.
  - +: The visualization is capable of visualizing differences in the value.
- pruning/filtering:

- -: The visualization does not allow filtering interesting nodes.
- o: The visualization supports culling, either so called frustum culling, that is nodes/items which are not in the viewport are pruned and/or LOC culling. In the latter case items which are too small to be perceivable are not generated.
- +: The visualization supports further methods to filter interesting, that is changed nodes.

References

[1] D. Keim, G. Andrienko, J. Fekete, C. Görg, J. Kohlhammer, and G. Melançon, "Visual analytics: Definition, process, and challenges," *Information Visualization*, pp. 154–175, 2008.

[2] "Moore's law." [Online]. Available: http://en.wikipedia.org/wiki/Moore's_law

[3] A. S. Foundation, "Apache subversion." [Online]. Available: http://subversion.apache.org

[4] J. J. Thomas and K. Cook, "Illuminating the path: The r&d agenda for visual analytics." [Online]. Available: http://nvac.pnl.gov/agenda.stm

[5] F. J. Anscombe, "Graphs in statistical analysis." [Online]. Available: http://www.jstor.org/discover/10.2307/2682899?uid=3737864&uid=2&uid=4&sid=56171835643

[6] M. Kramis and A. Seabix, "Treetank, designing a versioned xml storage." [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-126912

[7] W3C, "Document object model." [Online]. Available: http://www.w3.org/DOM/

[8] "Xmark benchmark." [Online]. Available: http://www.xml-benchmark.org/

[9] S. Rönnau, G. Philipp, and U. Borghoff, "Efficient change control of xml documents," in *DocEng*, vol. 9, 2009, pp. 3–12.

[10] "Delta xml." [Online]. Available: http://www.deltaxml.com

[11] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 41–52.

[12] K. Tai, "The tree-to-tree correction problem," *Journal of the ACM (JACM)*, vol. 26, no. 3, pp. 422–433, 1979.

[13] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *ACM SIGMOD Record*, vol. 25, no. 2. ACM, 1996, pp. 493–504.

[14] T. Lindholm, J. Kangasharju, and S. Tarkoma, "Fast and simple xml tree differencing by sequence alignment," in *DocEng*, vol. 6, 2006, pp. 75–84.

[15] Y. Wang, D. DeWitt, and J. Cai, "X-diff: An effective change detection algorithm for xml documents," in *Data Engineering, 2003. Proceedings. 19th International Conference on*. Ieee, 2003, pp. 519–530.

[16] G. Cobéna, T. Abdessalem, and Y. Hinnach, "A comparative study for xml change detection," *Research Report, INRIA Rocquencourt, France*, 2002.

[17] R. Therón, "Hierarchical-temporal data visualization using a tree-ring metaphor," in *Smart Graphics*. Springer, 2006, pp. 70–81.

[18] S. Bremm, T. von Landesberger, M. Heß, T. Schreck, P. Weil, and K. Hamacherk, "Interactive visual comparison of multiple trees," in *Visual Analytics Science and Technology (VAST), 2011 IEEE Conference on*. IEEE, 2011, pp. 31–40.

[19] Y. Tu and H. Shen, "Visualizing changes of hierarchical data using treemaps," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 6, pp. 1286–1293, 2007.

[20] M. Bruls, K. Huizing, and J. Van Wijk, "Squarified treemaps," in *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*. Citeseer, 2000, pp. 33–42.

[21] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Z. Zhou, "Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 453–462, 2003.

[22] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.

[23] M. Ishihara, K. Misue, and J. Tanaka, "Ripple presentation for tree structures with historical information," in *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation-Volume 60*. Australian Computer Society, Inc., 2006, pp. 153–160.

[24] Logilab, "Logilab - xmldiff." [Online]. Available: http://www.logilab.org/859

[25] D. Hottinger and F. Meyer, "Xmldiff report." [Online]. Available: http://archiv.infsec.ethz.ch/education/projects/archive/XMLDiffReport.pdf

[26] S. Graf, S. K. Belle, and M. Waldvogel, "Rolling boles, optimal XML structure integrity for updating operations," in *Proceedings of the 20th international conference on World wide web*, ser. WWW '11. New York, NY, USA: ACM, 2011.

[27] S. Graf, L. Lewandowski, and M. Waldvogel, "Integrity assurance for RESTful XML," in *Proceedings of the 2010 international conference on Advances in conceptual modeling: applications and challenges*, ser. ER'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 180–189.

[28] O. D. T. Committee, "Docbook." [Online]. Available: http://www.docbook.org/

[29] Saxonica, "Saxon - xslt and xquery processor." [Online]. Available: http://www.saxonica.com/

[30] T. Kaiserslautern, "Brackit." [Online]. Available: http://code.google.com/p/brackit/

[31] D. Holten, "Hierarchical edge bundles." [Online]. Available: http://www.win.tue.nl/~dholten/papers/bundles_infovis.pdf

[32] A. Holupirek, "Filesystem markup language." [Online]. Available: https://github.com/holu/fsml

[33] D. Arnold, "Lexical functional grammar." [Online]. Available: http://www.essex.ac.uk/linguistics/external/lfg/FAQ/WhatIsLFG.html

[34] Wikimedia, "Wikipedia data dumps." [Online]. Available: http://meta.wikimedia.org/wiki/Data_dumps

[35] M. Team, "Wiki2xml." [Online]. Available: http://www.modis.ispras.ru/texterra/download/wiki2xml.zip

[36] A. S. Foundation, "Hadoop - mapreduce framework." [Online]. Available: http://hadoop.apache.org/mapreduce/