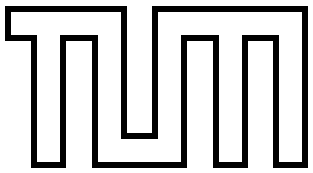


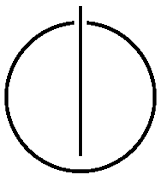
FAKULTÄT FÜR INFORMATIK

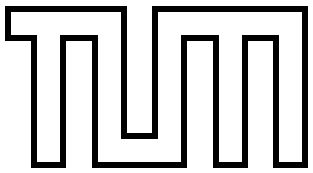
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

# Efficient Modelling of the Evolution of Hierarchical Data

Martin Raiber





# FAKULTÄT FÜR INFORMATIK

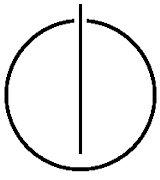
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

# Efficient Modelling of the Evolution of Hierarchical Data

# Effiziente Modellierung der Evolution von Hierarchischen Daten


| | |
|---|---|
| Author: | Martin Raiber |
| Supervisor: | Prof. Alfons Kemper, Ph.D. |
| Advisor: | Jan Finis, M. Sc. |
| Date: | November 30, 2012 |

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, November 30th, 2012

. . . . . .

*(Signature of candidate)*

# Acknowledgments

First and foremost I would like to thank my advisor Jan Finis. His ideas were valuable and are now a fundamental part of this thesis. The input he gave me was epic in proportion and probably increased the quality of this thesis by orders of magnitudes.

I would also like to thank my supervisor Prof. Kemper for allowing me to pursue this underrepresented field of study.

Last but not least, I would like to thank my support structure, friends and family, for putting up with me.

# Abstract

Knowledge is increasingly stored and exchanged via hierarchical data formats, e.g, via the *Extensible Markup Language* (XML). This knowledge is often regularly updated, and thus it is beneficial if one can efficiently model, find and express those changes in order to exchange them, to visualize them, or to reconcile them with other changes.

In this thesis, change operations on hierarchical data and the costs of those operations are defined, and a method is described which efficiently finds differences between hierarchical data, i.e. trees, and expresses them using the change operations in an approximately cost-minimal edit script. To support a wide range of applications, the method works both with hierarchical data where the order of elements is important and with hierarchical data where it is not important. It also supports a wide range of change operations, including copying or moving of whole sub-trees.

Extending existing methods, the cost of the edit scripts is reduced by first building feature vectors of sub-trees, saving them in index structures and then using them to find similar sub-trees rapidly. This does discover cost-reducing similarities which other methods miss and thus decreases the cost of the edit scripts, generated by the method developed in this thesis.

By comparing and evaluating the performance of several different index structures and by benchmarks with artificial and natural hierarchical data, the approach was optimized for speed and quality. Comparisons with other methods using the same operations show that the developed method generates edit scripts with less cost for most of the test data while having comparative runtime. The approach presented in thesis is therefore both better and more flexible than other comparable approaches.

# Zusammenfassung

Immer mehr Wissen wird in hierarchischen Datenformaten gespeichert und ausgetauscht, zum Beispiel mit XML. Da sich dieses Wissen oft ändert ist es vorteilhaft, wenn man ein Modell für diese Änderungen hat und sie effizient auffinden und ausdrücken kann, um sie auszutauschen, zu visualisieren oder um sie mit anderen Änderungen zusammenzuführen.

In dieser Arbeit werden eben solche Änderungsoperationen auf hierarchische Daten und deren Kosten definiert, um dann eine Methode vorzustellen, die effizient Änderungen zwischen verschiedenen Versionen hierarchischer Daten (Bäume) findet und diese dann mit der Hilfe der Änderungsoperationen ungefähr Kostenminimal in einer Liste von Operationen auszudrücken. Um ein weites Feld von Anwendungen zu unterstützen, funktioniert diese Methode mit Bäumen, deren Knotenreihenfolge wichtig ist, ebenso wie mit Bäumen, bei denen sie unwichtig ist. Die Methode unterstützt auch eine große Anzahl an Änderungsoperationen, unter anderem das Kopieren und Bewegen ganzer Teilbäume.

Aufbauend auf existierende Ansätze, reduziert die in dieser Arbeit vorgestellte Methode die Kosten der Liste von Operationen, indem ein Ansatz verwendet wird, der Eigenschaftsvektoren von Teilbäumen berechnet, diese in schnelle Indexstrukturen speichert, um dann schnell ähnliche Teilbäume zu finden. Ähnliche Baumstrukturen, die andere Ansätze nicht finden, werden so entdeckt. Dies ermöglicht die Änderungen zwischen den Versionen der hierarchischen Daten mit einer kleineren Menge von Operationen auszudrücken.

Durch den Vergleich von Performanz und Qualität verschiedener Indexstrukturen für die Ähnlichkeitssuche und durch Benchmarks mit synthetischen und natürlichen Daten wurde die Qualität und Geschwindigkeit der Methode optimiert. Vergleiche mit anderen Ansätzen, die das gleiche Ergebnis verfolgen, zeigen dann, dass die entwickelte Methode in den meisten Fällen Listen von Operationen mit geringeren Kosten erzeugt, während die Laufzeit vergleichbar bleibt. Der in dieser Arbeit präsentierte Ansatz ist somit besser und flexibler als andere, vergleichbare Ansätze.

# Contents

# 1 Introduction

Data may be organized with a varying degree of complexity. This Thesis, for example, is organized into chapters, sections and paragraphs and thus in a hierarchical fashion. This is better than using a less complex way, say a flat way, of organizing and presenting knowledge, as this knowledge is inherently hierarchical and not presenting it in a hierarchical fashion would remove this aspect. Thus for text documents a hierarchical way of organization seems most appropriate, as in many other areas, be it HTML and semantic web pages, XML or file systems. Having such a hierarchical knowledge representation one may ask the question about how this hierarchical knowledge changed over time and how this change can be efficiently obtained and modeled, given only snapshots of the data. Being able to do so would result in an efficient way of storing, using and analysing this evolution, without introducing change monitoring, which may be costly or impossible a posteriori.

Future search engines, for example, will probably increasingly work with hierarchical data. Even today, Google analyzes where in the HTML tree text is and uses this to gauge the importance of text fragments. If a hierarchical change detection algorithm such as what is developed in this thesis, is used, only changes between versions have to be analyzed and fed into the search engine's database. The search engine has an internal database, optimized for supporting fast search queries, in which it stores current versions of documents. If a document changes, it needs to change the copy plus the index structure for the searching. Namely it needs to know what it has to add, what to remove and what to move somewhere else in its internal database. Since changes to this database are costly, it is beneficial to have a pre-processing step which minimizes the required number of changes to update the copy. A schematic of such a process can be seen in Figure 1.1.

Detecting changes becomes difficult if a large number of changes accumulate between versions and if there is no change tracking mechanism. Both circumstances hold in the website example and many other cases.

Change monitoring is a more straight-forward application for a method which models and finds changes between versions of hierarchical data. Being able to view and highlight changes between versions of hierarchical configuration files or documents is a problem every change or content management software needs to solve. Often this is done using line based differencing algorithms, which give less than optimal results for hierarchical data. For example, in Figure 1.2, the line based diff is wholly disconnected from what the user did and thus not helpful if those changes are inspected. Using algorithms specialized for hierarchical data produces more meaningful results and often more accurately describes the changes that occurred. Another scenario where standard line based differencing tools fail is with orderless data, when the hierarchical data does not represent a document any more. Since

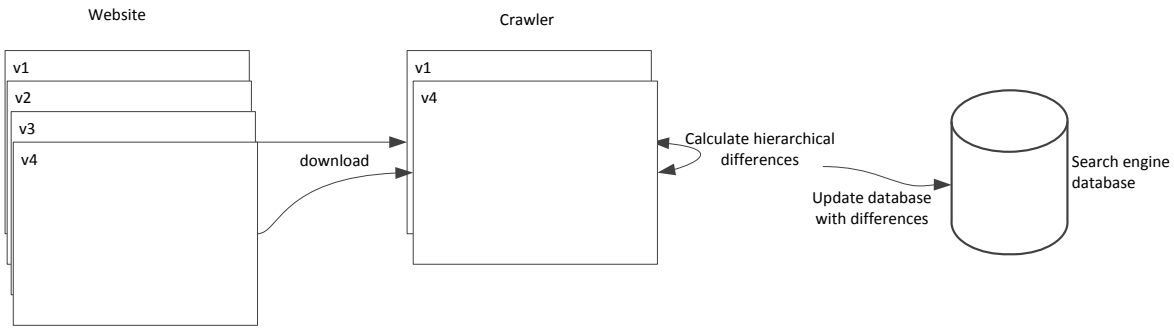



Figure 1.1: Being able to efficiently identify changes in documents the minimizes number of changes in further processing stages, e.g., of a website crawler.

the order of elements is not important, order changes should not be reported. Line based tools do not know about the hierarchical structure at all and thus report them.

Depending on what the hierarchical data represents further restrictions may apply: For example, there may be elements with the same name in XML but not in file systems. There may also be links, again in file systems between directories or the XML XLink, transforming the former tree into a graph. Often these properties are not only defined by the storage and exchange format, but by the concrete application. Therefore, how the changes of this hierarchical knowledge should be modeled depend on the application and vary significantly: One application may, for example, only allow deletion and insertion of single, new elements. Other applications may allow deleting whole sub-trees at once without additional cost. They may even be able to move or even copy sub-trees efficiently. This wide range of possibilities implies that any method which tries to solve this problem in general should be very flexible to fit to the internal data model of the application in which it will be used.

Another obstacle in applications of such a method is the amount of data it has to handle. Often it is not really necessary that the minimal amount of required actions to transform one set of hierarchical data into another is found, but that this search for such actions finishes in a reasonable, predictable amount of time. Looking at the potential applications, a linear runtime seems reasonable.

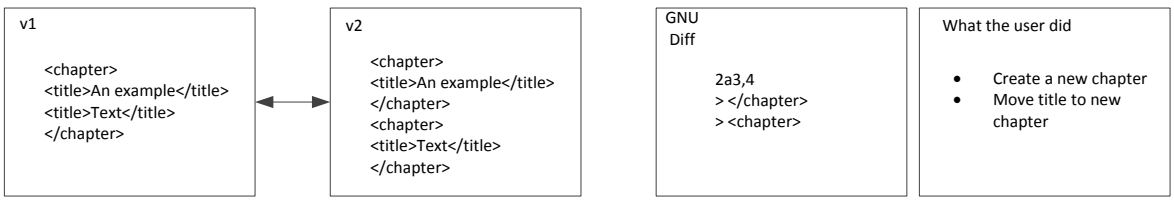



Figure 1.2: Differencing with a flat representation of hierarchical data is sometimes not meaningful.

This work introduces a novel method to calculate an approximately minimal list of actions to transform one tree into another in linear time. It will be configurable if the data order should be considered important or not. The set of actions the method can generate are configurable as well, such that one may allow the algorithm to move sub-trees but not to copy them. Finally, the method will work on any kind of hierarchical data given that it is supplied in a supported data format such as XML, HTML or JSON.

## 1.1 Representing Changes

There are several ways to represent the changes in hierarchical data. They differ in their usefulness in different application scenarios and their intuitiveness for human readers, as well as how succinct they represent changes. There are three general ways to represent the changes:

1. Show what did not change and is the same in both trees.
2. Show only what changed.
3. Define different operations on hierarchical data and represent the changes by means of those operations.

If the changes are saved, e.g., in a file, we can then either save this information by extending the existing data with change data, or by saving the change information separately. Figure 1.3 shows two such representations.
There is no established standard or common practice, of how the changes should be represented. We decided on representing them by operations, separate from the hierarchical data in our method. This is most flexible as new operations can be easily added, and comprehensible as we regularly perform those operations, e.g., when we rename, move or delete folders. Most importantly the operations the method produces can actually represent what the user did on the hierarchical data. Representing the changes separately was chosen because this results in "tree patches", i.e., running the method on two trees $A$ and $B$ gives us a set of operations $D$, which applied to $A$ transforms the tree to $B$. Then, instead of storing both tree $A$ and tree

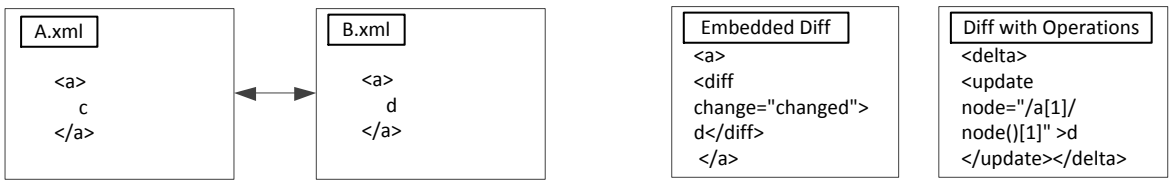



Figure 1.3: Different change representations. The left one extends the existing tree with the change information. The right one represents the changes via tree edit operations.

*B*, it is enough to store *A* and *D*. If the trees are not entirely different, less data is stored.
Supporting the other change representations would not be much of a problem. Actually, in the method presented in this thesis we first calculate which nodes are the same, and then need to invest additional work to transform this information into a set of operations. Representing the operations in the trees is as straightforward as applying them to tree *A* while modifying *A* in a way that makes clear what has been modified and how (e.g., like in Figure 1.3).

## 1.2 Applications

Computing changes directly on hierarchical data instead of on a flat representation has a wide range of applications. First, differencing and merging in revision control on hierarchical data produces fewer differences if the algorithm takes into account the hierarchical structure, compared to algorithms which dp mpt take the hierarchy into account. Having to inspect fewer changes is one obvious advantage. This kind of revision control system has itself a wide range of applications. For example, code can be represented as a tree [7], making diffing in traditional revision control systems for code in certain programming languages better and smarter.
As most documents nowadays are XML documents and thus are already in tree-form, changes in documents can be adequately modeled, calculated and displayed. This includes documents in HTML or XML.
As a personal anecdote the author once tried to give an administrator of a CMS system an automatically modified version of an XML document with only small changes. The automated step sorted the attributes of the XML document alphabetically. The difference view in the CMS system considered each reordering as a change and the administrator then refused to integrate the new version because he could not see what actually changed. In this case a smarter differencing tool would have been helpful.

It can also be useful for versioning of hierarchical data. By calculating a tree edit script between two snapshots, one snapshot can be removed and the changes saved more efficiently (similar to, e.g., [24, 8]). The change detection and correction method XyDiff [31] is, for example, used in the Xylene [1] XML data warehouse. Calculating the differences on the hierarchical data enables efficient storage, temporal queries and change control with change subscriptions.

A change detection and correction algorithm also enables an efficient synchronization of distributed hierarchical data, where bandwidth is critical, e.g., with mobile devices. Lindholm et al. describe a method [27], based on a tree edit script generating algorithm, which reconciles hierarchical data, i.e., data which is out of sync and has to be merged. This could be of broad interest for peer-to-peer distributed hierarchical databases.

Many more applications have been considered: Song et al. [45] and Hedeler et al. [17] calculate tree edit scripts for genomic and proteomic data. Zhang et al. [51] represent RNA structures as trees and calculate their distance. Images can be repre-

sented as trees, then the change detection and correction algorithm can be used in image analysis [6].

The following sections will show some use cases for calculating operations, which transform hierarchical data into one another.

### 1.2.1 Importing XML Data into Databases

Business data is often exchanged in the form of XML. This data is often imported into hierarchical databases or relational databases. Let's say we have XML data describing the products of a company as in Listing 1. This XML file is generated hourly by some internal system the company uses, but the company does not release new products or change prices that often. A contractor gets the task of building a web shop for the company selling the products. To generate the product pages they have to import this XML file. They are used to relational databases and therefore parse the XML file and put the products into a table. The Table 1.1 shows the database table. The costumer wants product changes to show up in the web shop within the

```
<?xml version="1.0" encoding="UTF-8"?>
<Company name="Imaginary Products Co. KG">
      <Products>
            <Product>
                  <Name>Space Elevator</Name>
                  <Price>12.99</Price>
                  <Available>true</Available>
            </Product>
            <Product>
                  <Name>Teleportation Device</name>
                  <Price>39.99</Price>
                  <Available>true</Available>
            </Product>
            [...]
      </Products>
</Company>
```

Listing 1: Example XML data an application may need to import

hour. The first solution they come up with is to hourly delete all rows in the table

| CompanyID | ProductID | Name | Price | Available |
|---|---|---|---|---|
| 1 | 1 | Space Elevator | 12.99 | 1 |
| 1 | 2 | Teleportation Device | 39.99 | 1 |
| .. | .. | .. | .. | .. |

Table 1.1: Table for the products in Listing 1

and then to reinsert the products from the XML. They do this in a transaction such that all other simultaneous processes can continue using the product table. After some time, the company has a lot more products. Because the web shop supports searches for names and price ranges, they had to add indices to the *Name* and *Price* column. Deleting and reinserting everything seems like a terrible idea now, especially since the data does not change that often. If it does change, it is in the order of changing the price of or adding one product.

Instead, one could save the old imported XML file and compare the new XML file with the old one. With the method developed in this thesis, one could then extract the changes between versions. Then one could read the changes and replay them on the internal data model. If, for example, the price of *Space Elevator* changes from $12.99$ to $13.99$, the change detection algorithm would output something like */Company/Products/Product[1]/Price/node()[1] changed to 13.99*. This would be easy to translate to an SQL update statement. The index structures would only be updated in parts which actually change. Of course the changes could also be detected manually in this case, but for more complex examples this becomes difficult – and a tool which solves this change detection problem in general more beneficial.

Newer relational database systems can directly handle XML data and can also create indexes on fields. A method which can detect changes between versions of XML data could therefore be directly integrated into a database system, such that changes to the indices are kept minimal.

### 1.2.2 Remotely Mirroring Filesystems

Another use case is using the change detection algorithm on file trees for bandwidth saving purposes when remotely mirroring filesystems, as may be the case for backups or file synchronization. Let $C$ be a client whose file system we want to synchronize with the server $S$. The synchronization process may work like this:

- $C$ sends its current file system tree to $S$
- $S$ looks at the differences between the snapshot of the file tree it currently has saved for $C$ and the new tree it received.
- $S$ loads all new and changed files from $C$ and modifies the file tree on $S$ such that both trees are identical. Therefore, It creates an up-to-date snapshot of $C$'s file system tree.

The problem appears if we want to avoid loading files from $C$ and do not have unique identifiers for files, such as a hash of their content (calculating the hashes is costly – it involves reading all the file contents). The otherwise available information like file name, size and last modification time are in general not enough to identify files uniquely. As an additional factor, we can, however, take into account the tree structure around the changed files. This is exactly what the method introduced in this thesis does. It finds changes in hierarchical data, taking into account

the tree structure of the problem and produces exactly what $S$ would need – namely a list of operations to transform the old snapshot for $C$ to the new one.

This can be seen in Figure 1.4, where the file tree on the server $S$ is different from the file tree on the client $C$. Several files and folders on $C$ have been added, renamed, deleted or moved. After receiving the file tree from the client, the server can use the change detection and correction method developed in this thesis to calculate the file tree differences. The server then has to execute only a few operations to update the file tree for $C$, such that the file tree on the server is identical to the file tree on the client.

### 1.2.3 Visualizing Changes on Hierarchical Data

The change detection and correction method tries to find a minimal list of operation to transform one version of hierarchical data into another. This list of operations, together with the affected locations in the hierarchical data can be shown to users, if they want to see what changed between the versions. This can be useful where changes to data or documents have to be reviewed, or where users simply want to see changes to past versions. The changes can either be shown in a linearization of the hierarchical data such as XML on a side-by-side basis or in an interactive tree structure. One popular tool to do this side-by-side comparison is DeltaXML [30]. A wide range of applications and companies using the product is listed on the website.

DeltaXML does not detect nodes moved in the hierarchy, probably because it would be difficult to show how the node moved in a side-by-side view of XML files. For that purpose we have developed an application which visualizes hierarchal data and allows a step-by-step replaying of changes between two versions of hierarchical data. A screenshot of this application is shown in Figure 1.5. It shows an edit script, and the tree the edit script is operating on.

### 1.2.4 Reconciliating Distributed Hierarchical Data

Another use case is the reconciliation of distributed and simultaneously modified hierarchical data. If, for example, the same document is changed simultaneously by two clients, a common task is to merge the resulting documents such that the changes of both clients are in a new common document. A differencing algorithm working on hierarchical data is a basic building block towards such a merge/reconciliation method. An example for how this could work on a basic document is shown in Figure 1.6. There a document is distributed to two clients, which modify it in a way that a normal line based merge tool could not handle without user intervention. Taking into account the hierarchical structure of the documents increases the number of merge cases which can be handled automatically.

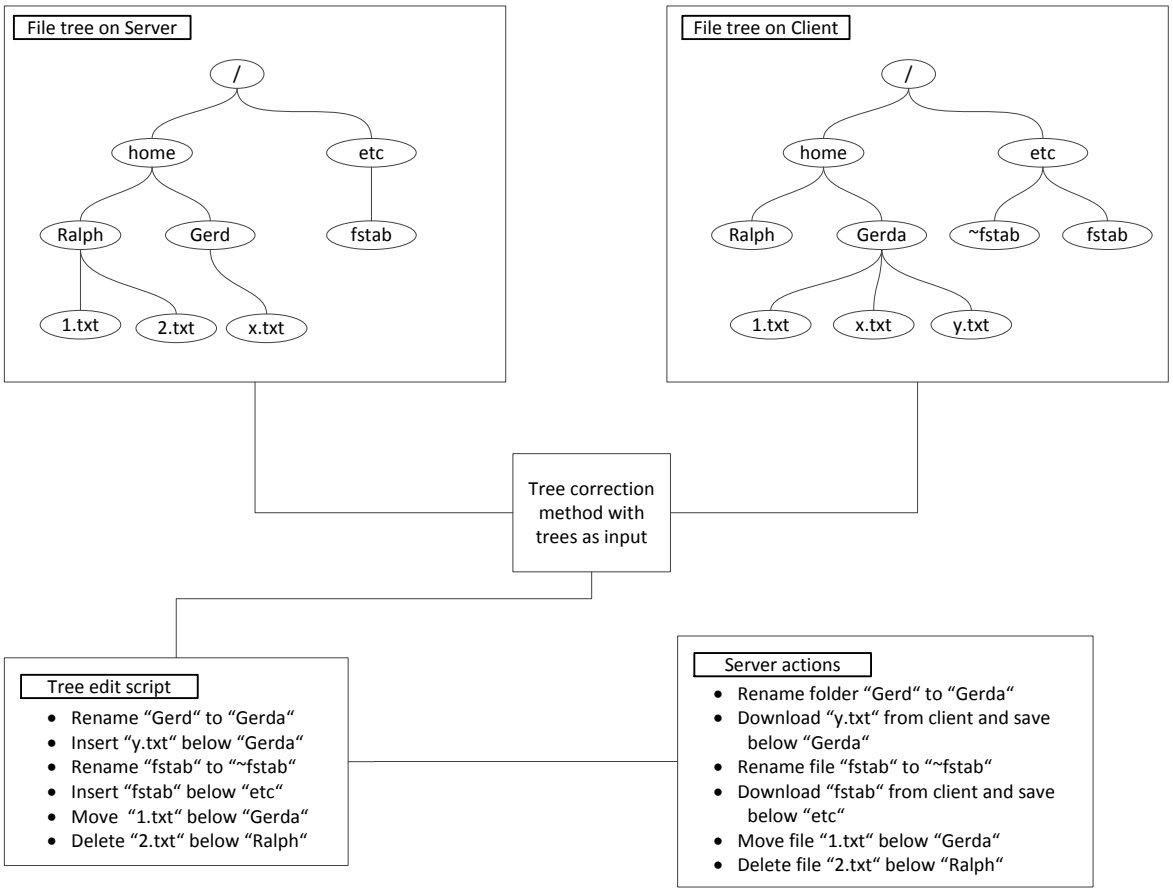



Figure 1.4: Synchronizing a file tree to a server. By using a change detection and correction method changes to the tree on the server can be kept small.

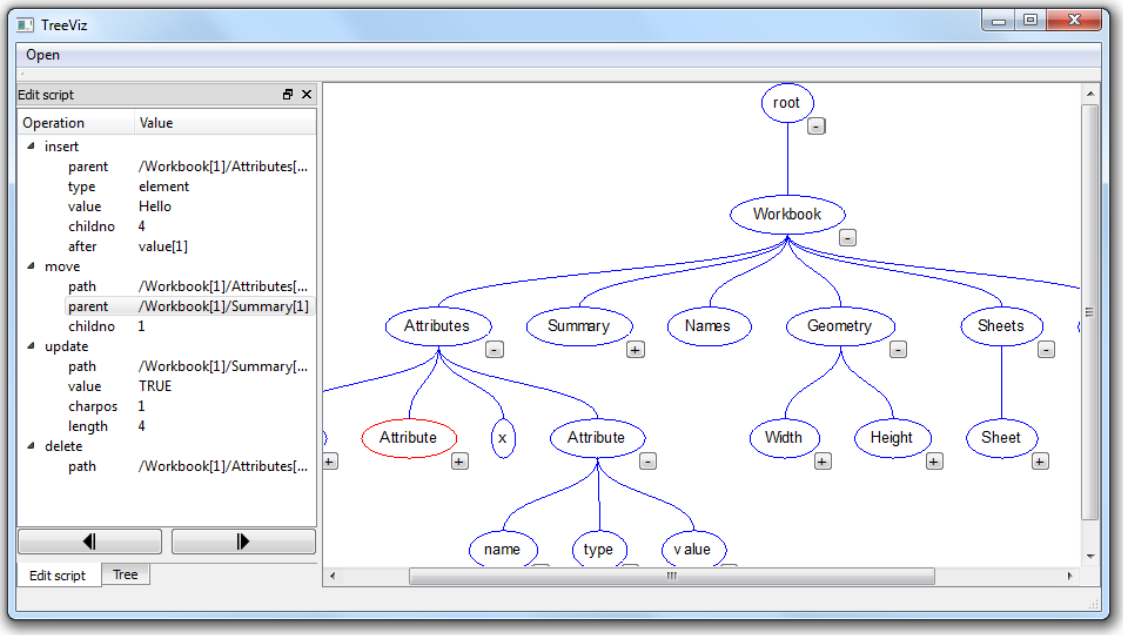



Figure 1.5: Screenshot of the change visualization tool showing an edit script and the tree it is operating on.

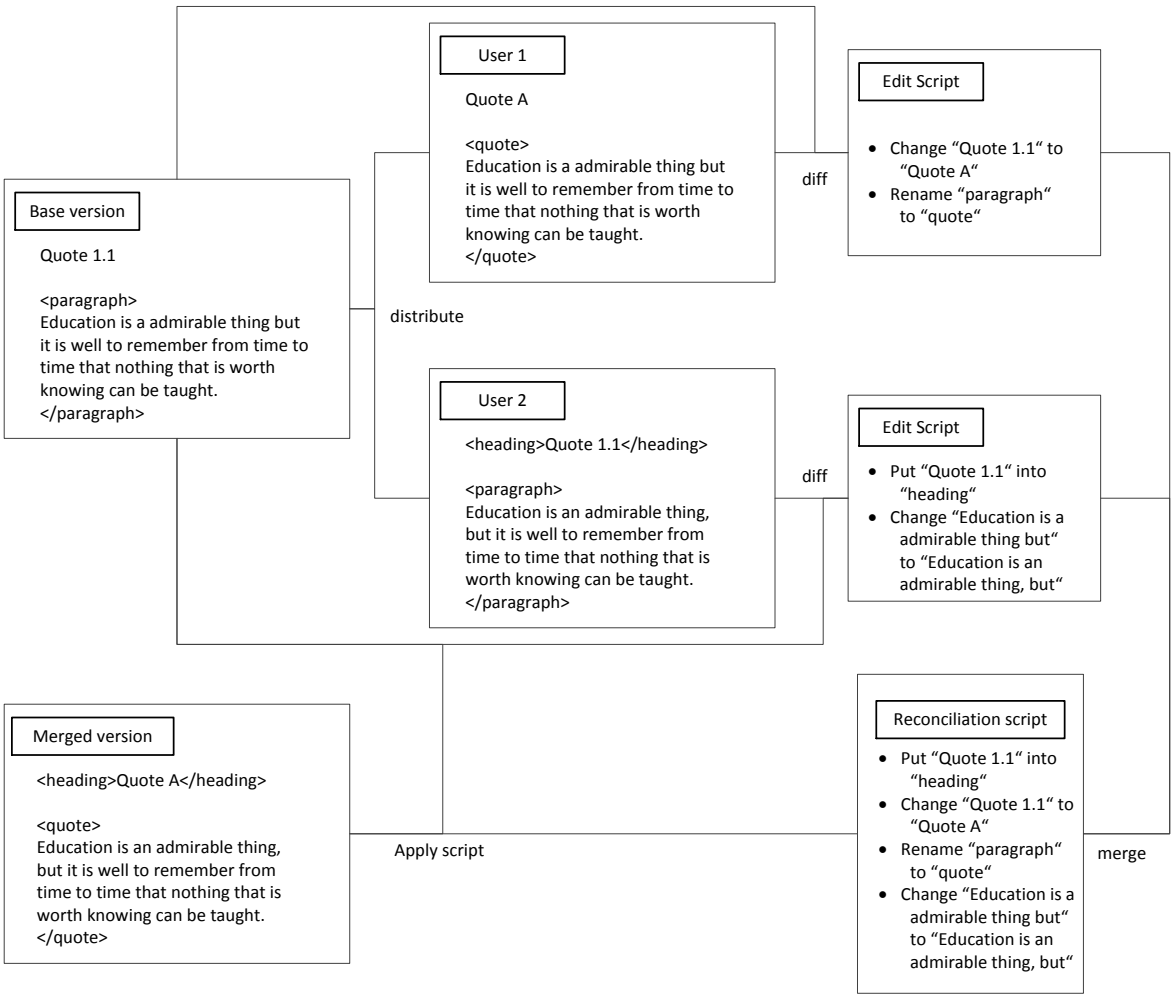



Figure 1.6: Example flow for the reconciliation of a distributed hierarchical document with the help of a hierarchical change detection and correction method after it was simultaneously modified.

# 2 Preliminaries

## 2.1 Tree

A tree $T$ is an undirected, acyclic, connected graph consisting of nodes $V(T)$ and edges $E(T)$ connecting them. Additionally, the a tree commonly has a root $root(T)$, which is one designated node of the graph, and which gives the nodes in the tree a hierarchical relationship. The distance between two nodes $n_1 \in V(T)$ and $n_2 \in V(T)$ is written as $dist(n_1, n_2)$ and is the length of the path between the two nodes in the tree. The distance of each node $n \in V(T)$ to the root is the node's level ($level(n) = dist(root(T), n)$) in the hierarchy. A tree's depth is the maximum level ($depth(T) = \max_{n \in V(T)} level(n)$). The parent ($parent(n)$) of a node is the next node on the path to the root, and a node's children ($children(n)$) are all other directly connected nodes. A trees degree is the maximum number of children of all nodes ($degree(T) = \max_{n \in V(T)} |children(n)|$). The siblings of a node are those nodes that have the same parent. Even if sibling order is not important as per data model, siblings still have an implicit order, i.e., we know which node is the leftmost sibling and can use a child index – often also called child rank – to address such a node. $child(n, 0)$ would give us the first child of a node, $child(n, 1)$ the second, and so on. $child\_rank(child(n, 0))$ does then give us the child index 0, because $child(n, 0)$ is the first child of $n$. This can be seen Figure 2.1, which shows how every node with parent can be addressed via child rank and parent. Given a node $n$ we can also find its left and right sibling ($prev\_sibling(n)$ and $next\_sibling(n)$), given they exist. A node without children is called a leaf node and a node with children an inner node. Every node $n$ is a root of a smaller sub-tree (we often say sub-tree rooted in $n$). The sub-tree consists of all nodes connected to $n$ after we remove the connection between $n$ and its parent. If a node $p$ is in this sub-tree, it is a descendant of $n$ and $n$ is an ancestor of $p$. By this definition a node is an ancestor and descendant of itself. For situations excluding the node, we use the terms proper ancestor and proper descendant. The least common ancestor of two nodes $n_1 \in V(T)$ and $n_2 \in V(T)$ is the node with the lowest level on the path from $n_1$ to $n_2$ in the tree. $lca(n_1, n_2)$ would be that least common ancestor.

The nodes in Figure 2.1 also have labels. Let $n \in V(T)$ be a node in the tree, then $l(n)$ would be the label of that node. It is a sequence of bytes and can be used to name nodes or to save data in them. Additionally to the label, some applications need a type $t(n)$. The type distinguishes different classes of nodes, which have to be handled differently.

The nodes in a tree can also be addressed by a unique identifier. If the labels in a tree are guaranteed to be unique, we can use the labels as identifier as in Figure 2.1.

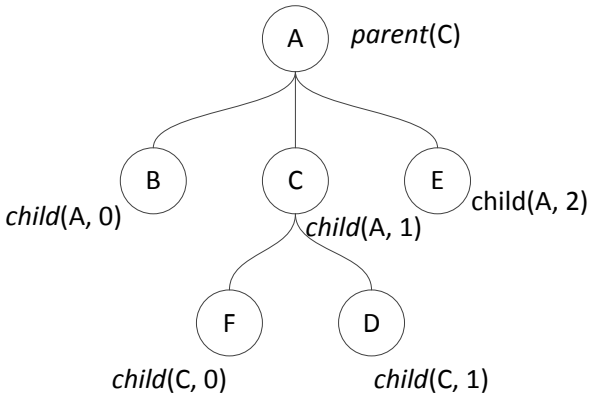



Figure 2.1: Every sibling of a node can be addressed by its parent node and child rank

As trees often do have nodes with same labels, we use pre-order numbers or XPath as addressing schemes, which unanimously identify nodes. Examples for this can be seen in Figure 2.2, where pre-order numbers are shown on the left hand side and XPaths on the right hand side. In this example addressing by label would not have worked, because there are two nodes with label "B".

As in graph theory, two trees $T_1, T_2$ are isomorphic if there exists a bijective function $f$ which maps all nodes $V(T_1)$ to $V(T_2)$, maps the root of $T_1$ to the root node of $T_2$ ($f(root(T_1)) = root(T_2)$), and for which holds:

$$\forall(x, y) \in E(T_1) : (f(x), f(y)) \in E(T_2) \tag{2.1}$$

If trees are isomorphic including child order, the mapping $f$ does not change the

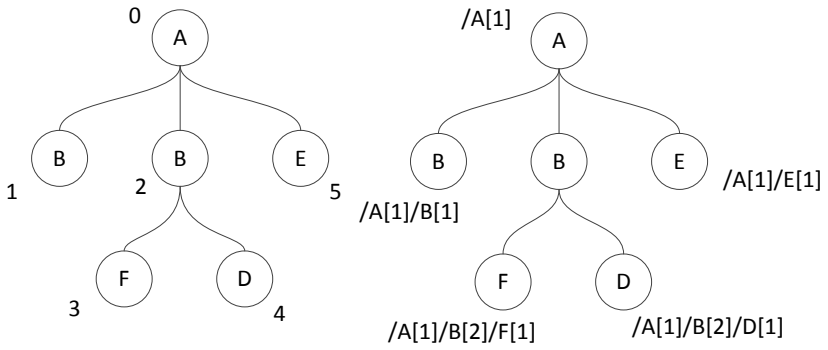



Figure 2.2: If labels are not unique nodes can be uniquely identified using for example their pre-order number or XPath

order of any children, that is Equation 2.1 holds and:

$$\begin{aligned}\forall(x,y) \in V(T_1): \quad & (\mathit{parent}(x) = \mathit{parent}(y) \wedge \mathit{child\_rank}(x) < \mathit{child\_rank}(y)) \\ & \Rightarrow (\mathit{parent}(f(x)) \neq \mathit{parent}(f(y)) \\ & \quad \vee \mathit{child\_rank}(f(x)) < \mathit{child\_rank}(f(y)))\end{aligned}$$

## 2.2 Tree Edit Scripts

The tree-to-tree correction problem or the task of calculating a minimal tree edit script between two trees is variously defined in the literature. The algorithm and its result significantly depend on the valid operations on the trees and whether the sibling order in the trees is important, i.e., the trees are ordered or unordered.

### 2.2.1 Basic Operations

An often used set of operations is $rename$, $insert$, and $remove$. Again, the specific parameters vary, but $insert$ and $remove$ are often symmetric such that an $insert$ operation can revert a $remove$. An example for the edit operations with resulting trees is given in Figure 2.3, where the operations are applied and reversed on one tree. Let $a$ be an address in an addressing scheme uniquely identifying nodes (e.g., labels, pre-order number, or XPath – unique address from now on), $l$ a label, and $s$ and $e$ integer numbers. Then the most flexible version of the basic operations can be defined as

- $rename(a, l)$ Changes the label of the node identified by $a$ to $l$.
- $insert(a, l, s, e)$ If the node identified by $a$ has children $c_0, c_1, ..., c_n$, $insert$ adds another node with label $l$ as child before $c_s$ and changes the parent of $c_s, .., c_e$ to $b$, if $e \geq s$.
- $remove(a)$. Removes node adressed by $a$ and attaches $children(a)$ to $parent(a)$.

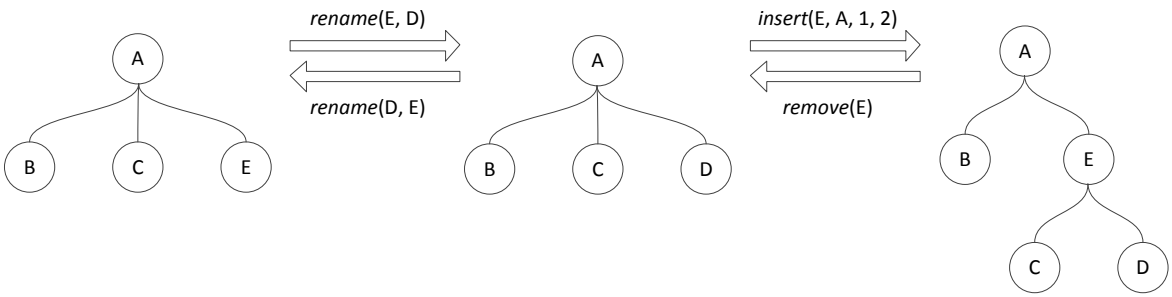



Figure 2.3: Basic edit operations on trees

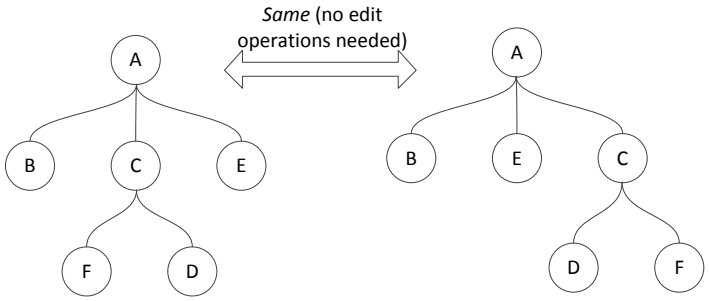



Figure 2.4: Unordered trees are equivalent and require no changes, because the sibling order is not important

### 2.2.2 Sibling Order

If the sibling order is not important, trees which would have previously required a lot of changes, may now require none at all. With the unordered model, the trees in Figure 2.4 are equivalent while the same trees with the ordered model require two corrections in Figure 2.5.

### 2.2.3 Cost Model

Sometimes we not only want to minimize the number of necessary edit operations: We want to minimize the tree edit script using a certain cost model for the operations. The model is an assignment of a certain cost to each operation. The simplest cost model is a uniform one, e.g., by assigning the cost of $1$ to each edit operation. The cost is then the number of operations. In a non-uniform cost model we assign different costs to operations. For example, we might make a $remove$ cheaper than an $insert$. If the cost does not change with the parameters of the operations, it is a fixed cost model. A non-fixed model might be interesting in scenarios where some nodes are more relevant or costlier to change than others. Most existing work uses a fixed, uniform cost model.

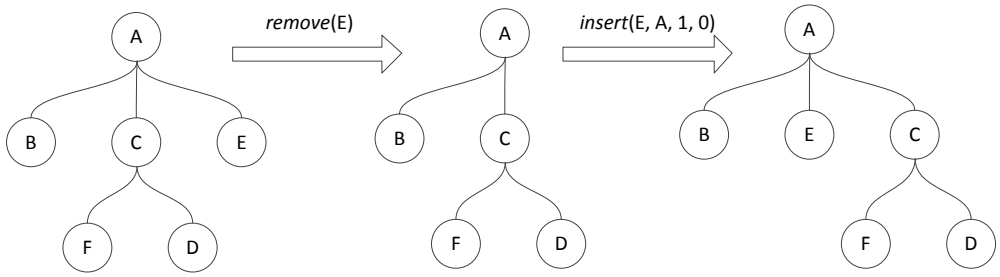



Figure 2.5: Ordered tree requires correction, because the sibling order is important

### 2.2.4 Sub-tree Operations

It is often convenient to remove or insert whole sub-trees. Either because the underlying application regards these operations as cheap, or simply because it makes the tree edit script more compact. Let $a$ be a unique address and $s$ be an integer number. Those operations are then defined as:

- $subtreeInsert(a, B, s)$ Node addressed by $a$ has children $c_0, c_1, ..., c_n$. Insert tree $B$ before $c_s$ as sub-tree. $B$ represents a whole tree, e.g., as XML.
- $subtreeRemove(a)$ Removes the node addressed by $a$ and all its descendants.

### 2.2.5 Move and Copy Operations

If for the application moving sub-trees is cheaper than removing and inserting them, those may also be used. Copying sub-trees might also be cheaper than inserting sub-trees. Let $a$ and $b$ be unique addresses and $s$ be an integer number:

- $move(a, b, s)$ Let $c_0, c_1, ..., c_n$ be the children of $a$. Move the sub-tree addressed by $b$ such that its root is before $c_s$ with node addressed by $a$ as parent.
- $copy(a, b, s)$ Let $c_0, c_1, ..., c_n$ be the children of $a$. Create a copy of the sub-tree addressed by $b$ and insert it before $c_s$ with node addressed by $a$ as parent.

## 2.3 Matching

A tree matching is a function which maps tree nodes from one tree to tree nodes of another tree. Given such a matching, we can construct a tree edit script relatively easy. The relationship between edit scripts and matchings is not unique: Given a matching we can construct different edit scripts, whereas an edit script implies a specific matching between the trees. The difference between those different edit scripts is mostly cosmetic, however. The order of *insert* and *remove* operations is, for example, not important. Therefore, since a matching is less ambiguous and because it is more intuitive to construct, it is preferable to first construct a matching and then transform this matching into an edit script instead of directly constructing an edit script. A matching for two example trees with edit script is shown in Figure 2.6. One can probably also see that generating the edit script from such a matching is relatively easy. We call the lines belonging to a matching, connecting the nodes mappings (red in Figure 2.6). Nodes, which have no mapping in $A$, are *deleted*, such as the node with label "I" in the figure. Nodes with no mapping in $B$ are *inserted*, such as the node with label "H". Nodes with their parent in $B$ not mapped to the parent of the node they are mapped to in $A$ are moved, such as the sub-tree rooted in the node with label "E" and nodes which are mapped to nodes with a different label are renamed, such as the node with label "B" on the left hand side. We discuss how to transform a matching into an edit script in detail in Section 5.2.

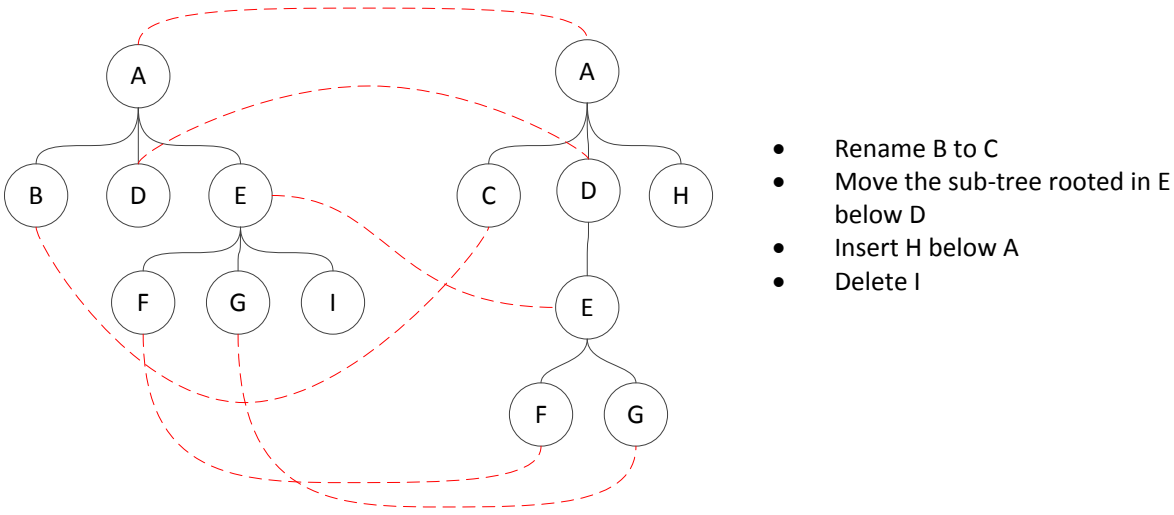



Figure 2.6: An edit script implied by a matching. The order of edit operations can be changed in this example.

Mathematically, a matching between two trees $A$ and $B$ is a function $M$ which maps a subset of nodes $S \subseteq V(B)$ to nodes in $V(A)$.

$$M : S \to V(A) \tag{2.2}$$

If every node in $A$ has maximally one node that maps to it (i.e., $M$ injective), it is implied by an edit script with *insert*, *remove*, *rename*, and *move*. If a node in $A$ has more than one node in $B$ that maps to it, the implying edit script must use a *copy* operation.

The best or optimal matchings are those implied by a minimum cost edit script. If a single mapped node in $M$ is not part of any best matching, we speak of a wrong mapping. If it is part of a best matching, it is correctly mapped. Even if all nodes are correctly mapped, the resulting edit script can still be not minimal because there can be more than one best matching. We call a mapping which is part of one best matching while the majority of matchings is part of another best matching, a disabling mapping. Obviously, as this is an optimization problem, there are different grades of wrong or disabling mappings. They may increase the cost of the edit scripts by different amounts. Since we are interested in an approximately cost-minimal edit script, we use methods that try to avoid mapping the wrong nodes to each other, but can give no guarantee that they map correctly.

## 2.4 Approximate List Matching

We have two lists of elements which are comparable $L_A = (A_1, A_2, \ldots)$ and $L_B = (B_1, B_2, \ldots)$. For the sake of simplicity let their sizes be equal: $|L_A| = |L_B| = n$. Then there is a preference matrix $P \in \mathbb{R}^{n \times n}$ which tells us how good mapping a

single element to another may be, e.g., $P_{A_1,B_1}$ returns how good mapping $A_1$ and $B_1$ would be. Numbers are smaller for a better mapping, zero denotes a perfect mapping. Let $M$ be all functions which map elements from $L_B$ to elements in $L_A$. For an optimal matching $m \in M$, the sum of all preferences should be minimal ($\arg\min_x f(x) := \{x | \forall y : f(y) \geq f(x)\}$):

$$M_0 = \arg\min_m \sum_{i=1}^{n} P_{m(B_i),B_i} \tag{2.3}$$

In literature this problem is a variant of the marriage matching problem [15]. The full marriage matching problem has two preference matrices one for how much elements in $L_A$ (men) prefer elements in $L_B$ (women) and one for the other way around. Here, we have only one, i.e., the preferences are considered to be symmetrical. The name suggests one application: Matching men to women for marriage with globally stable outcome. More useful applications exist for example in economics where, e.g., resources are efficiently allocated to consumers. In fact, the 2012 Nobel prize in economics was won for an application of the stable marriage problem. In this thesis it will be used to map sub-trees.

In the simplified case with only one preference matrix, the optimal solution can be found by enumerating all possible matchings and selecting the best. This has an often prohibitive runtime in $\mathcal{O}(n!)$. If we are satisfied with a heuristic solution, we can find a matching using a greedy method: We sort the preference matrix such that we can iterate over it getting the best mappings first.

Figure 2.7 shows an example for this. While iterating over the sorted result, we map the two elements to each other if both of them are not mapped. In the example, we first get $(A_1, B_2)$ and map them. Then we get $(A_1, B_3)$, but we cannot map, because $A_1$ is already mapped to $B_2$. Next would be $(A_3, B_1)$ which we again map to each other.

This greedy matching method requires us to sort the preference matrix and thus runs in $\mathcal{O}(n^2 \log n)$. If we want a heuristic method with non-quadratic runtime, we have to avoid looking at all the $\mathcal{O}(n^2)$ entries of the preference matrix. Let us select a threshold $t$ below which mapping two elements is "good enough". For every element in $L_B$ we look at a constant amount $c$ of random unmapped elements in $L_A$. Let $A_i$ and $B_j$ be the current considered mapping candidates. If $P(A_i, B_j) < t$, we map the two elements. The runtime complexity of this method is $\mathcal{O}(n)$. The resulting matching can be, however, very bad. Contrary to the methods before this marching method does also lead to outcomes where elements remain unmapped. We call it opportunistic matching.
It is used in practice: Since the lifetime of a human is limited, he can only look at a limited amount of unmarried partners and marries the partner he is looking at if the preference is below $t$. This leads to inefficiencies such as divorces if he later on finds a better partner. The greedy solution would be much more efficient[1].

[1]This judgement assumes that the amount of resources spend on computational power for a better allocation of men to women is less than the amount of resources spent on divorces and the loss of resources (e.g., labour) created by unhappy marriages and thus would be overall more efficient.

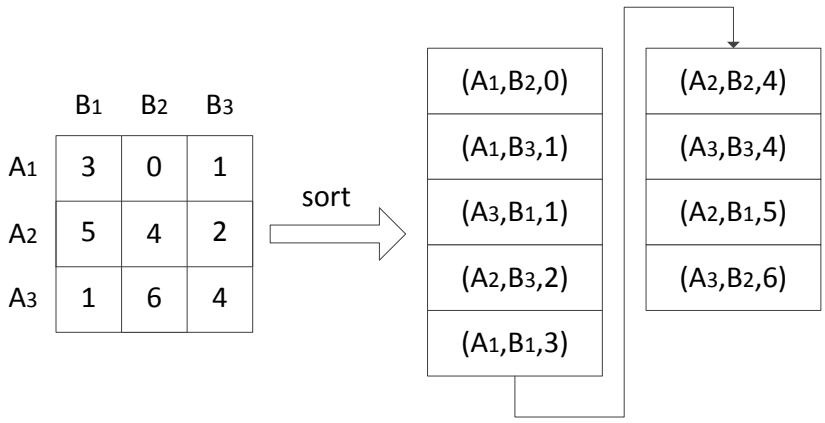



Figure 2.7: Example for how the preference matrix is sorted into a list such that the best mappings come first.

### 2.4.1 Top-down Matching

We have two trees $A$ and $B$ that have no two siblings with the same label and the same parent. This is, for example, the case in file systems, where one cannot create files or folders with the same name, given they are in the same folder. Then, a matching method would be to start at the root and to map nodes with the same label and type[2] to each other. If a node is mapped and is not a leaf node, the same matching method is recursively applied to its children and so on. When using this method, if a node's label is changed, the whole sub-tree starting at this node is not mapped anymore, such as in Figure 2.8 where no node in the sub-tree rooted in the node with label "E" on the left hand side is mapped, because the label changed.

While this is an obvious disadvantage, this does not happen too often in most cases: Nodes relatively close to the root node are often changed infrequently. This

[2]In the filesystem case there are two types: Files and folders.

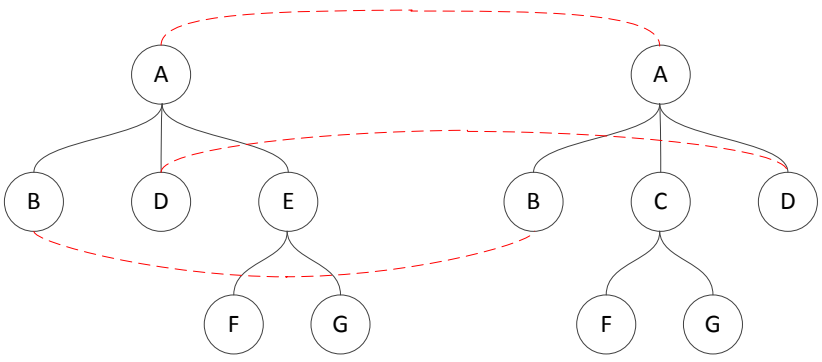



Figure 2.8: Example top-down matching, where a sub-tree is not mapped because a label changed.

pattern may even be an imposed constraint: If we look at file systems, changing folder names close to the root would invalidate many file system links and configurations and, as such, it is done only reluctantly. Similarly with XML data exchange, where modifying labels of leaf nodes is often supported by all involved applications, contrary to modifying element names in the upper levels of the hierarchy. This assumption – that the closer the nodes to the root, the more infrequently they are changed – may, however, not hold if presentation and data are not well separated. This is, for example, the case with HTML. If positioning an article or changing the layout necessitates an additional div container, such a straightforward approach is not able to map it anymore.

If we allow siblings with equal labels and same parent, the top-down method runs into some problems. Before, the label was an indication that nodes had similar sub-trees. This assumption is weaker now: Let's say we have two siblings with the same label. The algorithm maps them by looking at the order. As long as we do not change the order, everything is satisfactory. If we change the order the wrong sub-trees are mapped to each other, which causes larger edit scripts – see Figure 2.9, where the order of the node with label "D" is changed, which causes an opportunistic top-down matching, which maps to the first node with the same label, to incorrectly map the nodes. We have three possible remedies for such a situation:

1. Do not map nodes which have siblings with same names in either version of the tree to each other.

2. Map them only to each other if all nodes in both sub-trees are mapped to each other afterward.

3. Consider a mapping to all siblings with the same label and then map to the one with most sub-tree mappings.

While possibility two would work in Figure 2.9, it would fail if there were any changes. Possibility three does, however, result in a unusually high runtime complexity, as every sibling has to be compared to each sibling with the same name

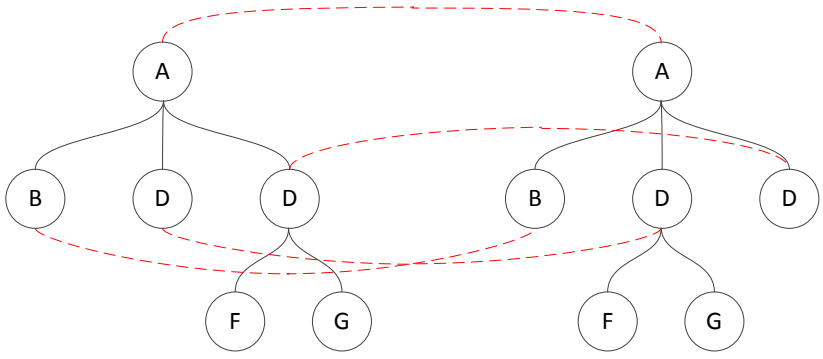



Figure 2.9: Example top-down matching, where nodes with same label are mapped incorrectly.

– potentially $n$ of those. The complexity when resolving this using opportunistic matching would then be $\mathcal{O}(n)$ – with the result potentially harmful, for greedy matching $\mathcal{O}(n^2 \log n)$. We use possibility one for our approach and take care of the remaining unmapped sub-trees in other matching steps. Let $A$ and $B$ be two trees. Let $M : V(B) \to V(A)$ be the produced matching which maps nodes from $B$ to $A$. The top-down matching algorithm which handles cases where the matching $M$ is already partially populated by other matching methods then works like this:

**Algorithm 2.1.** *Top-down matching*

1. *Set $a = root(A)$ and $b = root(B)$*
2. *Iterate over the sorted list (lexicographically by label) of children of $a$ and $b$. Let $c_a$ be the current child of $a$ and $c_b$ the current child of $b$.*
   - *If $c_a$ is already mapped ($M^{-1}(c_a)$ is set), advance $c_a$.*
   - *If $c_b$ is already mapped to a node with parent $a$ ($parent(M(c_b)) = a$), recursivly go to 2. with $a = M(c_b)$ and $b = c_b$. Advance $c_b$.*
   - *If $c_b$ is mapped to a node with $parent(M(c_b)) \neq a$, advance $c_b$.*
   - *If the labels and type of $c_a$ and $c_b$ are the same ($l(c_a) = l(c_b)$ and $t(c_a) = t(c_b)$) and if there is only one child of $a$ with label $l(c_a)$ and type $t(c_a)$ and one child of node $b$ with label $l(c_b)$ and type $t(c_b)$, map the two nodes and recursively go to 2. with $a = c_a$ and $b = c_b$. Advance $c_a$ and $c_b$.*
   - *If the label of $c_a$ is lexicographically smaller than the label of $c_b$ ($l(c_a) < l(c_b)$), advance $c_a$.*
   - *If the label of $c_b$ is lexicographically smaller than the label of $c_a$ ($l(c_b) < l(c_a)$), advance $c_b$.*

It can be implemented more efficiently using hash tables because then the runtime-dominating sorting can be avoided. It would consist of a step where we put the children of $a$ into a hash table with the label as key and a step where we iterate over children in $b$, looking up the current child by label in the hash table and mapping it to the child of $a$ if found there. We used variant without hash table because some applications, such as the filesystem example, exhibit trees that have already sorted siblings.

### 2.4.2 Bottom-up Matching

Bottom up matching methods first match leaf nodes and then consider other nodes higher up in the hierarchy. Contrary to the top-down matching, where a node can only be matched if all its ancestors have been matched, a bottom-up matching can only match a node if all its descendants have been matched.

**Definition 1** (Bottom-up Matching)**.** *In a bottom-up matching, for tree $A$ and $B$, if node $a$ is mapped to node $b$, all children $c_{a1}$, $c_{a2}$,... of $a$ are mapped to children $c_{b1}$,*

$c_{b2}$, ... of $b$. *In an unordered bottom-up matching any child of node $a$ can be mapped to any child of node $b$. In an ordered bottom-up matching children of $a$ have to be mapped to children of $b$ with the same child rank.*

This matching is very sensitive to changes in the lower levels of the hierarchy: One change in a node prevents all its ancestors from being mapped. That is, if a leaf node label is changed, all ancestors of that leaf nodes cannot be mapped anymore. This can be seen in Figure 2.10 where the node with label "E" on the left hand side is changed, causing both the nodes with label "A" and "C" to be unmapped.
Let $A$ and $B$ be the trees we want to match and $h$ be a hash table, which maps labels to a set of nodes. Let all nodes in tree $B$ be unmarked. Additionally, we can save a set of mapping candidate nodes for every node in $B$. For an ordered matching, a simple algorithm producing such a matching, which first produces mapping candidates for nodes in $B$ and then selects the final matching from those candidates works like this:

**Algorithm 2.2.** *Top-down matching*

1. *Set $n = 0$.*
2. *Collect nodes having maximal distance $n$ from any leaf node in tree $B$ and put them into hash table $h$, keyed by their label. Nodes for $n = 0$, $n = 1$ and $n = 2$ are shown in Figure 2.11 in an example.*
3. *Iterate through all nodes with maximal distance $n$ from any leaf node in tree $A$. Let the current node be $a$.*
   - *If $l(a)$ is found in $h$, iterate over the set $h[l(a)]$. Let $b$ be the current node: If all children of $a$ are mapping candidates of children of $b$ add $a$ as mapping candidate to $b$.*
4. *If more than zero nodes with maximal distance $n$ from any leaf node were found, set $n = n + 1$ and go to 2.*

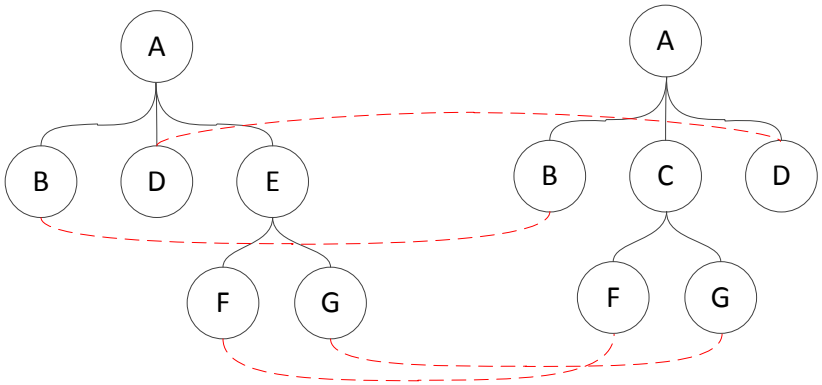



Figure 2.10: Example bottom-up matching

5. *Iterate through all unmarked nodes in B in pre-order. Let $b$ be the current node:*
   - *If $b$ has mapping candidates, map the current node and its whole sub-tree to the sub-tree of the candidate with lowest pre-order number. Mark all nodes in the sub-tree in B.*

This is similar to the method described in [47] by Valiente. In Figure 2.11 this method would correctly map the smaller sub-tree rooted in the node with label "A" on the left hand side to the same sub-tree on the right hand side. The rightmost node with label "D" on the right hand side is correctly mapped, because the pre-order number of the node being mapped to is lower than the pre-order number of the wrong mapping candidate.

An example output of this algorithm can be seen in Figure 2.12, which shows all the mappings that a bottom-up method can produce. The node with label "E" on the left hand side can, for example, not be mapped to the node with label "E" on the right hand side because its child was renamed. The node with label "D" cannot be mapped because it was renamed. This prevents us from mapping the node with label "A" because not all children of "A" were mapped.

Valiente et al. claim that this is easily extensible to unordered matching and that the complexity of such an algorithm is $\mathcal{O}(n)$. Looking close, the algorithm he describes in [47] seems to have $\mathcal{O}(n^2)$ in degenerated cases (e.g., if the root node has $\mathcal{O}(n)$ children) and is only extensible to unordered trees with unique child labels.
If there are no nodes with the same label and same parent, our bottom-up matching algorithm can also be applied to unordered trees if the children are sorted before being compared in step 3. Extending the algorithm to work without this restriction would be difficult as then sub-tree permutations would have to be considered. We did not use this method as hash matching is easier, faster, and produces the same results. It does not have problems with nodes with the same label and parent. We will also extend the hash matching to handle unordered trees.

### 2.4.3 Hash Matching

Instead of using the relatively complicated bottom-up matching, one can use a hash matching algorithm. The result is the same, but there is a small (negligible) chance, that wrong results are produced, due to hash collisions.

#### Hash Matching for Ordered Trees

Doing a matching with the aid of sub-tree hashes is easier with ordered trees, than with unordered trees. If one imagines, e.g., an XML representation of a tree, the ordered sub-tree hash would simply be a hash of the part of the XML file in which the sub-tree is described. This is only a way of thinking about it. We did not actually do it. Obviously, trees can be represented by XML and XML by a tree (one way to do that is described in Section 5.1.1). Because every node in a tree represents a smaller (sub-)tree as well, every sub-tree can be represented by XML. XML is a string of characters which can be hashed. Sub-trees with equal hashes are then isomorphic

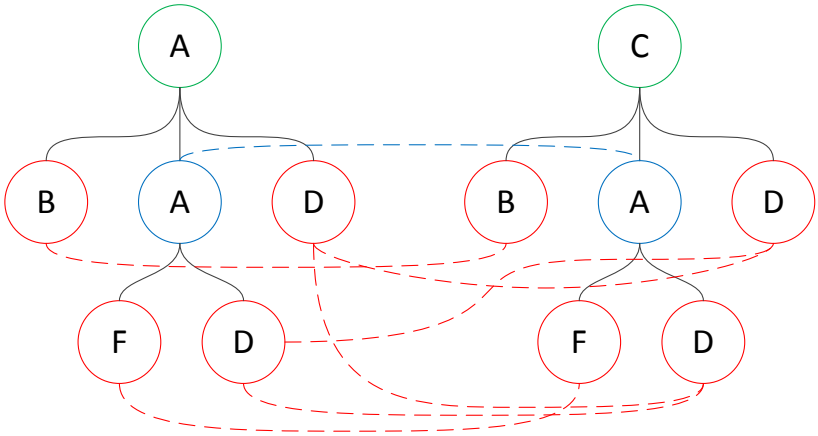



Figure 2.11: Candidate mappings during the bottom-up matching algorithm. Nodes considered with $n = 0$ are red, those for $n = 1$ blue and those for $n = 2$ green. The coloring for the candidate mappings is the same.

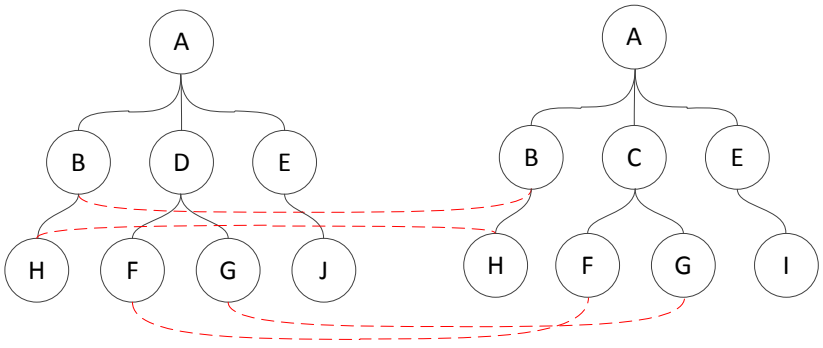



Figure 2.12: Example bottom-up matching produced by the algorithm. The node with label "E" cannot be mapped because its child was renamed.

including child order, in the same sense that XML files which are identical describe the same tree. If we start trying to map with decreasing sub-tree size, a bottom-up matching will be produced.

The calculation of the sub-tree hashes can be done directly by an iteration in post-order over the tree. The post-order traversal first processes the children and then their parent. Therefore, every time we process a node the sub-tree hashes of the children are already available. This is shown in Figure 2.13.
Let $h$ be a hash map which maps sub-tree hashes to a set of nodes in which those sub-trees are rooted. Let $Q$ be a priority queue working on tree nodes. Nodes with larger sub-tree size have higher priority. Let $sh(n)$ denote a function which returns a sub-tree hash for a sub-tree rooted in $n$. This sub-tree hash is calculated as already described and stored in the node. The hash matching algorithm for two trees $A$ and $B$ works like this:

**Algorithm 2.3.** *Hash matching ordered trees*

1. *First calculate hashes for all sub-trees in tree $A$ and put them into hash table $h$ with the hash of the sub-tree as key and the root of the sub-tree as value.*

2. *Calculate the sub-tree hashes in tree $B$.*

3. *Enqueue the root node of tree $B$ in priority queue $Q$.*

4. *Dequeue a node $n$ representing a sub-tree from $Q$ and look up in the hash table $h$ if there are isomorphic sub-trees in $A$.*
   - *If an isomorphic sub-tree was found, map the whole sub-tree to an unmapped one in the set $h[sh(n)]$.*
   - *If it was not found en-queue all children of $n$ to $Q$.*

5. *If $Q$ is not empty, go to 4. else terminate.*

Using the priority queue prevents small trees from being mapped first. Fixing those mappings first would have a high chance of producing disabling mappings and thus to incorrect results.

**Definition 2** (Maximal bottom-up matching)**.** *A bottom-up matching (Definition 1) with a maximum number of mapped nodes, i.e., there is no other bottom-up matching which maps more nodes.*

The hash matching algorithm produces a *maximal bottom-up matching*: If it would not produce one there would either exist an unmapped node in $A$ with its sub-tree being mapped to a sub-tree in B with unmapped root node. Obviously, as this is a bottom-up matching, whole sub-trees have to be mapped. Having a non-maximal bottom-up matching would mean there is a sub-tree which could be mapped, but is not or that mapping a smaller sub-tree disabled a better bottom-up matching. As

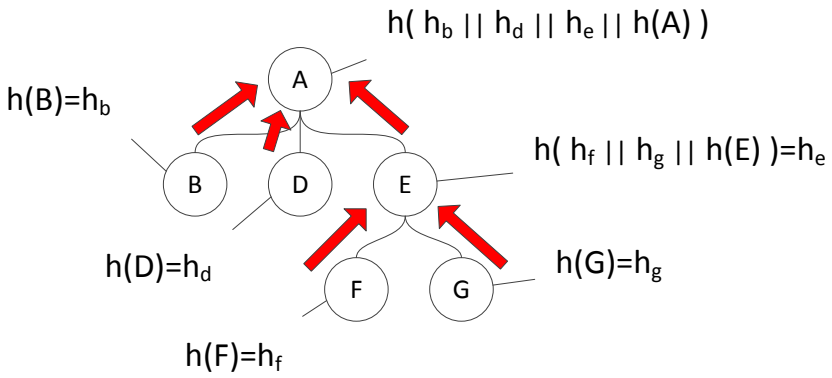



Figure 2.13: Calculation of sub-tree hashes (|| concatenates the string representation of the hashes, $h$ is the hashing function, which hashes the node's contents. If we have nodes with labels and types $h(n) = h(l(n)||t(n))$

the algorithm guarantees [3] to look at every unmapped node – looking at nodes with larger sub-trees first[4] – this node would have been discovered and mapped.

### Hash Matching for Unordered Trees

Matching unordered trees via hashes can be done using hashes insensitive to child order changes. This can be done using a commutative operator instead of the concatenation (||). See Figure 2.14, where the sub-tree hashes are calculated using the commutative operator $+$, i.e., the sub-tree hash of each node is the sum of all sub-tree hashes of its children plus the hash of its label.
Let $h$ be a hash map which maps sub-tree hashes to nodes in which those sub-trees are rooted. Let $Q$ be a priority queue working on tree nodes. Let $sh(n)$ denote a function which returns a sub-tree hash for a sub-tree rooted in $n$. This sub-tree hash is order invariant and calculated as already described and stored in the node. The matching algorithm for two trees $A$ and $B$ would then work as follows:

**Algorithm 2.4.** *Hash matching unordered trees*

1. *Calculate all unordered hashes for all sub-trees in $A$ and put them into hash table $h$ with the sub-tree hash as key and the root node of the sub-trees as value.*

2. *Calculate the unordered hashes of tree B.*

3. *Enqueue the root node of tree $B$ in priority queue $Q$.*

[3] The algorithm enqueues the root node of $B$ in $Q$. Sub-trees rooted in children of nodes in $Q$ are either mapped to identical sub-trees in $A$ or enqueued in $Q$. Because $Q$ is empty when the algorithm terminates every node has been considered.

[4] Every node is first enqueued in the priority queue $Q$, which returns nodes with larger sub-trees first. Because the sub-tree size of every parent is obviously larger than that of its children and because $Q$ returns nodes with larger sub-tree size first, $Q$ returns nodes with larger sub-tree size first.

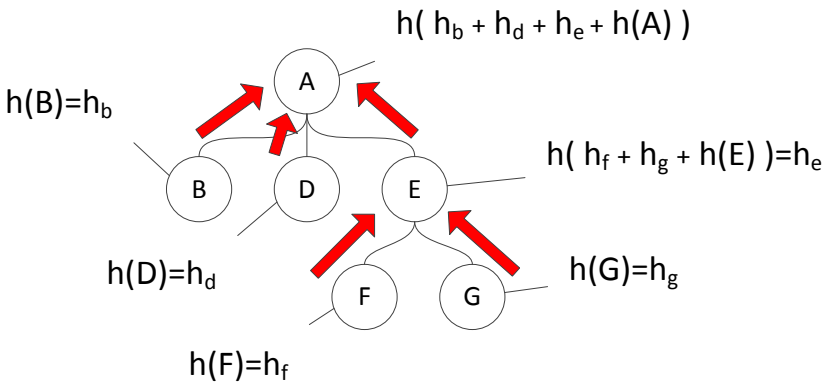



Figure 2.14: Calculation of sub-tree hashes ($h$ is the hash function hashing variable sized byte representations of integers or strings to a fixed size bit sequence, $+$ adds the bits returned by the hash function)

4. *Dequeue a node $n$ representing a sub-tree from $Q$ and look up in the hash table $h$ if there are isomorphic sub-trees in $A$.*
   - *If there are such nodes, map an unmapped node $a \in h[sh(n)]$ to $n$ using the algorithm below (Algorithm 2.5).*
   - *If there are no such nodes, enqueue all children of $n$ in $Q$.*
5. *If $Q$ is not empty go to 4. else finish.*

Mapping sub-trees is more complicated because sub-trees are only isomorphic excluding child order.

**Algorithm 2.5.** *Mapping sub-trees isomorphic exluding child order*

1. *Map the current node $a$ in $A$ to $n$ in $B$.*
2. *Iterate over all children of $n$. Let $c$ be the current child:*
   - *Find a node $k$ in $h[sh(c)]$[5] which is unmapped and has parent $a$.*
   - *Recursively call this algorithm with $k$ and $c$ as paramters.*

## 2.5 Grams

The concept of grams is widely used to convert sequences of elements of variable length into another representation – a list of grams – which facilitates calculating the similarity of those sequences. Often the name *gram* is preceded with the number of elements in the sequences which build on of those $n$-grams. The 2-grams of the sequence of characters *Hello!*, would, for example, be $\{He, el, ll, lo, o!\}$. Having such a representation, we can calculate the similarity between different words by

[5]The two sub-trees being mapped are isomorphic, therefore such a node can always be found

measuring how many grams are the same normalized by the total number of different grams. The string *Helo* would have grams $\{He, el, lo\}$ and would be similar because it has three grams in common out of five different grams. If every character can be represented with one byte, such a $2$-gram is essentially a 16bit number. We have reduced the task of finding a similarity measure on strings to the task of counting how many numbers two sets have in common, which is both computationally more efficient and easier to do, than doing it directly on strings. One can apply this concept of grams on trees as well, transforming the task of finding a tree similarity measure to the task of reducing a tree to a gram representation.

# 3 Related Work

Looking at the problem at a wider scope, there a great deal of related work. One may, for example, be able to modify methods which only calculate the number of operations needed to transform one tree into another (tree edit distance) in such a way that it calculates the associated operations as well. Then there are other methods which calculate tree edit scripts. They often have only a limited set of operations, e.g., only few support the *copy* operation and most of them work only on ordered hierarchical data. Some of them are XML specific, which should not be a problem as one can abstract the underlying method to arbitrary tree structures – but some use stronger assumptions and use XML IDs or only map XML elements with same paths to each other, causing them to be inapplicable in the general cases.

## 3.1 Tree Edit Distance Measures

Tree edit distance methods compute the number or cost of edit operations necessary to transform one tree into another. Those methods do not produce a list of operations to transform the tree, but only a number representing the effort one needs to transform it. Nevertheless, exact algorithms usually simulate the application of operations such as $insert$ and $remove$, they just do not keep track of them.

### 3.1.1 For Ordered Trees

Ordered trees are trees where the sibling order of elements is important and where changing it results in tree distance increases. One may further classify existing tree edit distance methods by the operations they consider. $Move$ and $copy$ operations do, however, make the problem significantly harder. The only known exact methods with polynomial runtime work only with $insert$, $remove$ and $rename$ operations, which are the operations considered as basic.

#### Exact Methods

Early on Zhang and Sasha [51] presented an exact algorithm which computes the edit distance between ordered trees. Its worst-case runtime is $\mathcal{O}(|T_1| * |T_2| * \min(depth(T_1), |leaves(T_1)|) * \min(depth(T_2), leaves(T_2)))$. With $|T_1|$ and $|T_2|$ being the number of nodes in the respective trees and $depth(T)$ and $|leaves(T)|$ the depth and the number of leaves of the trees. Thus, if the number of nodes is $n = \max(|T_1|, |T_2|)$ (we assume this definition of $n$ in the rest of this chapter) and the tree is balanced such that the depth is $d = \log(n)$, Zhang and Sashas algorithm runs

in $\mathcal{O}(n^2 \log^2 n)$. This algorithm still has a worst case runtime of $\mathcal{O}(n^4)$ for unbalanced trees, however. Klein et al. [22] improved this to a worst case runtime of $\mathcal{O}(n^3 \log n)$. Dulucq et al. [13] presented an algorithm with same worst case runtime but less average runtime. Recently Demaine et al. [12] defined the class of decomposition methods, which all the previous algorithms belong to, and presented an algorithm with worst case runtime of $\mathcal{O}(n^3)$ in this class. The best case and average time runtime is $\mathcal{O}(n^3)$ as well, though, so this still left room for improvement: RTED [37] has the same worst case complexity, but improves on the average runtime and thus beats all previous algorithms in all tree configurations.

By restricting the operations one can make the problem easier and the worst case complexity better. Those methods yield a distance which is always bigger than the tree edit distance without those restrictions. For example, Zhang introduced the constrained ordered tree edit distance [53] where distinct sub-trees are mapped to distinct sub-trees [1] This is done with worst case complexity $\mathcal{O}(n^2)$. The alignment distance considers instead the problem of aligning two trees [2]. This can be done in $\mathcal{O}(n^2)$ [20] if the degree (maximum number of children) is bounded. Contrary to the alignment distance of strings, however, this alignment distance can be bigger than the tree edit distance and thus can only be used as an approximation. Selkow's algorithm [42] can only delete and insert leave nodes, is a top-down matching method, and runs in $\mathcal{O}(n^2)$. A bottom-up distance for trees is a distance which guarantees to map all children of mapped nodes. Thus, if, for example, leaf nodes change, its parent cannot be mapped any more – but it can be done efficiently in $\mathcal{O}(n)$ [47].

Zhang's method has been extended with operations on whole sub-trees (previously mentioned as *subtreeRemove* and *subtreeInsert* in [4]. The complexity remains the same. Exact methods with *move* or *copy* operators have not yet been proposed – those operations would probably increase the complexity significantly.

### Approximate Methods

Because computing the exact tree edit distance is costly, and often prohibitively so, there is a strong incentive to approximate it. Often these approximations can be shown to be strictly smaller or larger than the exact distance. Then they can be used as a filter when one searches for a similar tree within a certain distance[3].

One of the simplest approximation methods is using the string edit distance be-

[1] More formally, for a constrained mapping M which maps nodes of tree A to nodes of tree B, it has to hold for every three mappings $M(a_1) = b_1, M(a_2) = b_2, M(a_3) = b_3$ that $lca(a_1, a_2)$ is a proper ancestor of $a_3$ iff $lca(b_1, b_2)$ is a proper ancestor of $b_3$.

[2] Two trees can be aligned by first adding dummy nodes such that they are isomorphic when labels are ignored. The alignment distance is then the number of labels which differ if the two isomorphic trees are overlayed.

[3] Let the task be to find trees with edit distance $t$ from tree $T$. Let $d_e$ be the exact distance from T to another tree. If we have a distance measure which is faster than calculating the exact distance and which calculates distances $d_u$ which are always greater than the exact distance ($d_u \geq d_e$), we can avoid calculating this exact distance for trees for with $d_u \leq t$. Having tree edit distances $d_l$ which are strictly smaller ($d_l \leq d_e$) and faster to calculate than the exact distances, allows us to exclude trees quickly from the similarity search if $d_l \geq t$.

tween the pre- and post-ordered tree labels, e.g., presented in [16] as lower bound[4]. The string edit distance can be computed in $\mathcal{O}(n^2)$. This gives a distance which is strictly lower than the exact distance – two different trees can have the same list of pre-ordered labels, as not every change in the tree structure has to cause a change in the label list.

Another strictly smaller approximation is the binary branch distance [50]. It converts trees with degree greater than two into binary trees. Those binary trees are then serialized to strings, which are then separated and put into lists, similar to q-grams for strings. Every tree is then represented by a list of tree-grams – in this case every node in the binary tree with its two children. Comparing the list of grams of different trees is more efficient, especially if those grams are hashed prior to comparing them. Again, two trees can have the same representing grams and still differ.

With p,q-grams [3] this concept of "q-grams for trees" becomes more flexible. In this work the grams are constructed using the ancestor relationship (configurable by $p$) and the sibling relationship ($q$). The method decomposes larger trees into very small trees with depth $p+1$ and $q$ leaves. Each of these small trees is then serialized to a string and hashed. A list of these hashes represents the data in the tree and its hierarchical relationships. The algorithm calculates the p,q-grams in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. The distance is shown to be an approximation and lower bound to the fanout weighted tree edit distance, i.e., the tree edit distance where the cost of each editing operation is weighted by the respective number of children of the node on which the operation works.

### 3.1.2 For Unordered Trees

If the order of siblings is not important, finding the exact tree edit distance becomes MAX SNP-hard [55]. Another example from this complexity class is MAX-3SAT. It is the problem of finding the boolean variables which satisfy the maximum amount of clauses in a conjunction of disjunctions of three variables (the disjunctions are the clauses). This is harder than NP where one just has to find one solution. Maximizing the number of satisfied clauses requires one to enumerate all solutions. A simple approximation within $0.5$ for MAX-3SAT: Set all variables to $true$ and count the number of clauses that are satisfied, then set all variables to $false$ and count the number of satisfied clauses. Output the one with more satisfied clauses as the solution. Since every clause is either satisfied if all variables are $true$ or if all variables are $false$, at least half of the clauses are satisfied in one case. Therefore, problems in this class can be approximated within some constant ratio, but not within an arbitrary one. This means that we can neither parametrize the quality of the result of the approximation method, nor run the method until some quality or time limit is reached. The estimation ratio is inherent in the method used, if and only if the method is polynomial and $P \neq NP$. This implies that all algorithms with runtime in $P$ cannot generate an unordered tree edit distance within $(1+\epsilon)$ times (with

[4] The edit distances strictly greater than the tree edit distance are also called upper bounds. The edit distances strictly smaller lower bounds.

$\epsilon > 0$) of the exact unordered tree edit distance. Instead, they may, for example, only generate approximations that are, e.g., within $1.5$ times of the exact distance.

Zhang et al. [44] presented an enumeration based exact algorithm for the unordered tree edit distance which runs in $\mathcal{O}(n^3 16^n)$. As this is clearly impractical, he simultaneously presents a heuristic solution based on searching in the enumeration space. The algorithm starts with a matching which maps sub-trees of same sizes. This mapping is then randomly improved by an iterative search method. Simulated Annealing is used to overcome local maxima. As the possible number of consecutive mappings in each state is bounded by $\mathcal{O}(n^2)$, this is nevertheless not very efficient for larger trees. In this case, they propose only using the initial heuristic.

Again, the operations can be constrained, such that disjoint sub-trees are only mapped to disjoint sub-trees. In this case, it can be solved in polynomial time $\mathcal{O}(n^2)$ [54] if the maximum number of children of all nodes is bounded. The alignment distance can be calculated with the same complexity under same conditions [20]. Valiente et al. claims that the bottom-up distance can be adapted easily to work on unordered trees. The complexity seems then to increase to $\mathcal{O}(n \log n)$, however.

By sorting the siblings lexicographically by label, prior to the pq-gram construction, the concept of pq-grams can be adapted to unordered trees. This is done in [2], is shown to be a quite good approximation, and runs in $\mathcal{O}(n \log n)$.

## 3.2 Tree Edit Script Producing Algorithms

This section will consider pre-existing methods which produce tree edit scripts, i.e., a list of operations which transforms one tree into another, whereas previously only the cost of such a script was important. Again the complexity of this depends on the data model (unordered or ordered) and the operations allowed to transform the trees.

### 3.2.1 For Ordered Trees

Even though they only calculate tree edit distances, most exact algorithms mentioned in Section 3.1.1 can probably be extended to produce the accompanying tree edit script without any increase in complexity: All these approaches use dynamic programming and produce an $\mathcal{O}(n^2)$ table of distances between sub-trees of the two trees. By tracing the best result in that table in a post-processing step the tree edit script could then be recovered in $\mathcal{O}(n^2)$ without increasing space complexity[5]. The overall worst case complexity would still be $\mathcal{O}(n^3)$, however.

Again, one can restrict the edit operations' power. In [7] $insert$ and $remove$ only work on leaf nodes. With such restricted operations deletion of a non-leaf node would lead to deletion and insertion of whole sub-trees, making the reduction in

[5] There is an implementation for this for the Zhang and Sasha [51] algorithm; For the other methods this is a conjecture. Other publications hand-wave this problem.

operation power more severe than the other power reductions like the align distance or the constrained ordered tree edit distance. The complexity is $\mathcal{O}(n^2)$ and thus as good as the align distance and the constrained ordered tree edit distance.

**With Move Operator**

The addition of a move operator makes the problem significantly harder ([31] claim by citing [52] that it does make it NP-hard, but the cited paper only proves this for unordered trees, so the claim is unproven). Therefore, only approximate methods have been developed. [10] imposes some restrictions on hierarchical documents such that there is a strict hierarchical order between labels: The example given are LaTeX documents where, e.g., a subsection is always in a section. The matching algorithm is a bottom-up one, which uses a heuristic optimized for text. As a bottom-up tree edit distance, this is sensitive to changes, e.g., in the leaf nodes. [25] extends this work and removes the hierarchical order restriction. Both have complexity $\mathcal{O}(n^2)$.

One successfully applied concept is that of tree hashes. By calculating a unique hash for each sub-tree – which can easily be done – those sub-trees can be found in other places by hash look-ups if they have not changed. This can be compared to the already mentioned bottom-up distance calculation, but is faster if there are only few changes. XyDiff [31] uses this method and then goes on to map nodes to each other, which are in the vicinity of nodes mapped by the tree hashes. For example, it may map the parent of a sub-tree mapped via hashes to the parent of the mapped node, if the label of that parent in the first and second tree are the same. The overall algorithm runs in $\mathcal{O}(n \log n)$ and produces good results if larger unchanged sub-trees are present. Producing an ordered tree edit script actually requires an additional reordering step here. By removing this step one can use the same approach for unordered trees.

Diffxml [33] implements the previously mentioned bottom-up method with hierarchical order restriction [10] and another previously mentioned method [7], where the $insert$ and $remove$ operations are restricted to leaf nodes. Both methods are by Chawathe et al. Additionally to the rather dubious choice of algorithms the diffxml implementation is flawed and fails its own test cases. With treepatch [23] the patch format gets extended and the shortcomings mentioned, but not fixed. Unfortunately, diffxml is the first result on Google if one searches for "xml diff".

In [39, 26, 27] a three-way merging algorithm for XML is described, which includes an algorithm for calculating diffs between XML documents (3DM). It works in a bottom-up fashion, mapping trees using their content. It also uses the neighbourhood of tree nodes to produce mappings, thus if the left and right siblings of a node are mapped to each other it assumes that the node in-between is in between the mapped nodes of those siblings. The algorithm has worst-case complexity $\mathcal{O}(n^2)$ and $\mathcal{O}(n \log n)$ if changes between trees are small.

In [49] it is assumed that one can find a key for each node that does not collide with the keys of its siblings. Nodes are then mapped only to siblings. This can be done in $\mathcal{O}(n)$, but the result may be really bad compared to other methods because the tree structure is not considered at all – it is basically a top-down label matching.

The algorithm also only detects moves to siblings. By removing the reordering step via those move operations, this method also works for unordered trees.

### 3.2.2 For Unordered Trees

As mentioned, some methods in the previous section can be adapted, such that they work on unordered trees as well. There are some specialized algorithms as well, however.

One of the earliest such works is MH-DIFF [9]. It first considers all possible mappings between the two trees. Then those mappings, which obviously can only increase the cost, are pruned. The problem is then reduced to a bipartite weighted matching problem by assigning an approximate cost to each possible mapping of each node. The overall algorithm has complexity $\mathcal{O}(n^2 \log n)$. Most effort is spent in the pruning phase. As the method emits $update$, $insert$, $remove$, $move$ and $copy$ operations, it is similar to the work presented in this thesis, its time complexity is, however, quadratic, whereas ours is linear on average.

In [48], a method without $move$ and $copy$ operations and $insert$ and $remove$ operations which are restricted to leaf nodes, only nodes with the same path are mapped to each other, i.e., nodes can only be mapped to other nodes if all ancestors of both nodes have the same labels. This reduces the search space in such a way that the worst case complexity is $\mathcal{O}(n^2)$ if the maximal number of children in both trees is bounded. As previously discussed, restricting operators on leaf nodes sometimes leads to a severely inflated tree edit script. The additional path restriction does not add to script quality as well. The work does, however, introduce the concept of a tree hash that is invariant to the sibling order and thus quite useful. It is used to reduce the runtime of the algorithm in cases where changes between trees are small. It is not used, like in other algorithms, e.g., XyDiff, to produce a bottom-up mapping. The average runtime of this method is improved in [45] with regards to hierarchical biological data. The worst case complexity is not improved.

## 3.3 Summary

Table 3.1 shows all the available (i.e., online as download) tree diff tools, which we could test and which could be considered competitors to the method described in this thesis, with accompanying academic work, their properties and shortcomings. Of the few methods only XyDiff has linear complexity. It has a solid implementation in C++ and Java as well. The other methods are all quadratic in runtime. All the methods produce only approximately minimal tree edit scripts. Matching quality and runtime have been evaluated in [40, 17]. We compare the methods listed here with our own method in Section 6.4.

| Tool | Worst-case complexity | Supported operations | Ordered/ Unordered | Comment |
|---|---|---|---|---|
| X-Diff[48] | $\mathcal{O}(n^2)$ | *insert*, *rename*, *remove* | Unordered | No *move* support, long runtimes |
| 3DM[39] | $\mathcal{O}(n^2)$ | *insert*, *rename*, *move* | Ordered | Implementation emitted no *move* or *delete* operations. This caused wrong edit scripts. |
| DiffXML [33] | $\mathcal{O}(n^2)$ | *insert*, *remove*, *rename*, *move* | Ordered | Excessive amount of edit operations even for simple changes. Slow for larger trees. |
| XyDiff[31] | $\mathcal{O}(n \log n)$ | *insert*, *remove*, *rename*, *move* | Ordered | Java and C++ version available. C++ version unmaintained. |

Table 3.1: Tree edit script generating methods with implementation and describing academic work.

# 4 Matching Method

This chapter will have an in-depth description of the proposed matching method used to construct the approximate cost-minimal tree-to-tree matching. This method can be roughly separated into three steps:

1. A simple straightforward matching step which tries to find obvious common structures in both versions of the tree. The nodes mapped in this step do not have to be considered in subsequent matching steps and thus significantly improve their speed.

2. Construction of feature vectors for unmapped sub-trees of both trees, i.e., vectors which are similar if sub-trees are similar and have fixed length, small representation contrary to the complete sub-tree structure, which is variable length. Then, construction of appropriate index structures for those feature vectors in order to have fast look-ups of nearest neighbours.

3. Mapping of previously unmapped sub-trees with the help of feature vectors.

## 4.1 Method Overview

Figure 4.1 gives an overview over the proposed method. First the trees are read from files. XML, HTML, and JSON is supported. This is described in Section 5.1. Then a hash-matching step is performed, excluding small sub-trees. How the hash-matching is done was already described in Section 2.4.3. Afterwards, a top-down matching is done as described in Section 2.4.1. Subsequently, feature vectors of sub-trees in both trees are constructed. How this is done is described in Section 4.4. The size of the feature vectors is reduced as described in Section 4.4.3. Next, they are put into an efficient index structure supporting nearest neighbour queries. Index structure candidates are described in Section 4.5. The feature vectors and the index structure are then used to map previously unmapped sub-trees – see Section 4.6. Thereafter, another hash-matching is performed as of Section 2.4.3, this time including small sub-trees. Being done with the matching phase, the tree edit script is constructed from the generated matching. How this is done is described in Section 5.2. Finally, we show how this tree edit script can be saved as an XML or JSON file in Section 5.3

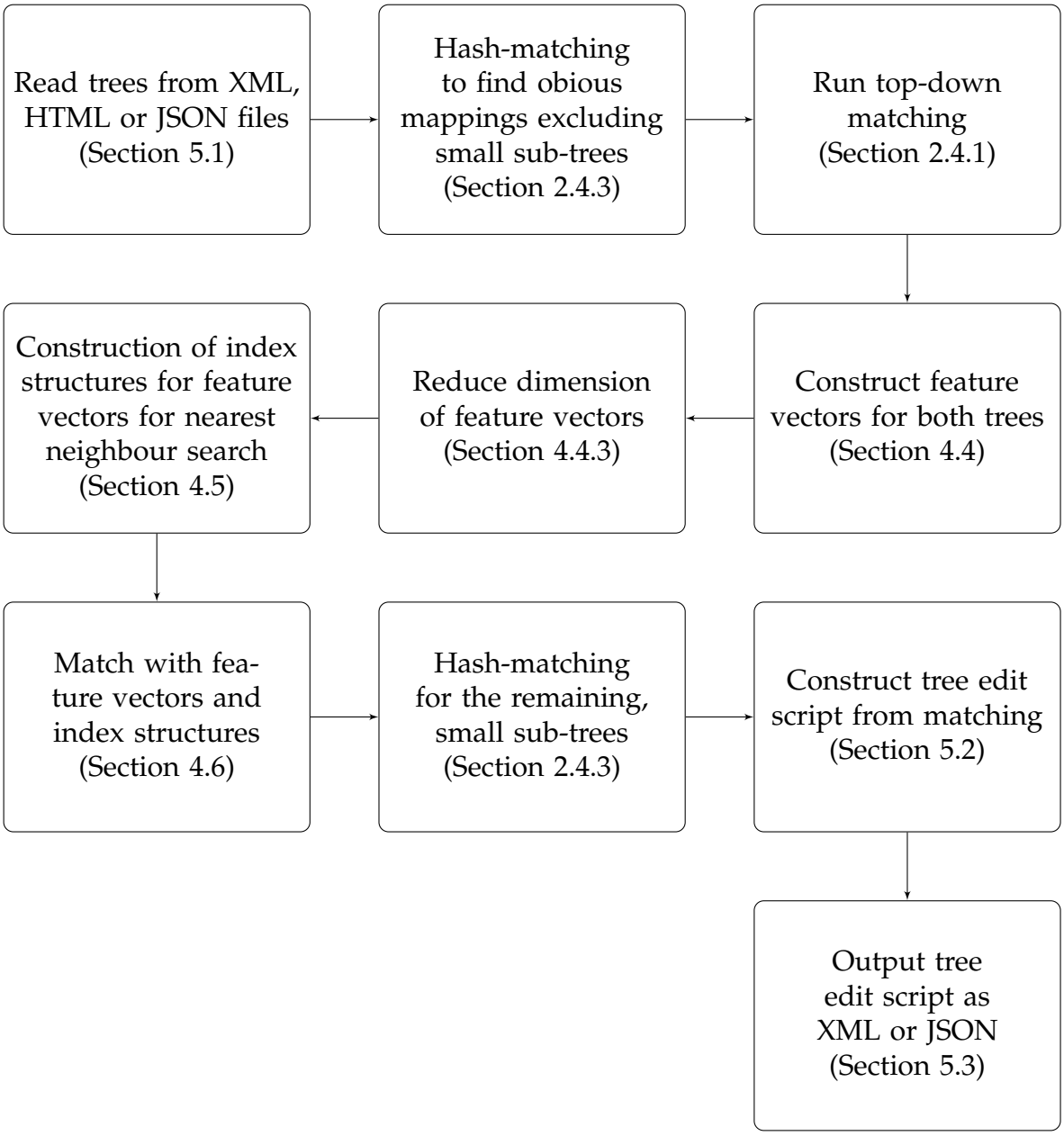



Figure 4.1: Overview of the steps of the change detection and correction method from input to output with references to where those steps are described

## 4.2 Finding Simple Matchings

In this step of the algorithm, we try to map large parts of the trees as possible. The goal is not to find all possible mappings – only to find the obvious ones. We will use methods and concepts already described in literature and successfully applied. Those are top-down matching [42, 49] and matching using sub-tree hashes [31], as in Section 2.4.1-2.4.3.

This part of the proposed method uses a combined top-down and hash matching to rapidly find obvious mappings. First, a bottom-up hash matching is done, followed by a top-down matching. There are some difficulties, however, if those methods are used as a pre-processing step to other more sophisticated matching algorithms. The hash matching can, for example, produce mappings like the one in Figure 4.2, where it maps a smaller sub-tree to the wrong sub-tree. Once trees are mapped like this, succeeding matching algorithms can cause a lot of edit operations. This is an artifact of the basic approach used in the bottom-up hash matching algorithm.

We decided to implement measures to prevent some of these incorrect hash mappings. Those are:

1. Sub-trees have to be sufficiently large in order to be mapped in the hash matching step. We only map sub-trees which contain more than 3 nodes, at first. This could have prevented the wrong mapping in Figure 4.2, where for example the node with label “E” is incorrectly mapped. Only mapping sub-trees with more than three nodes would have caused the bottom-up matching to find no mappings in this example. This is better than producing incorrect mappings, as a top-down matching would produce better results.

2. Nodes are preferably mapped to nodes having ancestors with equal labels in both trees. This avoids the problem exhibited in Figure 4.2 as well, because for example the node with label “E” on the left hand side would be correctly mapped to the node with equal label and parent “C” on the right hand side.

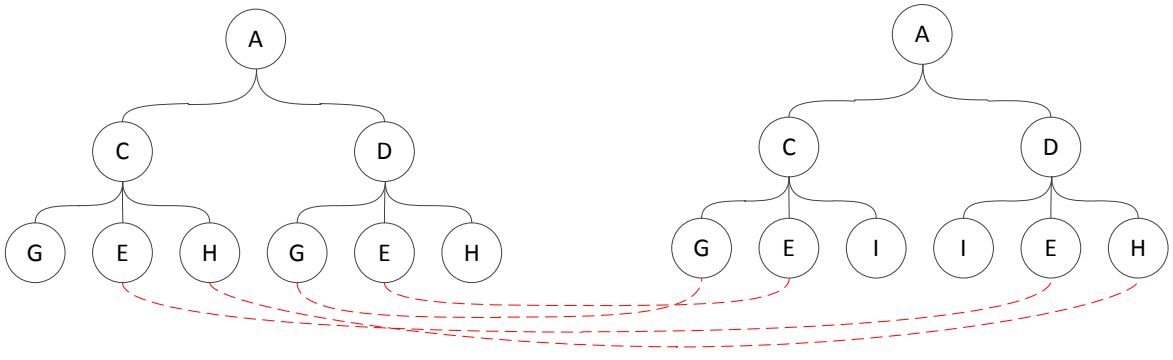



Figure 4.2: Bad bottom-up matching result: Mapping node with label “C” on the left to the one on the right now, causes a lot of moves.

3. Nodes are preferably mapped to nodes with same path.

4. If a sibling of a node $a$ is already mapped to node $b$, $a$ will be preferably mapped to nodes having the same parent as $b$. This avoids the problem in Figure 4.3, where the node with label "E" is incorrectly mapped. After correctly mapping node with label "G", "E" would be mapped to the correct node, because "G" is a sibling of "E".

These measures do not prevent the algorithm from creating any wrong or disabling mappings[1]. Their usefulness can be increased by mapping parents of mapped sub-trees to each other in the bottom-up hash matching step like in XyDiff [31]. If two sub-trees are mapped to each other, one looks at their parents and maps them if they have the same label. This is done recursively upwards towards the root, with the size of the mapped sub-tree determining how often we recurse upwards. The dependence on the sub-tree size prevents a small mapped sub-tree from disabling a lot of other correct mappings. Once ancestors are mapped, measure 4. works more effectively.

## 4.3 Feature Vectors

Feature vectors are a widely applied concept used to embed high dimensional data into lower dimensional spaces. One area where they have been used extensively is computer vision. Let's take the task of searching for similar images as an example. This is useful for object recognition. We have an image database and an input image for which we want to find similar images. Let the images have, for example, a size of $1024 \times 768$ pixels with three colors each – a dimensionality of more than 2 million. Using standard classification or clustering methods on data with such a

[1] All incorrect mappings can only be avoided if we use a non-heuristic matching method, which would lead to a high runtime complexity.

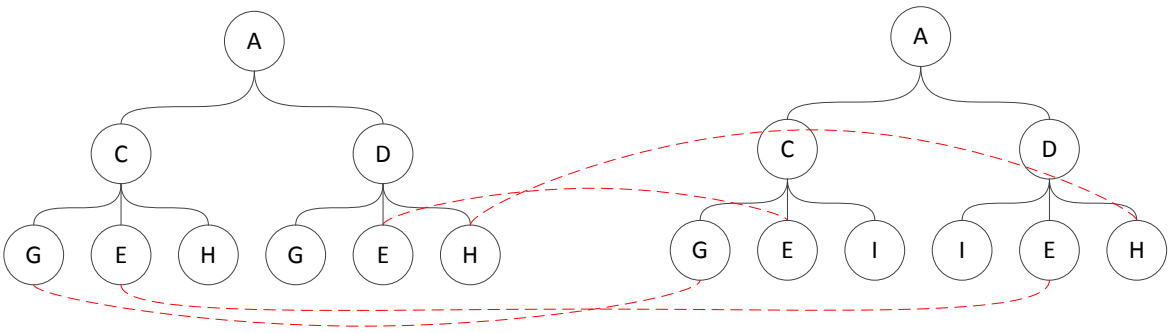



Figure 4.3: Bad bottom-up matching result: Children of one node are mapped to nodes with different parents even though they could be mapped to nodes with same parent.

high dimensionality is not possible. The size of the data has to be reduced drastically. This is done by looking closely at what is significant in the images and what we mean if we say "similar" images. Usually, corners in images are significant – though this may vary with the motive – and similar means the content of the image is roughly the same, disregarding illumination changes, noise, small changes in viewpoint, scale, rotation, and small distortions. The whole similarity search for images may then work as follows:

1. Find significant portions (features) in the image, e.g., corners. Examples for corner points are shown in Figure 4.4.

2. Build vectors describing the features in a way that is invariant to illumination changes, noise, small changes in viewpoint, scale and rotation, with the property that the vectors are similar under some metric distance function, if the features are similar. How such a feature vector may look like for image features is shown in Figure 4.4 on the right hand side.

3. Put the feature vectors into structures which allow fast nearest neighbour queries, i.e., fast searches for similar features.

4. Output the image as similar which has most features in common with the input image.

One of the most popular algorithms which constructs such image features is SIFT (Scale-invariant feature transform) [29]. Everyone who has stiched a panorama before has probably used it or a variant thereof. SIFT transforms the features into a 128 dimensional vector, which has most of the desired properties. In the paper they then use a k-d tree (Introduced later in Section 4.5.1) for nearest neighbour queries in the feature vector space.

## 4.4 Construction of Feature Vectors

The combined bottom-up hash matching and top-down matching, often significantly reduces the number of unmapped nodes. There are, however, some cases where sub-trees cannot be mapped or the matching algorithm cannot find any mappings at all. Examples where the previous matching algorithm fails are easily constructed:

- Add a sibling to every leaf node, change the label of every leaf node or change the label of one child of every node. This prevents the bottom-up hash matching from mapping any nodes – all the sub-tree hashes change.

- Change the label of the root node. This prevents the top-down matching from mapping any nodes.

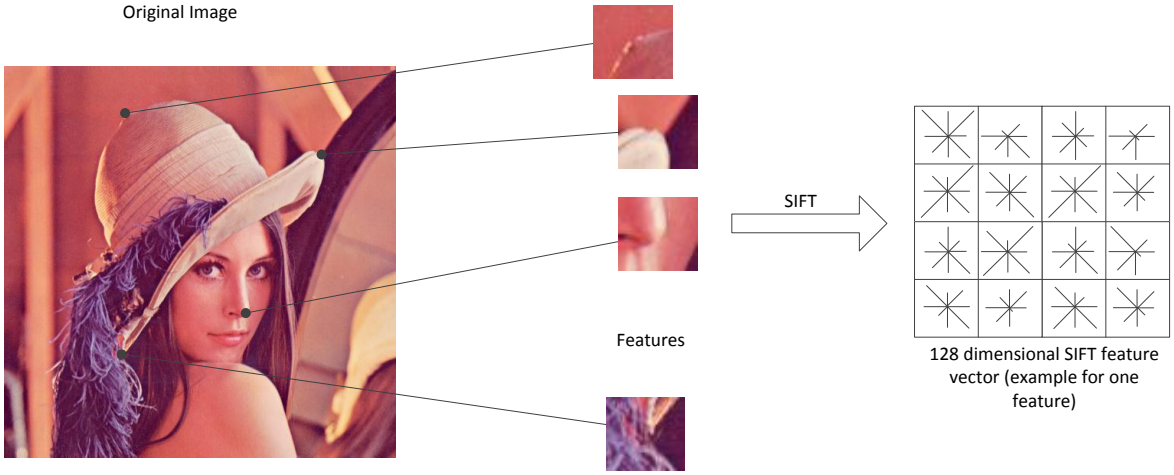



Figure 4.4: Generating feature vectors for similarity search for images with SIFT

In practice these kinds of changes are structural changes. For example, in file systems a versioning system may add a *.svn* folder to every folder or in XML attributes are renamed, deleted or added automatically. Even so such changes do happen, and should be handled.
Feature vectors are used to map those remaining, unmapped nodes. Feature vectors are fixed length vectors with the property that similar sub-trees produce similar vectors with regard to a distance metric. To be specific, two similar sub-trees may have two feature vectors with the distance in, e.g., euclidean space small. By mapping the sub-trees into a regular space, data structures for efficiently organizing and querying multidimensional data can be used to find nearest neighbours to a feature vector representing a sub-tree. The probability of those nearest neighbours being similar is then high.
The constructed feature vector should be invariant to changes in the order of siblings if the matching is done on unordered trees, and it should change with changes in the sibling order if it is done on ordered trees. For example, the order independent sub-tree hash already mentioned in section 2.4.3 would qualify for a one dimensional feature vector which is similar if sibling order is changed. It is, however, unsuited otherwise as even one edit operation such as *insert*, *remove* or *rename* would change the sub-tree hash drastically. This is caused by the hash function essentially being a random function and the upward propagation of those random values – a change in a leaf node randomly changes the value of all the sub-trees which contain this leaf node.
Once we have a good algorithm which produces feature vectors we can calculate the similarity by comparing feature vectors. This similarity measure is, however, only an approximation. Due to their low dimensionality the information feature vectors contain is – by design – often drastically reduced from that of the data structures representing the original sub-trees: Different sub-trees may be mapped to one feature vector. A similarity in the feature vectors does thus not automatically mean a

similarity in the sub-trees (false positives). Therefore, the nearest neighbours in the feature vector space are merely used as candidates, and the similarity there is only used as a hint which still has to be confirmed.

### 4.4.1 Grams for Trees

As with string similarity, a successfully applied concept is to represent the large trees by short excerpts. These excerpts – also called shingles or grams, capture the tree in a sufficiently representative way. Describing the sub-tree by a set or bag of grams enables us to use efficient similarity algorithms which work with such sets. This basically turns the abstract problem of finding similar sub-trees into the more well defined and tractable task of finding sets with large intersections. How we construct the grams depends on what we want them to capture. If we are not interested in the tree structure and only in the leaf nodes, we may as well construct the grams only from leaf nodes. Similarly, if we do not want to map sub-trees with reordered siblings, we should construct grams which capture the order of nodes. There are several ways to theoretically construct the tree-grams:

- *All nodes.* Having nodes in common is a necessary condition for similarity. If structural changes are rare, simply adding the label of every node in the sub-tree to the bag of grams might be enough. This representation is invariant to structural changes and thus specifically invariant to the changes in the order.
- *Leaf nodes.* Having leaf nodes in common might be enough of a similarity indication for sub-trees. If this is the case one simple algorithm to construct a bag of grams representing the sub-trees is by iterating over the sub-tree and adding all the leaf nodes to the bag of grams. This representation is invariant to structural changes as well.
- *Nodes with ancestors.* We add each node together with $q$ of its ancestors as sequence to the bag. Captures moves of nodes to other ancestors. Does not capture order changes or if a node is moved to another node with same ancestor labels.
- *Nodes with siblings.* We add each node together with $p$ siblings left and right of it as sequence. Captures order changes and moves to other nodes.
- *Nodes with ancestors and siblings.* We add each node together with $p$ siblings and $q$ ancestors. Captures order changes and moves. If we sort the siblings of both trees prior to extracting the grams, this representation is invariant to order changes. By changing $p$ and $q$, the amount of sibling and ancestor information captured by a gram can be adjusted.
- *Other content.* If we have text nodes with a lot of content, we might want to treat that content not as a single piece of information, like smaller nodes. Then we might want to add normal string n-grams (See Section 2.5) of that text to the structure describing the whole sub-tree.

### 4.4.2 Nodes with Ancestors and Siblings: p,q-Grams

Because they capture both ancestor and sibling relationships, are configurable, can be made invariant to small order changes, and have been already successfully used in [3, 2] the concept of p,q-grams – grams with $p$ siblings and $q$ ancestors – is used to construct the feature vectors representing sub-trees in our approach. Those grams can be generated in $\mathcal{O}(n \log n)$ time and thus within the self-imposed runtime limits. How such p,q-grams are constructed can be seen in Figure 4.5, where the steps of the construction are shown using an example tree. Invariance to order changes is obtained through sorting siblings. Augsten et al. show in [2] that if this is done the permutation of a constant amount of siblings changes only a constant amount of p,q-grams. After sorting the trees the p,q-grams are build by sliding a window of fixed size over the children of every node, and then selecting all possible permutations of $p$ children as "base" and $q$ ancestors as "stem". Permuting the children within the window makes the p,q-grams robust to modifications of those children, called "children error" in [2], while still capturing sibling relationships. The "stem" captures ancestor relationships in the p,q-grams. In Figure 4.5, the sorting changes the order of the children of the node with label "A". The results of the sibling permutation within a window of size $3$ are shown as well for the children of the node with label "A". If this property is not desired, those two steps can be skipped.
After sorting the siblings, smaller trees with a "stem" representing the $p$ ancestors of and a "base" representing the $q$ siblings is constructed. If the node does not have $p$ ancestors or $q$ siblings, dummy nodes are used. In Figure 4.5 $p = q = 2$ and the dummy nodes have asterisk labels. The siblings are taken from within a sliding window of a specific size and permuted, such that all possible combinations of siblings within that window are constructed. The p,q-grams are then serializations[2] of the constructed trees. The bottom of Figure 4.5 shows the serializations of all the p,q-grams of the example tree.

### 4.4.3 Dimensional Reduction

Even though the comparison of trees may now be easier, the actual size of the tree representations has now actually gone up. We will now try to represent each subtree with as few dimensions as possible.
In literature this is often called a signature vector, or if we extract something significant and use it as representative, a feature vector.

#### Hashing the Grams

Saving the grams as strings is inefficient as the information entropy is quite low[3]. We can mitigate this quite easily by hashing the string representations of the grams.

[2] Serializing the tree in this case is easy, because we know the tree structure by way of parameters $p$ and $q$: We can simply put the labels of $p$ ancestors and $q$ siblings into an array of size $p + q$.
[3] To be exact, the information entropy of english is as low as 0.6 to 1.3 bits per letter [43]. As the grams often contain english parts, the information entropy of the grams is low.

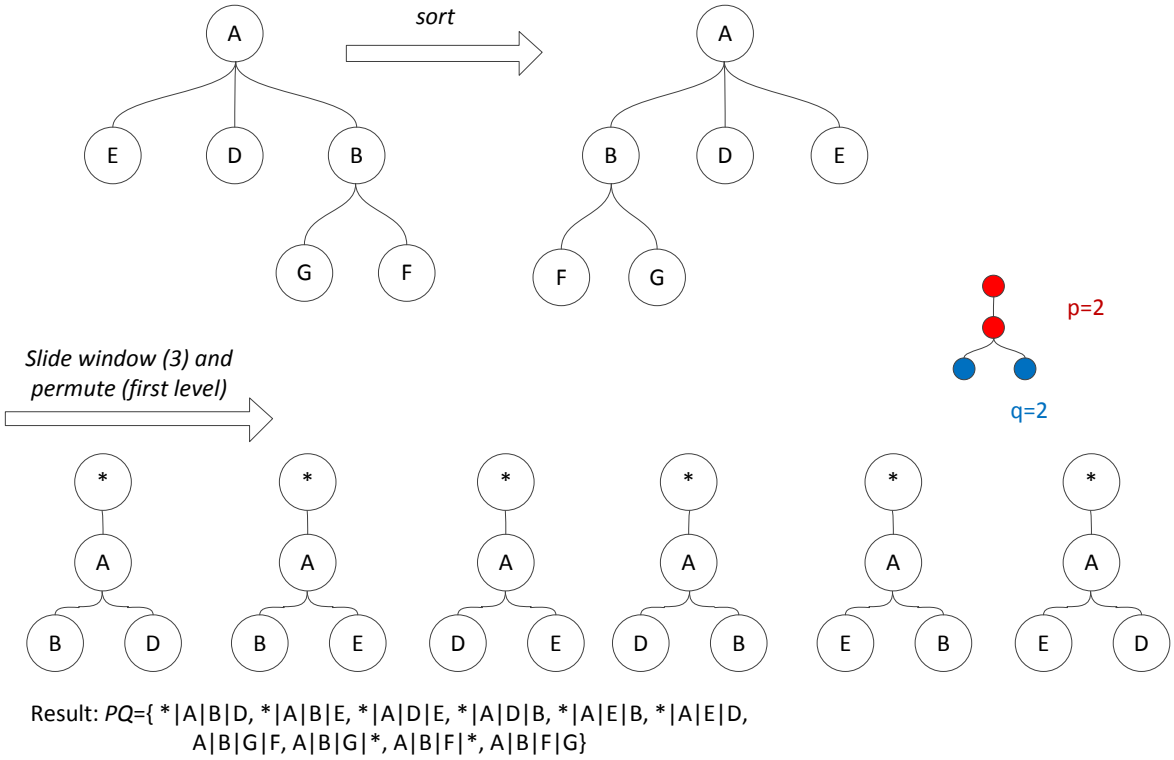



Figure 4.5: Construction of a bag of p,q-grams representing a tree

With appropriate length, e.g. 32bit, processing henceforth becomes faster as well. Note that the length of the hash output depends on the number of possible different p,q-grams and on the collision properties of the hashing function. A range of $[0, 2^{32})$ should satisfy the first requirement, that there are enough possible different p,q-grams, in most cases, however.

### Reduction via MinHash

To reduce the dimension of the signatures of the trees one can use the concept of MinHash: Let $S_A$ be a set of numbers representing tree A and $S_B$ be a set representing tree B. $min\{S_A\} = min\{S_B\}$ is then an indication for A being similar to B. By changing how $S_A$ and $S_B$ are constructed this minimum changes. If it is still the same, even though the construction method changed, it is another indication that $S_A$ and $S_B$ are similar.

Let $Pr(X)$ denoting the probability with which a random event $X$ occurs. Let $S$ be a set of hashes with a length of $b$ bits and $F$ a family of permutations from which permutation $\pi \in F, \pi : [0, 2^b) \rightarrow [0, 2^b)$ is chosen at random and it holds

$$\forall s \in S : Pr[min\{\pi(S)\} = \pi(s)] \approx \frac{1}{|S|} \tag{4.1}$$

$min\{\pi(S_A)\} = min\{\pi(S_B)\}$ would be another indication of similarity. This can be

quantified: We define similarity as the Jaccard index:

$$J(A,B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \tag{4.2}$$

A Jaccard index of $1$ would mean identical sets and $0$ dissimilar sets. It is the ratio of common elements in the set to the number of total elements. If there are $x$ common elements and $y$ dissimilar elements between $A$ and $B$, $J(A,B) = \frac{x}{x+y}$. Then with equation 4.1:

$$Pr[min\{\pi(S_A)\} = min\{\pi(S_B)\}] = \frac{x}{x+y} = J(A,B) \tag{4.3}$$

I.e., the probability of choosing one of the common elements with $\pi$ out of the $|S_A \cup S_B| = x + y$ different elements is $\frac{x}{x+y}$, because we have $x$ common elements.

In practice hash functions are used instead of permutations. If the probability of collisions is low, they behave like permutations. Of course, the variance of one single hash function is too high to use it as any kind of estimator for the Jaccard index. One can get there by combining several different minima from different hash functions. What we have then is a several times smaller set from a larger one, i.e., a dimensional reduction. This construction can also be seen in Figure 4.6, where such a feature vector is created by combining the minimal values of hashes of several p,q-grams.

**The Appropriate Similarity Measure**

We already saw the Jaccard index (4.2) as a similarity measure. The question is now, whether this is the appropriate measure. Dividing by $|S_A \cup S_B|$ normalizes the Jaccard index using the size of the participating sets to $[0,1]$. This causes counter-intuitive mappings in some cases. For example, if we map the sub-trees of node $A$ and $B$ in Figure 4.7, using the Jaccard index between those sub-trees: Let all sub-trees $S_i$ have the same size and all the elements between all sub-trees be dissimilar. Then the similarity index between $B$ and $B_1$ is $J(B,B_1) = \frac{|S_3|+|S_4|}{|S_2|+|S_3|+|S_4|} = 0.66$ and $J(B,B_2) = \frac{|S_3|}{|S_3|+|S_4|} = 0.5$. If we count the edit operations needed we would add $S_2$ from $B$ to $B_1$ and remove $S_4$ from $B$ to $B_2$, which would result in the same number of edit operations. The Jaccard index $J(B,B_1)$ and $J(B,B_2)$ is not the same, however.

The normalization of the Jaccard index also emphasizes differences between small sub-trees and relativises differences between large sub-trees. If we remove one item from a sub-tree containing two items, the Jaccard similarity index would be $0.5$. If we remove half of the items in a sub-tree containing $200$ items, it would be the same. While this would not influence the nearest neighbours of sub-trees, it would make it difficult to develop estimation methods because there one often chooses the estimation parameters such that the estimation method has the best precision for a certain range of similarity. Most of the time one wants the similarity index estimation method to be precise for a high similarity and less precise for low similarity.

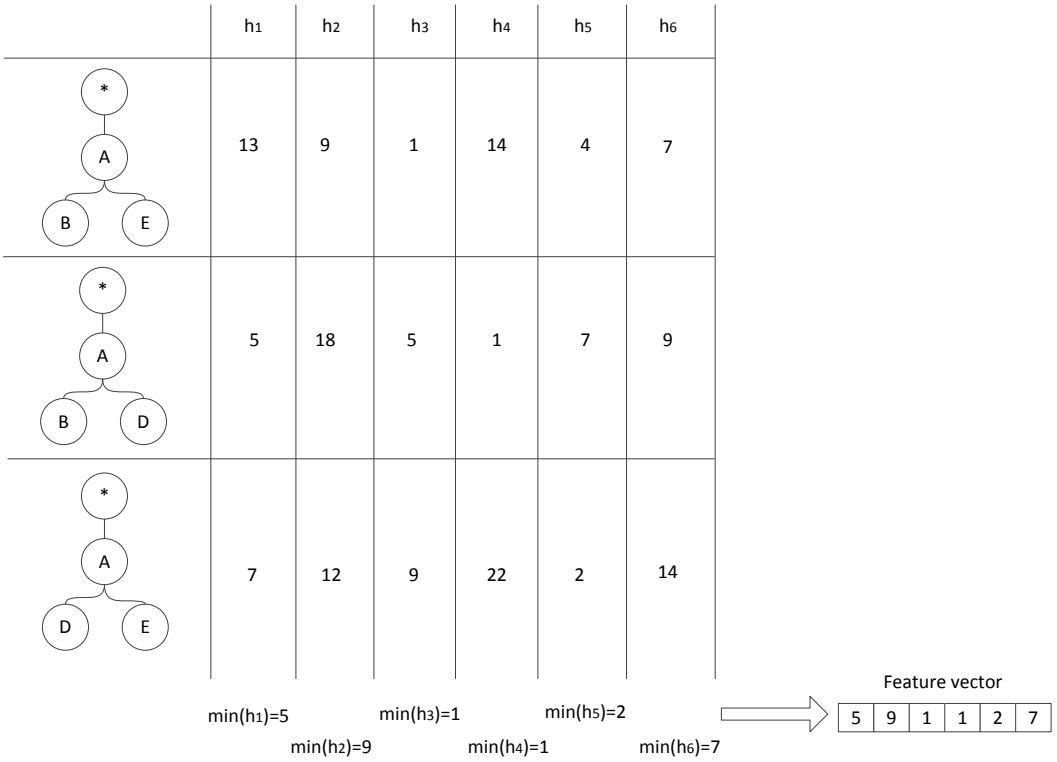



Figure 4.6: Contruction of feature vectors using the minimum of several hash functions (MinHash).

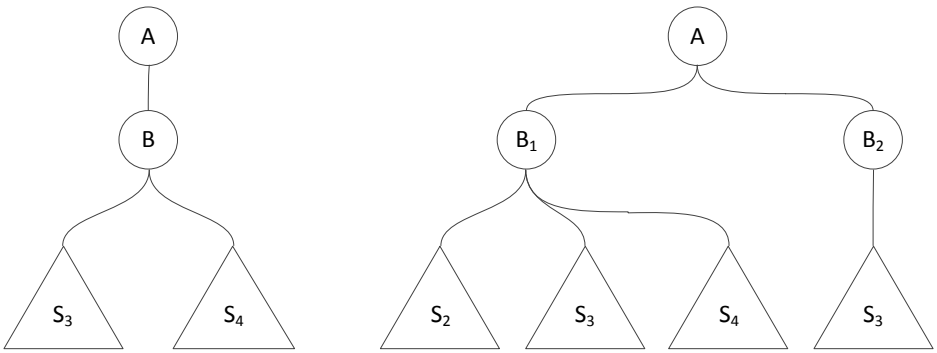



Figure 4.7: Case where using the Jaccard index to map sub-trees leads to counterintuitive results.

An example for such behaviour can be given with the *MinHash* method. After the dimensional reduction to $d$ hash values via *MinHash*, one separates those $d$ hash values into $b$ bands. The bands are then hashed and put into different hash tables, such that we can look up which sub-trees belong to a certain band. If we now want to perform a nearest neighbour search, we can calculate the bands for the sub-tree in question and look them up in the hash tables. The sub-trees found in the hash table are then likely nearest neighbours. Their Jaccard index can be estimated by looking how many times the same sub-tree is found in the same bucket in the different hash tables.

This can be seen in Figure 4.8. There we have four vector representations of sub-trees $A_1$-$A_4$, from which we take a "MinHash" four times. To avoid having to write down the permutations $\pi$, the *MinHash* is simulated here by taking the four smallest elements from the vectors representing the trees in order. Those MinHashes are then separated into two bands which are put into hash tables. To find out the Jaccard similarity to tree $B$, we do the *MinHash* and construction of bands there, as well. The bands are then looked up in the hash tables, which give back a list of sub-trees which have the same bands. The Jaccard index can then be estimated by looking how often one sub-tree is found in all the lists divided by number of bands.

Since we do not want many potential nearest neighbour candidates, we choose the size of the bands in a way to minimize the number of sub-trees returned, i.e. in the hash buckets, to a reasonable amount. Bigger bands give back fewer candidates with high Jaccard index. A sub-tree $A$ with Jacard index $J(A, B) = 0.5$ might then not be found as nearest neighbour of $B$ because the bands differ so much that both of the trees are never in the same bucket.

Instead of the Jaccard index, we could look at the symmetric difference for set similarity:

$$D(A, B) = |S_A \cup S_B| - |S_A \cap S_B| \tag{4.4}$$

This directly reflects the number of elements we have to remove from $A$ and add to $B$ if we want to transform $A$ to $B$ and as such should approximately reflect the required edit operations. It is a distance measure, i.e., the distance $D(A, B)$ between identical sets is zero while the distance between entirely different sets is $|A| + |B|$.

### From Sets to Bags

Contrary to sets an element is allowed several times in a bag (or multi-set). For example, the bag $b = \{a, c, a, c\}$, reduced to a set, would be $S = \{a, c\}$. By using bags instead of sets, we use information about the frequency of elements which may be important in some trees. Thinking, for example, of HTML pages some kinds of p,q-grams could occur frequently. As a distance metric the symmetric difference can be used:

$$D_b(A, B) = |A| + |B| - |A \cap B| \tag{4.5}$$

With $A$ and $B$ being multi-sets the intersection $\cap$ is a multi-set intersection – every element in the intersection appears as often as the minimal number of times it occurs in $A$ and $B$. For example $\{a, a, d, a\} \cap \{a, c, a\} = \{a, a\}$.

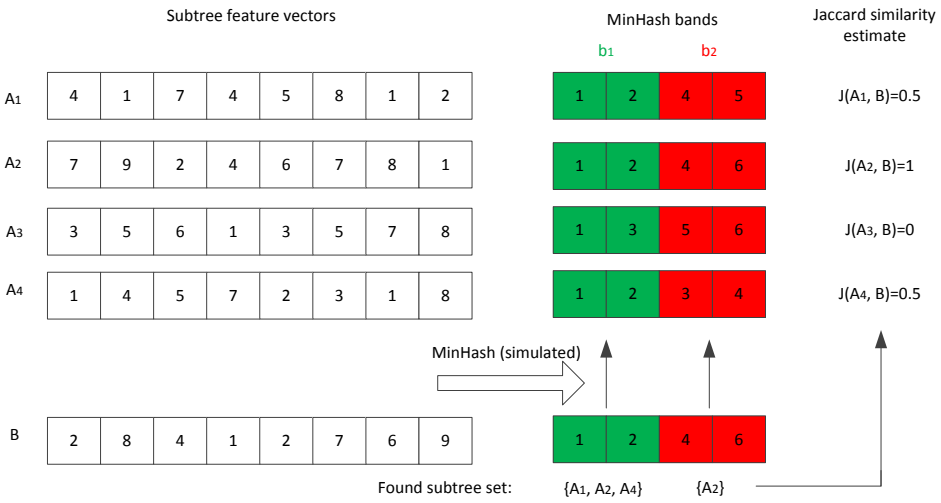



Figure 4.8: Using MinHash and bands in hash tables to efficiently estimate the Jaccard index. MinHash is simulated here by taking the four smalles elements in order.

### Hash Histograms

A straightforward method to efficiently estimate the symmetric difference is by first pseudo-randomly reducing the dimensionality of the elements, then building histograms, counting the frequency of the reduced elements, and finally defining a similarity metric on the histograms, giving an estimation of $D_b$. As it turns out the histograms can be constructed in a way such that the similarity metric is the squared euclidean distance. Because this is a proper metric, we can use efficient index structures for nearest neighbour search to find similar trees in the space imposed by this metric. If we have $n$ frequencies and $A$ is a multi-set of elements and $h$ a hash function, the frequencies $U_A$, being stored as a vector of dimension $n$, are defined as

$$\forall i \in [n] : U_A[i] = |\{x | x \in A, h(x) \mod n = i\}| \tag{4.6}$$

That means for every element $x \in A$ we increase the appropriate frequency in $U_A$: $U_A[h(x) \mod n] += 1$.
Let the number of equal elements of two bags $A$ an $B$ be $x$, the number of different or unmapped elements in $A$ be $y_A$, the ones from $B$ be $y_B$ and the bag distance (symmetrical difference) be

$$D_b(A, B) = y_A + y_B \tag{4.7}$$

Let $x + y_S = |S|$. For any bag $S$, any $x$ and any $y_S$ the expected value of $U_S$ is:

$$E[U_S[i]] = \frac{x}{n} + \frac{y_S}{n} \tag{4.8}$$

thus

$$E[U_A(i) - U_B(i)] = \frac{x}{n} + \frac{y_A}{n} - \frac{x}{n} - \frac{y_B}{n} = \frac{y_A}{n} - \frac{y_B}{n} \tag{4.9}$$

Therefore in sum:

$$E\left[\sum_{i=0}^{n-1} U_A[i] - U_B[i]\right] = y_A - y_B \tag{4.10}$$

If we now add $\delta_S = \left|\frac{y_A - y_B}{n}\right|$ to each of the frequencies of the smaller bag this expected value and thus the mean of $U_A[i] - U_B[i]$ becomes zero.

Let $y = 2\min(y_A, y_B)$ and $y_d = |y_A - y_B|$. The probability that two of the elements accounting to $y$, one from $A$ and one from $B$, increases the same frequency is $\frac{1}{n}$. $y_s = \frac{y}{n}$ of unequal elements thus increase the same frequency on average. If we look at the difference $\Delta_{AB} = U_A(i) - U_B(i)$, we notice that this is a binomial distributed random variable, because we have $\frac{y - y_s + y_d}{n}$ independent pseudo-random events for each difference, each either increasing or decreasing the difference with equal probability $p = 0.5$. Thus

$$U_A[i] - U_B[i] \sim \mathcal{B}(\frac{y - y_s + y_d}{n}, 0.5) - \frac{y - y_s + y_d}{2n} \tag{4.11}$$

For a larger number of events the binomial distribution is approximately a normal distribution and the sample variance of those $n$ random events $X$, which either increase or decrease $\Delta_{AB}$, can then be estimated with the well known formula and $\mu$ the expected value of $X$:

$$\sigma^2 = \frac{1}{n}\sum_{i=0}^{n-1}(X_i - \mu)^2 \tag{4.12}$$

With the expected value of the difference zero $E[U_A[i] - U_B[i]] = 0$, because we added $\delta_S$ to the smaller multi-set and $\sigma^2 = np(1-p)$ for the binomial distribution, the variance of this distribution is then:

$$\frac{y - y_s + y_d}{4n} = \sigma^2 = \frac{1}{n}\sum_{i=0}^{n-1}\left(U_A(i) - U_B(i)\right)^2 \tag{4.13}$$

With $y_s = \frac{y}{n}$ it follows:

$$y(1 - \frac{1}{n}) + y_d = 4\sum_{i=0}^{n-1}\left(U_A(i) - U_B(i)\right)^2 \tag{4.14}$$

$$= \sum_{i=0}^{n-1}\left(2 \cdot U_A(i) - 2 \cdot U_B(i)\right)^2 \tag{4.15}$$

The symmetrical difference between the bags is

$$D_b(A, B) = y + y_d \tag{4.16}$$

If we now multiply each of the frequencies with 2 beforehand for $n \gg 0$ with Equation 4.14 and the euclidean norm defined as $||x|| = \sqrt{\sum_{i=0}^{n-1} x_i^2}$:

$$D_b(A, B) = \sum_{i=0}^{n-1}\left(U_A(i) - U_B(i)\right)^2 = ||U_A - U_B||_2^2 \tag{4.17}$$

This is obviously a metric, because it is the square of the euclidean distance between the frequency vectors. It is not a proper metric for the underlying bags, because it is only an approximation. For example

$$A \neq B \implies D_b(A, B) > 0 \tag{4.18}$$

might not hold, i.e., two bags in the accompanying space can be different and simultaneously have distance zero. For large $n$, when the approximation becomes exact, it does however become a proper metric as we can show the triangle inequality for the exact symmetric difference $D_b$. If we separate $y$ into the two components $c_A$ and $c_B$ respectively belonging to the bags and use $y_d = |y_A - y_B|$ and that for any numbers $a$ and $b$ $|a + b| \leq |a| + |b|$:

$$\begin{aligned}
D_b(A, C) &= |y_A - y_C| + y && (4.19)\\
&= |y_A - y_C| + (c_A + c_C) && (4.20)\\
&\leq |y_A - y_B + y_B - y_C| + (c_A + c_B + c_B + c_C) && (4.21)\\
&\leq |y_A - y_B| + |y_B - y_C| + (c_A + c_B) + (c_B + c_C) && (4.22)\\
&= |y_A - y_B| + (c_A + c_B) + |y_B - y_C| + (c_B + c_C) && (4.23)\\
&= D_b(A, B) + D_b(B, C) && (4.24)
\end{aligned}$$

Therefore using this histogram hashing scheme, we can efficiently estimate the symmetric difference, which is a metric. It is thus suited for nearest neighbour search.

### Spherical Distance

A similar technique is to project each element in each bag onto a $n$-dimensional unit sphere and then adding up all the resulting vectors. This can be thought of as adding the element randomly distributed simultaneously to each of the $n$ frequencies, instead of adding each element from the bag to one random frequency, like in the previous approach. Random sphere points can be easily constructed by assigning each vector element a uniformly distributed number in $[-1, 1]$ and then normalizing the vector such that its distance from the origin is $1$. The same vector has to be given to two elements that are equal.
If we have two bags $A$ and $B$ with $x$ elements in common and $y_A$ and $y_B$ different elements; $v_x$ denoting the vectors of those common elements, $v_A$ and $v_B$ those of the different objects and $a$ and $b$ representing the sum of all vectors of all elements in one bag, we have:

$$\begin{aligned}
a - b &= \sum_{i=0}^{x}(v_x[i]) + \sum_{i=0}^{y_A}(v_A[i]) - \sum_{i=0}^{x}(v_x[i]) - \sum_{i=0}^{y_B}(v_B[i]) && (4.25)\\
&= \sum_{i=0}^{y_A}(v_A[i]) - \sum_{i=0}^{y_B}(v_B[i]) && (4.26)\\
&= \sum_{i=0}^{\min(y_A, y_B)}(v_A[i] - v_B[i]) + \sum_{i=\min(y_A, y_B)}^{y_A}(v_A[i]) - \sum_{i=\min(y_A, y_B)}^{y_B}(v_B[i]) && (4.27)
\end{aligned}$$

The expected value of one unnormalized vector element is $E[v[i]_k] = 0$, the variance is $Var[v[i]_k] = \frac{1}{3}$. The expected value of a normalized vector element is $E[v(i)(k)] = 0$ and the variance $Var[v(i)(k)] = \frac{1}{3n}$. Thus the variance of the difference $Var[v_A[i]_k - v_B[i]_k] = \frac{2}{3n}$. With $y = 2\min(y_A, y_B)$ and $y_d = |y_A - y_B|$, we have $\frac{y}{2}$ differences and $y_d$ single elements. By again using the well known formula to calculate the variance from samples, we have:

$$Var[a-b] = \frac{y}{3n} + \frac{y_d}{3n} = \frac{1}{n}\sum_{i=0}^{n-1}(a[i]-b[i])^2 \tag{4.28}$$

$$\implies y + y_d = 3\sum_{i=0}^{n-1}(a[i]-b[i])^2 \tag{4.29}$$

$$= \sum_{i=0}^{n-1}(\sqrt{3}\cdot a[i] - \sqrt{3}\cdot b[i])^2 \tag{4.30}$$

If we incorporate those factors this results in the same distance measure as before:

$$D_b(A,B) = \sum_{i=0}^{n}(a[i]-b[i])^2 = ||a-b||_2^2 \tag{4.31}$$

Note that whereas the result for large differences is the same, this method is better than the previous histogram hashing method for small differences or small $n$. As an example, let to bags $A$ and $B$ differ in just one element. The chance that it increases the same frequency is $\frac{1}{n}$. If this happens the two bags will have distance zero, even if they are different. The chance that it happens here is smaller because we use $n$ random numbers per element. The chance that they are the same is negligible. This advantage is lost with larger differences as single elements become less relevant and the number of dimensions $n$ of the vectors is the restricting factor. The disadvantage of this method is that we have to generate $n$ random numbers per element, e.g. by $n$ different hash functions, whereas the previous method only required one, which could be a straightforward modulo operation of the hash of the element. For the data we tested with, $n \approx 20$ turned out to be a reasonable compromise between performance and quality. We use a pseudo-random generator seeded by the initial hash value to generate the $n$ different random numbers per element. With a simple pseudo-random generator, this is reasonably fast even for larger $n$.

**Error Analysis**

Being a sum of squares $D_b$ is chi squared($\chi^2$) distributed with $n$ degrees of freedom, if each component is distributed as $\mathcal{N}(0,1)$. In the first histogram hashing approach $U_A[i] - U_B[i]$ is Bernoulli distributed and thus approaches the normal distribution for a larger amount of grams. In the spherical distance approach $a - b$ is a sum of uniformly distributed random variables and for larger sums, i.e., lots of grams,

approaches a normal distribution as well. If

$$D_b = \sum_{i=0}^{n-1} X_i^2 \tag{4.32}$$

in the histogram hashing approach the variance $\sigma_B^2$ of $X_i$ is

$$\sigma_B^2 = \frac{y - \frac{y}{n} + y_d}{n} \tag{4.33}$$

and for the spherical distance it is

$$\sigma_S^2 = \frac{y + y_d}{n} \tag{4.34}$$

If we normalize the variances of the random variables to one we have:

$$\frac{1}{\sigma^2} D_b = \frac{1}{\sigma^2} \sum_{i=0}^{n-1} X_i^2 \sim \chi^2(n) \tag{4.35}$$

The mean and the variance of the chi squared distribution is:

$$\mu_{\chi^2} = n \qquad\qquad \sigma_{\chi^2} = 2n \tag{4.36}$$

It's probability density function is:

$$f(x, n) = \frac{x^{(n/2)-1} e^{-x/2}}{2^{k/2}\Gamma(k/2)} \tag{4.37}$$

With the Gamma function:

$$\Gamma(k) = \begin{cases} (k-1)! & for\ \lceil k \rceil = k \\ \sqrt{\pi}\frac{(2k-2)!!}{2^{(2k-1)/2}} & for\ \lceil 2k \rceil = 2k \end{cases} \tag{4.38}$$

The cumulative distribution function is:

$$F(x, n) = \frac{\gamma(n/2, x/2)}{\Gamma(n/2)} \qquad \gamma(n, x) = x^n e^{-x} \sum_{k=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a + n + 1)} x^n \tag{4.39}$$

With (4.35) and (4.36):

$$E[D_b] = \sigma^2 n \qquad\qquad Var[D_b] = 2\sigma^4 n \tag{4.40}$$

For the histogram hashing method the variance is therefore:

$$Var[D_b] = 2\frac{y - \frac{y}{n} + y_d}{n} \tag{4.41}$$

and for the spherical distance method:

$$Var[D_b] = 2\frac{y + y_d}{n} \tag{4.42}$$

As expected the variance of the distance measure decreases with larger $n$ and increases with a larger number of differences. With the help of the probability density function we can see in which specific bounds the distance will probably be in. Let's say $D_b(A, B) = 100$, $n = 20$ and we want to see the in which bounds the distance will be in with probability $.8$. We are looking for $x$ with $y = x/\sigma^2$:

$$F(y, n) = 0.1 \tag{4.43}$$

$$\implies x = 2 \cdot 6.22 \cdot \frac{100}{20} = 62.2 \tag{4.44}$$

$$F(y, n) = 0.9 \tag{4.45}$$

$$\implies x = 2 \cdot 14.21 \cdot \frac{100}{20} = 142.1 \tag{4.46}$$

To get $F^{-1}$ SciPy [21] (The *gammaincinv* function) has been used . So the probability that the distance is $62.2 \leq D_b(A, B) \leq 142.1$ is $80\%$ with $E[D_b(A, B)] = 100$ and $n = 20$. This expresses with numbers what we already knew: That we reduce the precision of the distance measure significantly by reducing the dimensionality of the feature vectors. We discard information in the feature vectors at random, which is obviously destructive. In the next section we present a weighting scheme which discards less relevant information.

### Weighting Grams

Lets say we have two trees represented by bags $A$ and $B$. $S_A$ and $S_B$ are bags representing sub-trees in $A$ and $B$. If we now have a gram $e \in A$ for which $e \notin B$ and we simply remove it from $A$ our distance measure between those sub-trees will be (again with $x = |A| \cap |B|$, i.e., the common elements of the multi-sets):

$$D_b(A, B) = |A| + |B| - x \tag{4.47}$$

$$D_b(A \backslash \{e\}, B) = |A| - 1 + |B| - x = D_b(A, B) - 1 \tag{4.48}$$

if $e$ is only once in $A$. Removing $e$ decreases the distance of all sub-trees containing $e$ in $A$ to any sub-tree in $B$. The nearest neighbours of sub-trees of $A$ in $B$ are thus unaffected. The reverse is not true. There could be two identical sub-trees in $A$, except that one contains $e$. The distance of a sub-tree in $B$ is then equal to both trees after $e$ is removed. Before removing $e$ the sub-tree containing that element was further away. Thus, this can potentially change the nearest neighbours in a detrimental way because removing that $e$ requires an edit operation, which the other sub-tree, not containing $e$, does not require.
Therefore, we have to save how many elements we removed and add that to our distance measure such that it remains the same (in this case one). If we record via

an additional dimension in the vector describing each sub-tree how many elements we removed and if $v_A$ and $v_B$ the vectors representing the sub-trees with $v_A(n)$ the number of elements removed in $A$ and $v_B(n)$ the number of elements removed in $B$ our distance measure would be

$$D'_b(A,B) = \sum_{i=0}^{n-1}(v_A(i) - v_B(i))^2 + |v_A(n) - v_B(n)| \tag{4.49}$$

We have seen with equations 4.43-4.46 that the variance of the distance can be unusually high and that therefore the distance measure can be nearly useless for small $n$ and large expected distances. There is no easy way to solve or diminish this problem without making assumptions about the underlying data.

One more general assumption we can make is that having less frequent grams in common is more significant than having frequent grams in common. If we look, for example, at HTML documents, two sub-trees having a *br* element in common are not that rare, while having a long text node in common which appears infrequently in the document is a strong indication of a correct mapping.

One frequently used method is to weight the elements according to their frequency combined with their rarity(*tf* $\cdot$ *idf* – term frequency times inverse document frequency) [41]. The thought being that we want to improve both recall and precision and weight the elements accordingly. The *tf* term should improve recall, i.e., emphasizes elements which occur frequently. This would, for example, give the element that occurs in $A$ and not in $B$ zero weight because it does not occur in tree $B$. The inverse document frequency improves precision, i.e., decreases the number of mapping candidates. It should punish elements which occur often throughout the tree such as the previous *br* element. If we view every sub-tree as a different "document", we should look at the number of times an element occurs with a different parent in the two trees.

The term frequency is then best defined as the minimum term frequency in each of the trees normalized to one, i.e., divided by the maximum occurring frequency. If the element $e$ occurs $c(e,A)$ times in $A$ and $c(e,B)$ times in $B$, we can define the term (or element) frequency as

$$\mathit{tf}(e) = \frac{\min(c(e,A), c(e,B))}{\max(\max_{k\in A}\{c(k,A)\}, \max_{k\in B}\{c(k,B)\})} \tag{4.50}$$

Let $dp(e,B)$ be the number of different parents an element $e$ has in tree $B$ and $\mathit{innerNodes}(B)$ the number of inner, non-leaf nodes in $B$, then the inverse document frequency is

$$\mathit{idf}(e) = \begin{cases} -\log \frac{\mathit{innerNodes}(B)}{\mathit{dp}(e,B)} & if\ \mathit{dp}(e,B) > 0 \\ 0 & else \end{cases} \tag{4.51}$$

The weight $w$ of each element is then $w(e) = \min(1, \mathit{tf}(e) \cdot \mathit{idf}(e))$. Each element is weighted before adding it to either a frequency in the hash histogram method, or

point in a hyperdimensional sphere in the spherical distance method. Simultaneously we save the weight change in $v_A(n)$, i.e., for each element

$$v_A(n) := v_A(n) + 1 - w(e) \tag{4.52}$$

As distance function Equation 4.49 can still be used.

## 4.5 Efficient Index Structures

The sub-trees are now represented by a $n$-dimensional vector and are arranged in a $n$-dimensional space imposed by the distance function. We can then use standard methods to build index structures to efficiently find nearest neighbours in such a space. We will focus on k-d trees, hierarchical k-means, and k-means locale shared hashing (KLSH). These methods have been shown to be useful in practice [29, 35, 36] and are simultaneously easy to implement.

### 4.5.1 k-d Tree

A classical index structure for multidimensional data is a k-dimensional tree structure (k-d tree). It is a balanced binary tree, where on each level a hyperplane separates the set of multidimensional points into two equally large sets. The hyperplanes are perpendicular to a depth-alternating axis, i.e., if the axis are represented by numbers $x, y, z, \ldots = 0, 1, 2, \ldots$ the axis is selected as $d \mod n$ with $d$ the current depth in the k-d tree and $k$ the dimensionality of the data points.
Let $D$ by a set of input data points having dimensionality $k$ for which we want to speed up nearest neighbour queries. The k-d tree is then constructed as follows:

**Algorithm 4.1.** *k-d tree construction*

1. *Set the current data points $D_c$ to $D$.*
2. *Set the current dimension to $d = 0$.*
3. *Sort all the current data points $D_c$ using the value in dimension $d$.*
4. *Select the median of the sorted points.*
5. *If the array only contains one data point, create a k-d tree leaf node – storing the data point – and return.*
6. *Create a k-d tree inner node storing the coordinates of the median. Store the other data points from the start of the sorted array to the median and from the median to the end of the array in the left and right child of that k-d tree node.*
7. *Set $d = (d + 1) \mod k$*
8. *Let $D_l$ be the first half of the sorted $D_c$ and $D_r$ the second half. Recursively go to 3., once with $D_c = D_l$ and once with $D_c = D_r$.*

An example for a two dimensional k-d tree build with this algorithm can be seen in 4.9 for $k = 2$. It shows that every inner node halves the data points in one dimension.

The $n$ nearest neighbours of a point can be found using following algorithm. It uses a priority queue *pq* where data points closer to the query point have lower priority, i.e., where the points furthest away from the query point are on top.

**Algorithm 4.2.** *k-d tree nearest neighbour query*

1. *Start with the root node as current node $l$.*
2. *If $l$'s coordinates are closer to the query point than the top of pq and pq already contains $n$ items, remove the top element from pq and add the current node's coordinates. If pq contains less then $n$ items always add $l$'s coordinates.*
3. *If $l$ is a leaf node return.*
4. *If the query point's coordinate in the current dimension ($d \mod k$, with $d$ the depth of $l$ in the k-d tree) is smaller than $l$'s coordinate in that dimension, recursively go to 2. with the left child as current node $l$, else go to 2. with the right child as current node $l$.*
5. *If the distance in the current dimension from query point to $l$'s coordinate in that dimension is smaller or equal than the top of pq, recursively call 2. with the other child as well.*

Afterwards, the $n$ nearest neighbours are in the priority queue.

As can be seen, the building of the k-d tree is dominated by the sorting operation and therefore runs in $\mathcal{O}(n \log^2 n)$. The runtime complexity of the $j$ nearest neighbour queries is less obvious. If the points are well distributed, the dimensionality of the data $k$ is fairly low and the recursion in 5. does not happen its runtime is $\mathcal{O}(j \log n)$. In degenerated cases, for example, if all the data points are at one coordinate, the

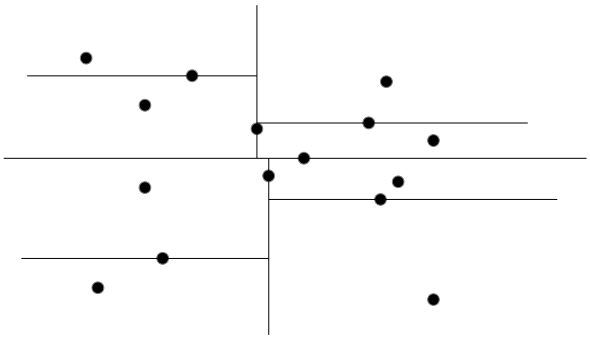


Figure 4.9: Stylized two-dimensional k-d tree with data points

query time goes up to $\mathcal{O}(j \times n)$. Other degenerated cases can be found in a k-d tree introduction by Moore [32]. With higher dimensionality $j$ nearest neighbour search always degrades to $\mathcal{O}(j \times n)$ and is thus not faster than a straightforward linear search. This is due to the curse of dimensionality and cannot be avoided [18]. Therefore, one of the following approximate nearest neighbour algorithms has to be used with data of higher dimensionality. Note that the exact algorithm can be transformed to an approximate algorithm by branching out only a limited amount of times in step 5 (Best bin first approach – see next section).

### 4.5.2 Best Bin First

By minimally modifying the k-d tree $n$ nearest neighbour query algorithm it can be transformed into an approximate nearest neighbour search. This is then called best bin first approach [5]. It works by introducing another priority queue during query time which has points closer to the query point on top. Then, instead of recursing in step five ,in the previous section, one adds the branch to this priority queue. Then, after it has run, we rerun the nearest neighbour search with points from the priority queue, taking closer points first. This is only done a limited amount of times. If we run it for all points in the priority queue, we would have the exact nearest neighbours. Heuristically running it only for close nodes, gives us the approximate nearest neighbours, but also a speed up.

### 4.5.3 Locale Sensitive Hashing

Locale sensitive hashing (LSH) is based on the idea of having a hash function which hashes nearest neighbours to same buckets. We already looked at this approach in connection with the *MinHash* algorithm at Section 4.4.3. Now, that we have vectors in $\mathbb{R}^n$ and euclidean spaces, we would have to use a euclidean version of LSH like E2LSH [11], where the points are projected onto random lines with the line segments being hash values, or by embedding the euclidean points into Hamming space and doing the LSH there [19].

It has been shown [36] that those kinds of LSH methods perform sometimes poorly because they do not adapt to the data distribution. Essentially they divide the space into cells with about same volume – each cell having a hash value. If the point density is low or high the algorithm does not react, i.e., shrink or expand the volume of the cell, returning too few or too many results. Because of this we did not use these methods and concentrated on the two following ones.

### 4.5.4 Hierarchical k-Means

Hierarchical k-means has been successfully used to cluster high dimensional data [35]. It works by recursively finding centroids for the data point clusters and then arranging those clusters into a tree structure.

### k-Means Clustering

This method partitions the data into $k$ Voronoi cells. Each cell is defined by the centroid (i.e., average coordinate) of all data points belonging to it. Data points, in turn, belong to the cell with minimal center distance. Given the data points $(p_1, p_2, \ldots, p_n)$ and looking for $k$ clusters $S = (S_1, S_2, \ldots, S_k)$ which partition the points and with respective mean $\mu_i$, the problem of finding a set of means which minimizes the distance from means to cluster points can be formulated as:

$$S_{min} = \arg\min_{S} \sum_{i=1}^{k} \sum_{p \in S_i} d(p, \mu_i) \tag{4.53}$$

with $d(x, y)$ the distance function. Solving this problem exactly is *NP*-hard. There is, however, a very popular iterative algorithm which converges fast to an approximate solution [28]. It works as follows:

**Algorithm 4.3.** *k-Means clustering*

1. *Set* $\mu_1 = p_1$.
2. *For* $i = 2 \ldots k$*: Search* $(p_1, \ldots, p_n)$ *for the point* $p_l$ *for which the distance to all the previously defined means* $(\mu_1, \ldots, \mu_{i-1})$ *is maximal. Set* $\mu_i = p_l$.
3. *Iterate through* $(p_1, \ldots, p_n)$ *and attach the the point to the partition* $S_i$ *with nearest mean, i.e., where* $d(\mu_i, p_l)$ *is minimal.*
4. *Recalculate each mean* $(\mu_1, \ldots, \mu_k)$ *with* $\mu_i = \frac{1}{|S_i|} \sum_{p \in S_i} p$
5. *Go to 3. or terminate after a sufficient amount of iterations.*

As another finishing criterion convergence (the means do not change any more), could be used. As each iteration involves iterating over all data points twice, the $k$-means are calculated in $\mathcal{O}(n)$ with a constant number of iterations (our default is 20).

### Hierarchical k-Means

In HKM the clustering is done recursively till the number of data points in each cluster is below a certain threshold $t$. The created clustering is then arranged in a tree with the branch-out factor being the number of clusters $k$ as children of each node. Let $D$ by a set of input data points, then

**Algorithm 4.4.** *Hierarchical k-means clustering*

1. *Set the current data points* $D_c$ *to* $D$.
2. *Cluster* $D_c$ *using* $k$*-means clustering.*

3. *Recursively go to 2. for the data points in clusters with size greater than $t$ and if the current depth of the tree is smaller than the max depth. Set $D_c$ to the points in the cluster.*

The max depth abortion criterion is necessary to prevent endless recursion in some degenerated cases, like when all data points have the same coordinates. A resulting HKM clustering can be seen in Figure 4.10.

Approximate $j$ nearest neighbour queries on HKM trees work by recursively finding the cell the query point belongs to. Usually one selects the threshold $t$ such that the number of points in a cell is much larger than $j$ so that it is enough to examine one of them in most cases. If there are less than $j$ points in a cell, one can either return fewer neighbours, or examine sibling cells, as well. As having cells with fewer than $t$ points indicates highly clustered data points, we return fewer nearest neighbours in those cases. With a constant branch-out factor and if the data can be clustered, this query does then run in $\mathcal{O}(\log n)$ on average.

### 4.5.5 KLSH

K-means locale sensitive hashing [36] uses $k$-means to convert space coordinates into hash values. Since $k$-means is sensitive to the distribution of data points, this is better than directly hashing the vectors. K-means almost never converges to the global optimum with high dimensional data. The result is a local minimum and the initial assignment of the means determines which minimum it converges to. Therefore, by randomly choosing the initial means $\mu$, different k-means clusterings can be generated. Nearest neighbours then have a higher chance of being in the same cluster in each of the clusterings than other point pairs.
As an example, we have $1000$ data points. We cluster those into $10$ partitions with $k$-means ($k = 10$) with two different random initializations. If a data point $p_i$ is in partition $S_{1,o}$ and $S_{2,p}$ for the two different clusterings, then its LSH value is $h(p_i) = k \cdot o + p$. This would give us $100$ different hash buckets with $10$ data points per bucket on average. For finding the $j$ nearest neighbours, we first look to which partitions the query position would belong (by finding the closest cluster means $\mu$) and then look up the approximate nearest neighbours in the table and return the closest ones. If there are not enough points in the table, less then $j$ nearest neighbours are returned. By selecting $k$ and the number of partitions such that the average number of points per hash bucket is a lot larger than $j$ this can be avoided in most cases. If it does happen, one can fall back to a linear scan or simply return fewer nearest neighbours, which is what we do.

## 4.6 Matching with Feature Vectors and Index Structures

Once we have calculated the feature vectors for each sub-tree and put them into a data structure which enables fast $k$-nearest neighbour searches we can find mappings for previously unmapped sub-trees. We iterate in pre-order over tree *B* and

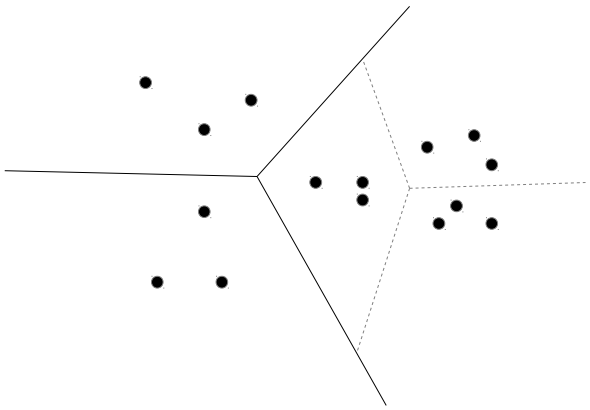


Figure 4.10: Stylized example 2D HKM clustering with $t = 3$ and $k = 3$. The second level cluster boundaries are gray. The dots are data points.

execute a $k$-nearest neighbour[4] search with the feature vector of each unmapped node as input. We still have to decide to which, if any, of the $k$ nearest neighbours we want to map to, i.e., we have to figure out if a nearest neighbour is a false positive. This has to be done in $\mathcal{O}(\log n)$ to remain within the overall runtime bounds. We use an iterative deepening top-down matching for that. Let $S_B$ be the unmapped sub-tree and $N$ the set of $k$ nearest neighbour sub-trees. For each tree $S_A \in N$, we then perform the iterative deepening top-down matching returning a cost. The iterative deepening top-down matching is only allowed a constant amount of node label comparisons $c_{max}$ and works as follows:

**Algorithm 4.5.** *Iterative deepening top-down matching*

1. *Set the max depth $d_{max} = 1$.*
2. *Perform a top-down matching of $S_A$ and $S_B$ using Algorithm 2.1 with max $c_{max}$ node label comparison and max distance $d_{max}$ from sub-tree root.*
3. *If the number of comparisons performed is lower than $c_{max}$ and the max distance from sub-tree root prevented the algorithm from visiting some nodes set $d_{max} = d_{max} + 1$ and go to 2. else return the result of the top-down matching.*

We then map $S_B$ to the tree in $N$ with minimum cost, approximated by the iterative deepening top-down matching, if this minimum cost is smaller than the cost of inserting $S_B$. If we have $\mathcal{O}(\log n)$ look-ups in the $k$-nearest neighbour index structure, we can thus have $\mathcal{O}(n)$ unmapped nodes and still remain within $\mathcal{O}(n \log n)$.

Figure 4.11 gives an overview over the steps involved in the feature vector matching.

[4] $k$ is a parameter to our method. The default is 10.

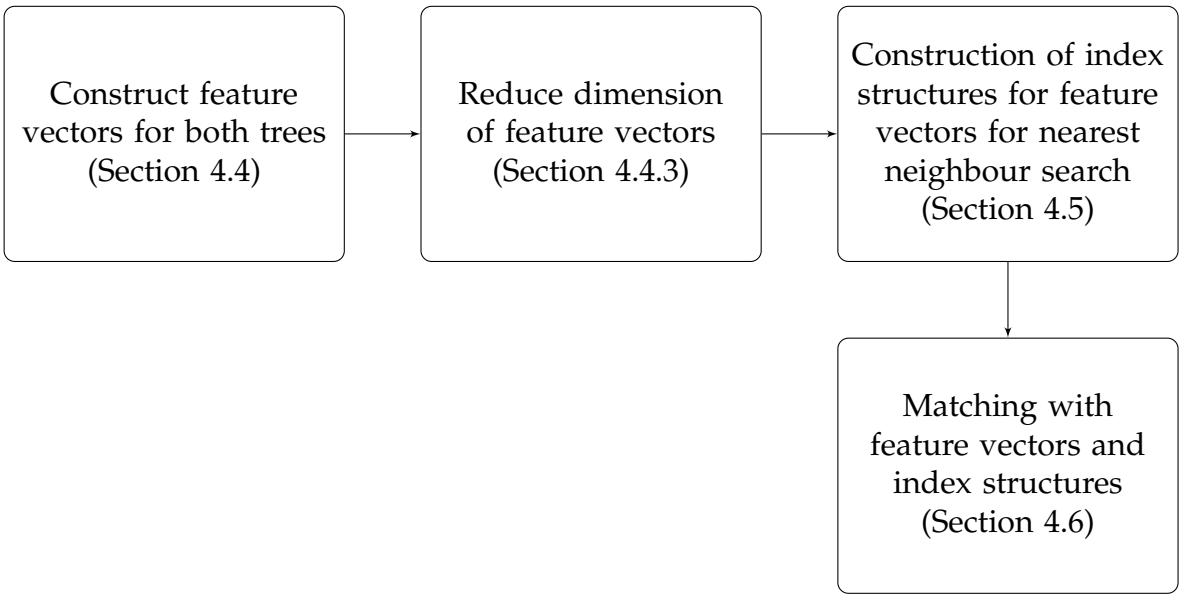



Figure 4.11: Overview of the steps involved in the feature vector matching

# 5 Edit Script Calculation

This chapter shows the edit script calculation in its entirety, i.e., how the method presented in the previous chapter is embedded within the whole method, how the different input formats are read and internally represented and how the final edit operations are generated from the matching and how those edit operations can be saved.

Following steps are necessary for the edit script calculation:

1. Reading and parsing of the tree into an appropriate in-memory tree structure and generation of a tree index structure for fast queries on tree and tree manipulation.

2. Calculating an approximately cost-minimal matching as described in Chapter 4.

3. Construction of a tree edit script in an appropriate output format using the previously calculated matching.

## 5.1 Reading and Parsing Tree Inputs

Depending on the application different tree inputs have to be handled. Some applications may have their tree structures already in memory, however, then this step may be skipped. This section will give a brief overview over supported input formats, their purpose and how they are represented in the internal tree structure.

### 5.1.1 XML

XML is an important document exchange standard. It is used by a lot of applications, for example Microsoft Office, as well as for configuration files, protocols, etc. Parsers are readily available because of its widespread use. XML can be seen as a representation of a tree with some special properties:

- There are different types of tree nodes: Element nodes with attributes, text nodes and comments. It does not make sense to *move* or *rename* them into each other. For example, changing an element node to a text node does not make sense.

- The order of elements may be important (this depends on what kind of data is stored in the XML file) while the order of attributes is always insignificant.

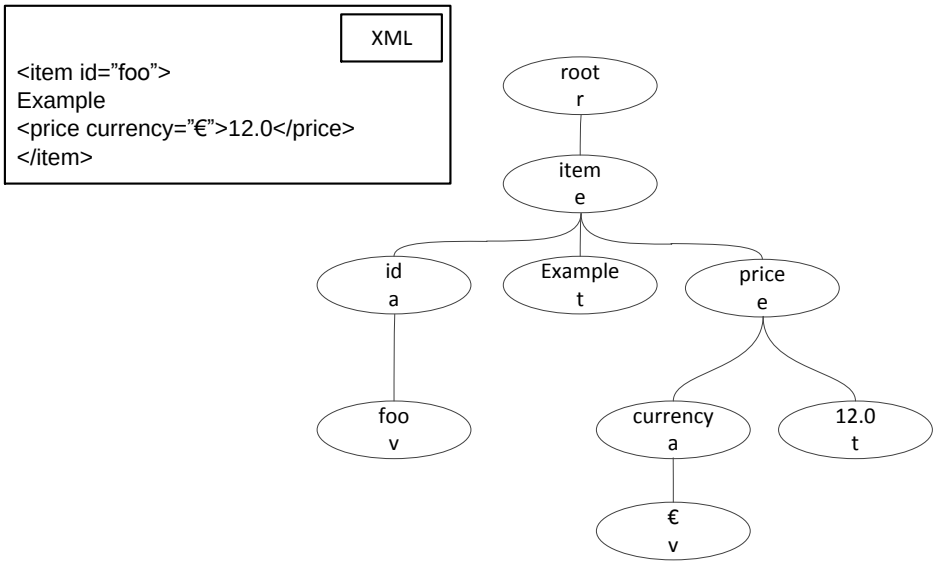



Figure 5.1: Internal representation of an example XML file

- There is no separation between text nodes. E.g., there may not be two text nodes following each other, as those would be interpreted as one continuous text node.

How an XML file is internally represented as a tree can be seen in Figure 5.1. To make this representation reversible and to distinguish for example attributes from elements, every node has a label and a type. In this example the types are *e* for elements, *t* for text nodes, *a* for attribute names and *v* for attribute values.

### 5.1.2 HTML

Having an HTML parser enables the calculation of differences between versions of web pages. The parser must be robust and understand different HTML versions as web pages are often not as error-free as XML files used for data exchange. Our implementation uses *htmltidy* [38] to read HTML files and to fix errors browsers would be able to fix. The internal representation is the same as the one for XML mentioned in the previous section.

### 5.1.3 JSON

JSON is a straightforward hierarchical data exchange format, popularized by its easy use in JavaScript (it is the JavaScript object notation). It consists of objects, which are key-value pairs, and arrays, which are lists of values. A value can be either a string, number, object, or array. An example JSON file and its internal representation as a tree can be seen in Figure 5.2. The types are *a* for array elements, *k* for keys and *v* for strings and numbers.

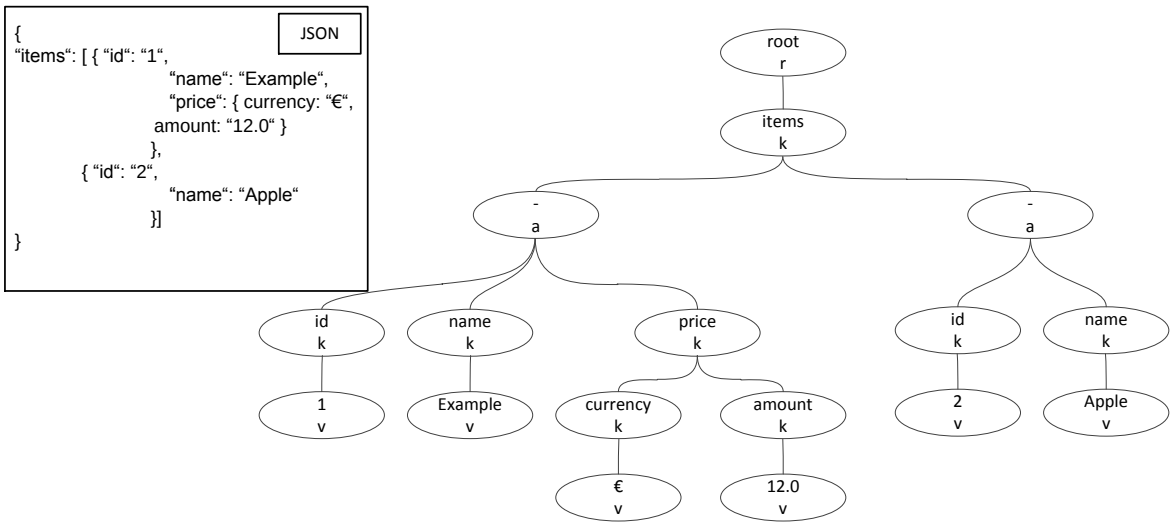



Figure 5.2: Internal representation of an example JSON file

## 5.2 Edit Script Generation

We transform the matching generated using the method described in Chapter 4 into an edit script. Some applications prefer the matching for further processing, however. In this case we are then already finished after we found the matching and need not compute the edit script. Other applications require the script and thus necessitate this step.
We already established that generating the tree edit script involves applying the tree edit operations on one tree and thus requires the tree implementation to be sufficiently fast when executing those operations (Section 5.2.2). Let $A$ and $B$ be the two trees. Let $M$ be the matching that maps nodes from tree $B$ to tree $A$. The algorithm to calculate the edit script works as follows for *insert*, *move* and *rename*:

**Algorithm 5.1.** *Edit script generation*

*Iterate in pre-order over tree $B$. Let $n$ be the current node:*

1. *If $n$ has no mapping in $M$, insert $n$ as child of the mapped node of $n$'s parent ($M(parent(n))$) and emit the* insert *operation. Map $n$ to the newly inserted node in tree $A$. Let $h$ be the node with label "H" in Figure 5.3. $h$ does not have a mapping. It has to be inserted below the node with label "A" which is the node the parent of $h$ is mapped to.*

2. *If $n$ has a mapping in $M$ and the parent of $n$'s mapped node is not the parent of the node mapped to $n$ ($parent(M(n)) \neq M(parent(n))$), move the mapped node in tree $A$ ($M(n)$) such that it is a child of the mapping of $n$'s parent $M(parent(n))$. Emit the* move *operation. Let $e$ be the node with label "E" in Figure 5.3. The sub-tree rooted*

*in $e$ is moved in Figure 5.3, because the node that the parent of $e$ is mapped to is not the node the parent of $e$ is mapped to. $M(e)$ is then moved below the node the parent of $e$ is mapped to.*

3. *If the label of $n$ is different from the label of the matching node ($l(M(n)) \neq l(n)$) , change the label of the node being mapped to to that of $n$ and emit the* rename *operation. This has to be done for the node with label "C" in Figure 5.3.*

As can be seen, this method requires the guarantee that the currently visited node's parent has a mapping. As we iterate over the tree in pre-order and the currently visited node is guaranteed to have a mapping after the steps are executed, this is not an issue.

For *copy* operation support we need to save which nodes in tree $A$ have already been moved. If we encounter a mapping normally a move (step 2.) with the node being moved already, i.e., the node was already involved in a move operation in step 2., the node is copied instead of moved. The copy operation's parameters are the same as those for the move operation (see 2.). After the copying, all the node's descendants in tree $B$ need to be mapped to the new copy in tree $A$, which previously did not exist. This is done by constructing $M^{-1}$ for the sub-tree to be copied. After the copy is created in $A$ we iterate over it and if a node has a mapping in $M^{-1}$ to a node in $B$, we can adjust the mapping of this node in $B$ to the correct copy in $A$. This is shown Figure 5.4, where a copy of the sub-tree is created. Then the mappings of nodes in tree $B$ are adjusted to the new sub-tree in $A$.

After this iteration in pre-order over tree B, the only difference between tree $A$ and $B$ is that tree $A$ still has some nodes that have been deleted and are missing in $B$. In Figure 5.3 this would be ,for example, the node with label "I". Iterating over tree A and emitting a *remove* operation for every node without mapping in $M^{-1}$, removes those nodes and makes tree $A$ and $B$ isomorph. We do not want to delete those nodes earlier, because then we would not be able to support deleting whole sub-trees, as such a sub-tree could contain a node which is moved. By removing at the end we now have the guarantee that if a node in $A$ has no mapping, the whole sub-tree rooted in that node can be safely removed. Note that if sub-tree removes are not supported, it is trivial to transform a sub-tree remove to single removes simply by iterating in post-order over every removed sub-tree and emitting a *remove* for every node.

### 5.2.1 Reordering Nodes

If the order of sibling nodes is important, we have to additionally reorder nodes which are not yet correctly ordered. An example where we have to do that is shown in Figure 5.5. The trees $A$ and $B$ are now identical, except where the order of sibling nodes differs. We have a bijective mapping between the two trees, i.e., every node in $A$ can be mapped to one node in $B$. Using this mapping, we can iterate in pre-order over tree $A$ and reorder the siblings using the order found in tree $B$. Every node in $A$ gets the pre-order number of the node being mapped to in $B$ as its order

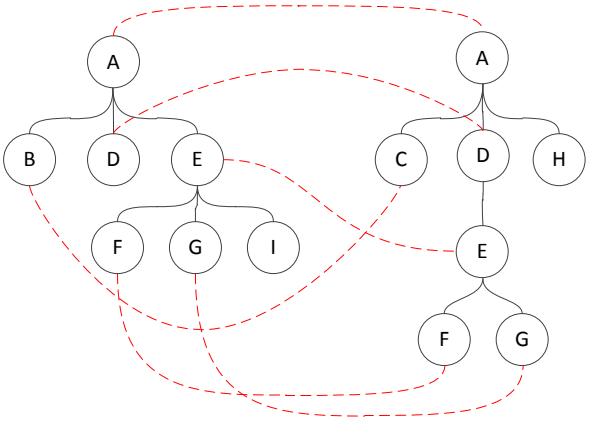



Figure 5.3: Example matching for which we want to generate a edit script.

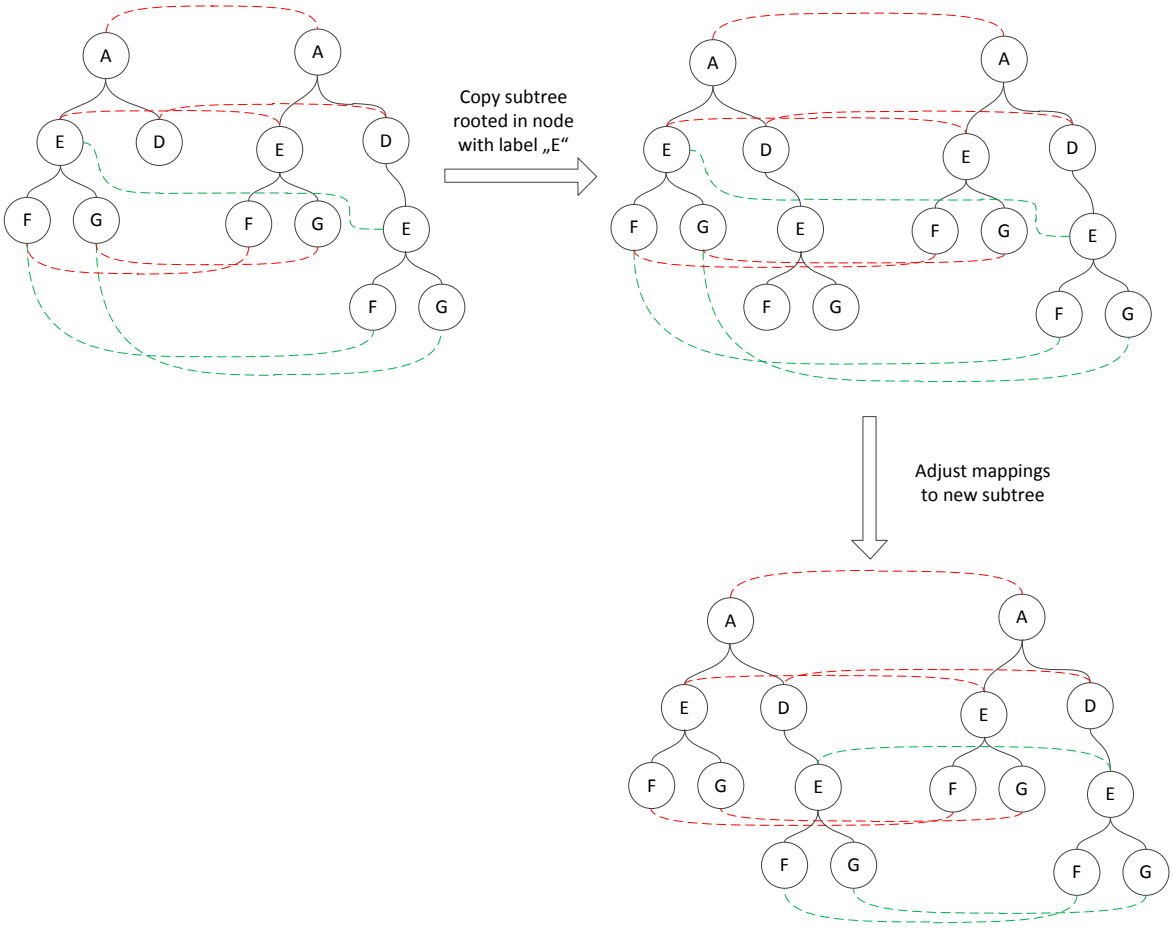



Figure 5.4: Mappings to copies have to be adjusted to the new copy.

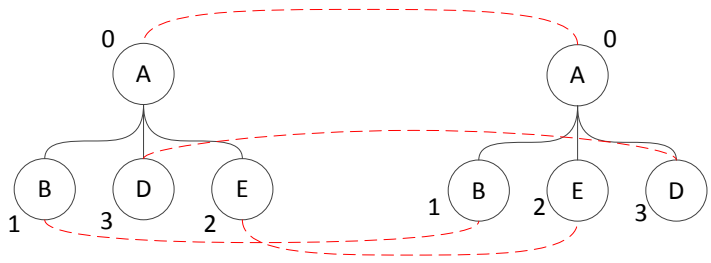



Figure 5.5: Matching with necessary reordering. Additionally to the matching, pre-order numbers are shown for tree $B$. The numbers on the left are the pre-order numbers of the node being mapped to on the right.

number. In Figure 5.5 those order numbers are shown next to the nodes. Then we reorder the sibling using this number. We do this reordering by first finding the longest increasing sub-sequence using the algorithm described in [14] which is attributed to Knuth and runs in $\mathcal{O}(n \log n)$. It finds the last longest increasing sub-sequence. In the example, with input $\{1, 3, 2\}$ it would thus find $\{1, 2\}$ as the largest increasing sub-sequence and not $\{1, 3\}$ because it is not the last one. After this step, we can go through the siblings again and put all the nodes which are not in the found sequence to their right place, such that we then have siblings with strictly increasing order number. In the example this would be done by moving the node *D* to the position after *E* because 3 is not in the longest increasing sub-sequence.
It is clear that this leads to the minimal amount of *move* operations for reordering. The largest increasing sub-sequence comprises all the nodes which do not have to be reordered. All the nodes not in this sequence have to be moved somewhere else, to be part of the increasing sequence. Because we can only move one node at a time, we require exactly one move operation per out-of-sequence node to correctly order all siblings.

### 5.2.2 Tree Representation

For many edit script output formats we actually have to perform all the edit operations in the script on one tree to be able to properly address nodes involved in change operations. The tree representation should therefore be able to perform all the tree edit operations as fast as possible. This is a complex requirement – the tree implementation has to perform, for example, $insert$, $remove$ and $move$ in $\mathcal{O}(\log n)$ for every kind of tree, even degenerated ones while retaining the ability for sufficiently fast queries (such as parent queries).
Additionally the tree implementation needs to allow fast iteration in pre- and post-order over the tree as well as iteration over every child of each node because use those kinds of iterations are often used in the matching step in Chapter 4. For those output formats which have paths in operations, such as XPath, a fast path calculation method for nodes is required. This usually involves calculating the child rank (i.e., the index of a node below its parent) and ascending recursively to the parent

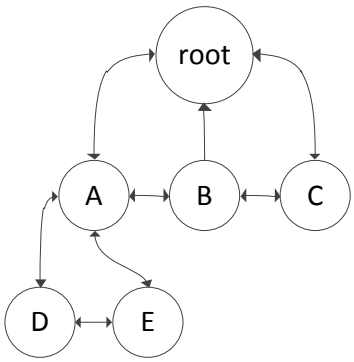



Figure 5.6: Pointers between nodes in the tree implementation

up to the root node of the tree.

A relatively straightforward way to implement trees is with parent and sibling pointers: Every node has a pointer to its parent, previous and next sibling, first child and last child (See Figure 5.6). The necessary operations then are easy to implement as well (with $n$ denoting the number of nodes in a tree, $c$ the maximal number of children of a node in a tree and $h$ the depth of the tree):

| Operation | Implementation | Time Complexity |
|---|---|---|
| Iterate over $c$ children | Use pointer to first child. Then use the pointers to the next siblings. | $\mathcal{O}(c)$ |
| Iterate in pre-/post-order over tree | Pre-/Post-order can be implemented using the child iterator via using recursion or a stack data structure. | $\mathcal{O}(n)$ |
| Get node parent | Use parent pointer | $\mathcal{O}(1)$ |
| Subtree-Insert | Same as inserting an item into a double linked list ($\mathcal{O}(c)$) plus updating the sub-tree sizes of all the node's ancestors ($\mathcal{O}(h)$) | $\mathcal{O}(h)$ if inserting in front or back, else $\mathcal{O}(c+h)$ |
| Subtree-Remove | Same as removing an item from a double linked list ($\mathcal{O}(1)$) plus updating the sub-tree sizes of all the node's ancestors ($\mathcal{O}(h)$) | $\mathcal{O}(h)$ |
| Move | Combined *Subtree-remove* and *insert* | $\mathcal{O}(h)$ |
| Child rank | Iterate over children of parent | $\mathcal{O}(c)$ |

With this implementation, generating paths requires $\mathcal{O}(n)$ time. With $d$ changes between two trees this causes a runtime complexity of $\mathcal{O}(nd)$ in the edit script generation step. Path generation could be sped up by using skip lists or trees to increase the performance of the child rank operation. We did not do this, so there is certainly room for improvement there. It is, however, near optimal in the first few steps of

the algorithm, where no changes to the trees are made and the complexity of child-, pre- and post-order iteration, as well as the parent look-up, is important. This implementation thus is a good choice if there are few changes, if we only need a mapping between trees and not a tree edit script or if the tree is not degenerated.

## 5.3 Output

As the edit script is just a set of operations with parameters, we can use any language which supports expressing this as output language. It is also beneficial for testing and implementation purposes that the output is human readable. Since we already have XML and JSON support and since for those data formats parsers are readily available, we decided to output the set of operations in those formats. To not invent another standard, names and parameters are as for DiffXML by Mouat et al. [33]. Example output for XML can be seen in the Listing 2 and for JSON in Listing 3. The specification of this *Delta Update Language* (DUL) can be found on the homepage of DiffXML [34].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<delta>
  <insert nodetype="1" parent="/Workbook[1]/Attributes[1]"
          childno="4" name="Test" />
  <move node="/Workbook[1]/Attributes[1]/Attribute[3]"
          parent="/Workbook[1]" childno="2" />
  <update node="/Workbook[1]/Attribute[1]/value[1]/node()[1]"
          charpos="1" length="4">TRUE</update>
  <copy node="/Workbook[1]/Attributes[1]/Attribute[1]"
          parent="/Workbook[1]" childno="3" />
  <delete node="/Workbook[1]/Geometry[1]" />
</delta>
```

Listing 2: Example XML patch file

```
[
{ "operation": "move",
      "node": "Workbook.Attributes.Attribute[2]",
      "parent": "Workbook.Summary", "childno": 1 },
{ "operation": "update", "node": "Workbook.Summary[0]",
      "text": "Attribute" },
{ "operation": "update", "node": "Workbook.Summary[0].value",
      "text": "TRUE" },
{ "operation": "copy", "node": "Workbook.Summary[0]",
      "parent"="Workbook.Sheets", "childno": 1 },
{ "operation": "insert", "nodetype": "1",
      "parent": "Workbook.Sheets", "childno": 0,
      "key": "value", "value": "TRUE" },
{ "operation": "delete", "node": "Workbook.Summary.Item[0]" }
]
```

Listing 3: Example JSON patch file

# 6 Evaluation

## 6.1 Comparison of Feature Vector Index Structures

In order to find out which of the feature vector index structure candidates presented in Section 4.5 is most suited we tested them with test data. We generate that data by taking random sub-trees from various freely available XML files [46], taking sub-trees having at least 10 nodes and at most 100 nodes. For larger sub-trees the runtime of the benchmarks would be prohibitively long. For each sub-tree we then generated a feature vector using pq-grams and the random sphere points methods. Additionally each tree is modified randomly, and another feature vector is calculated. We then measured performance, recall and precision with both HKM and KLSH as well as performance for the k-d tree and a linear search with the feature vectors of the modified trees as query points.

Table 6.1 shows the respective precision, average distance of the query point to nearest neighbours, recall, runtime and setup time for the different methods. Precision denotes the fraction of returned nodes which is a correct nearest neighbour, i.e., the nearest neighbours returned by the linear scan or k-d tree method (exact solution). Therefore, a precision of $0.5$ denotes that 5 of the 10 approximate nearest neighbours were correct nearest neighbours. The other returned points can still be close to the query point. Recall shows here how many of the points were scanned linearly. The setup time is the time needed to build the data structure. The runtime is the time needed for 10000 different $k$-nearest neighbour ($k = 10$) queries on 10000 different points. The points were generated as specified above. The number of dimensions per point was $d = 20$. The parameters for the KLSH scheme were $j = 10$ for the $j$-means with two $j$-means runs (Exactly as in the example in the previous section 4.5.5). HKM was run with $j = 10$ for the $j$-means, as well. Clusters were subdivided as long as they exceeded $200$ data points. The maximum depth was $5$. The precision and recall are averaged over all 10000 queries. The run and setup time is the accumulative time over all queries. Both of them are averaged over 20 runs on identical hardware (Core i5 M460). Table 6.2 shows the precision, average distance of the query point to nearest neighbours, recall, runtime and setup time for the different index methods for the same parameters except with 20000 different $k$-nearest neighbour queries and 20000 random trees. The precision of HKM is around $0.5$ in both cases. As mentioned the HKM method has parameters. With those parameters we could increase, e.g., the number of recalled nodes and thus trade-off an increased precision or shorter setup time for a longer query time. How the parameters are selected in detail, obviously depends on the runtime budget of the application using the method presented in this thesis. The parameters selected

are just a starting point which does work well for our test data. Similarly for the different index structures: Those are a parameter to our method, as well. Thus, if the runtime budget of the application using the method is high, one could use the linear scan method to find nearest neighbours.

Figures 6.1-6.6 show results for a varying amount of data poinnts and number of dimensions for those data points. All the non varying variables and the test data set were the same as for Table 6.1 and Table 6.2. Every result is the average of 10 runs. Figure 6.1 and 6.2 show that the precision of HKM is better than that of the BBF method and does not decrease exponentially with increasing number or dimensionality of the data points. Figure 6.3 and 6.4 show that the setup time of all methods is linear to the number of dimensions and the number of data points. Figure 6.5 and 6.6 show that the query time for HKM is much lower than for the other methods and that the query time of the KLSH method is higher for a large dimensionality or number of data points and increases faster than the other methods.

As it turns out, the choice between feature vector index structures is thus relatively clear. The k-d tree performance is worse for a high number of dimensions (e.g., 20), than the performance of the linear search. The k-d tree with best bucket first nearest neighbour has a lower precision than HKM while having higher query runtimes. The performance of KLSH, while in some cases better than HKM, is unpredictable, as it sometimes puts too many points into one bucket which causes a near linear query time. HKM, on the other hand, handles such cases well if the maximal depth of the HKM tree is high.
As an example, we had a case where there were 10000 points with two of them being outliers. Initializing KLSH with $k = 2$ created one cluster for the two outliers and one cluster for the 9998 other points. Repeating the $k$-means algorithm with a different initialization did not change this. As a result, only two buckets were used one of which contained 9998 points. Therefore, most queries resulted in a linear scan of those 9998 points. HKM on the other hand, while doing the same clustering on the first level, continued clustering the 9998 points on the second level, where they could be separated into clusters of roughly equal size. This is not anecdotal

| Method | Precision | Average dist | Recall | Runtime(ms) | Setup time(ms) |
|---|---|---|---|---|---|
| Linear scan | 1 | 3.43 | 10000 | 6665 | 0 |
| K-d tree | 1 | 3.43 | - | 15244 | 7.7 |
| BBF-Tree | 0.41 | 3.89 | - | 652.1 | 7.7 |
| KLSH | 0.77 | 3.58 | 790 | 360.95 | 131 |
| HKM | 0.52 | 3.77 | 120 | 140.5 | 270.3 |

*Precision*: Average proportion of found nearest neighbours to correct, i.e., found in the exact result, nearest neighbours.
*Average dist*: Average distance of query point to found nearest neighbours.
*Recall*: Average number of points which were scanned lineararily during query time.

Table 6.1: $k$-nearest neighbour benchmark results for different index data structures for 10000 points

| Method | Precision | Average dist | Recall | Runtime(ms) | Setup time(ms) |
|---|---|---|---|---|---|
| Linear scan | 1 | 3.32 | 20000 | 26437 | 0 |
| K-d tree | 1 | 3.32 | - | 60768 | 18.6 |
| BBF-Tree | 0.38 | 3.79 | - | 1332.2 | 19.3 |
| KLSH | 0.80 | 3.45 | 1814.35 | 1500.6 | 261.5 |
| HKM | 0.50 | 3.63 | 124.95 | 321.2 | 612.65 |

Table 6.2: $k$-nearest neighbour benchmark results for different index data structures for 20000 points

evidence, this is a failed test case in which KLSH failed to speed up the nearest neighbour search, even though the data can be clustered. It exposes a fundamental flaw in the KLSH method which does increase the runtime for the matching step to $\mathcal{O}(n^2)$ from $\mathcal{O}(n \log n)$ on average.

Since the BBF method is worse than HKM, except for setup time, for which HKM makes up by having a much better query performance, the KLSH method has this fundamental flaw and the k-d tree and linear scan having a prohibitive runtime, we chose the HKM index structure as default and use it as index structure while evaluating the overall approach in the remainder of this chapter.

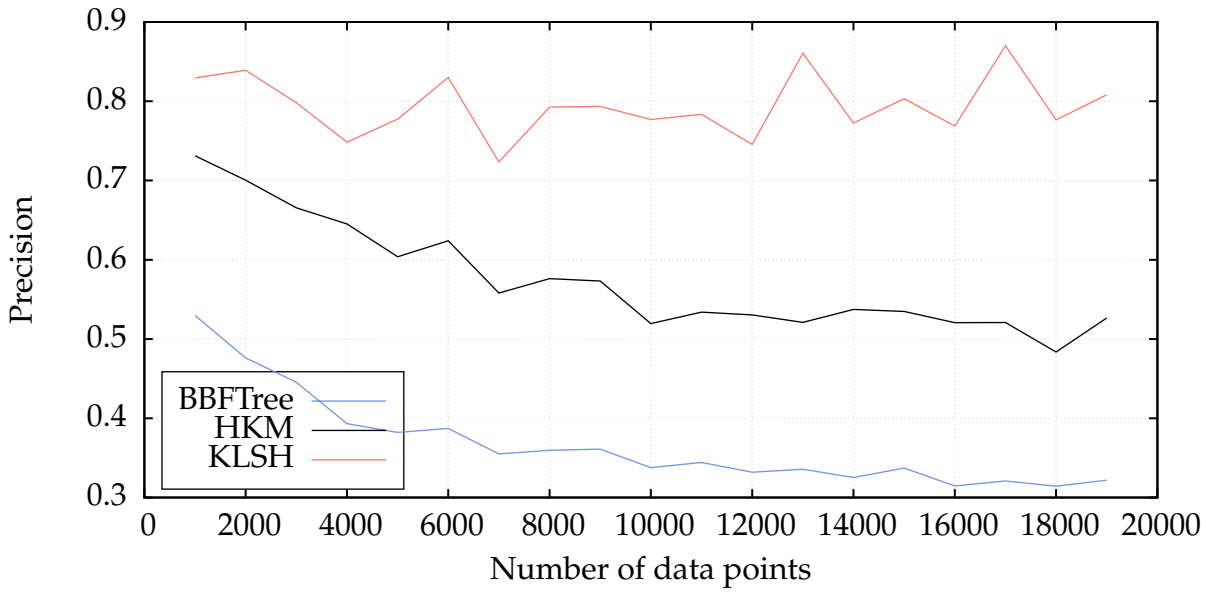



Figure 6.1: Feature vector index structure precision with varying number of data points

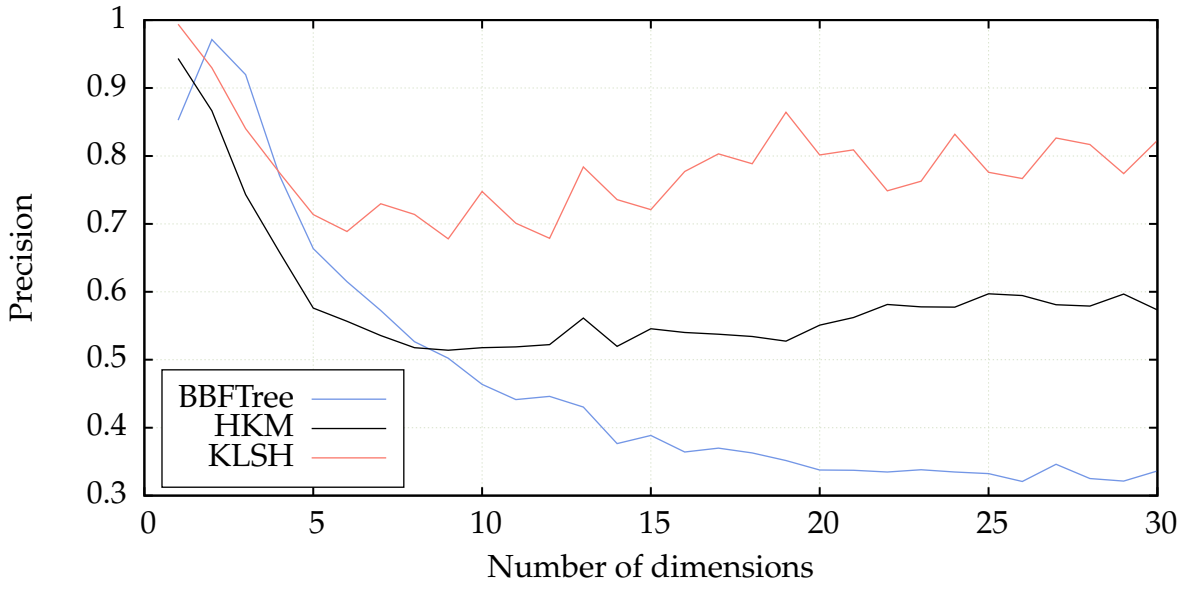



Figure 6.2: Feature vector index structure precision with varying data point dimension

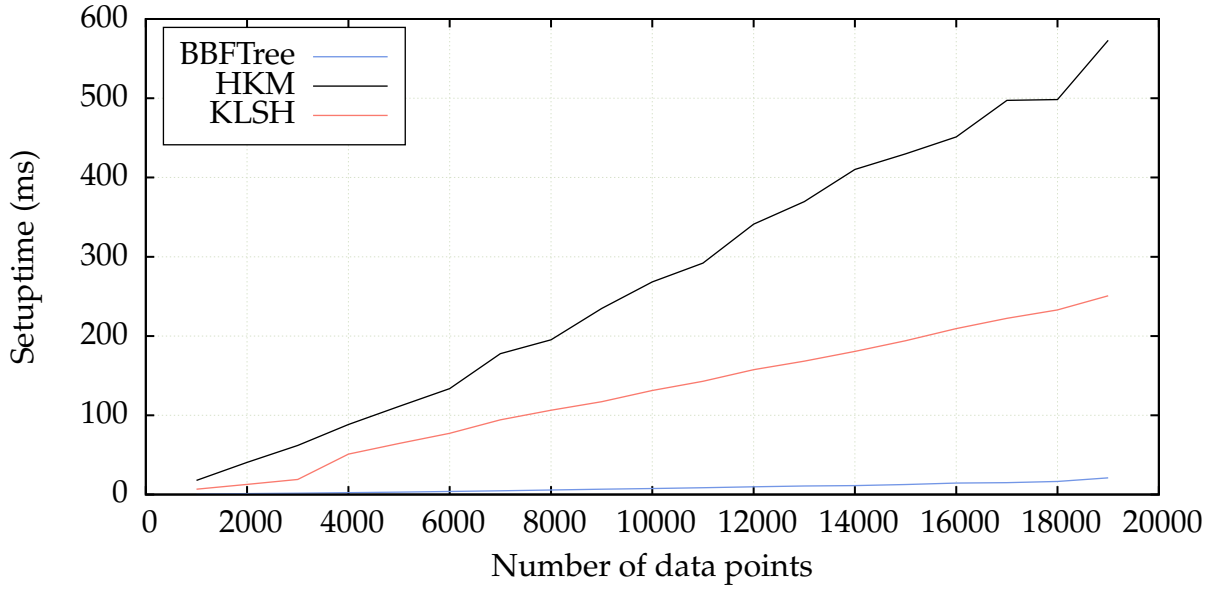



Figure 6.3: Feature vector index structure setup time with varying number of data points

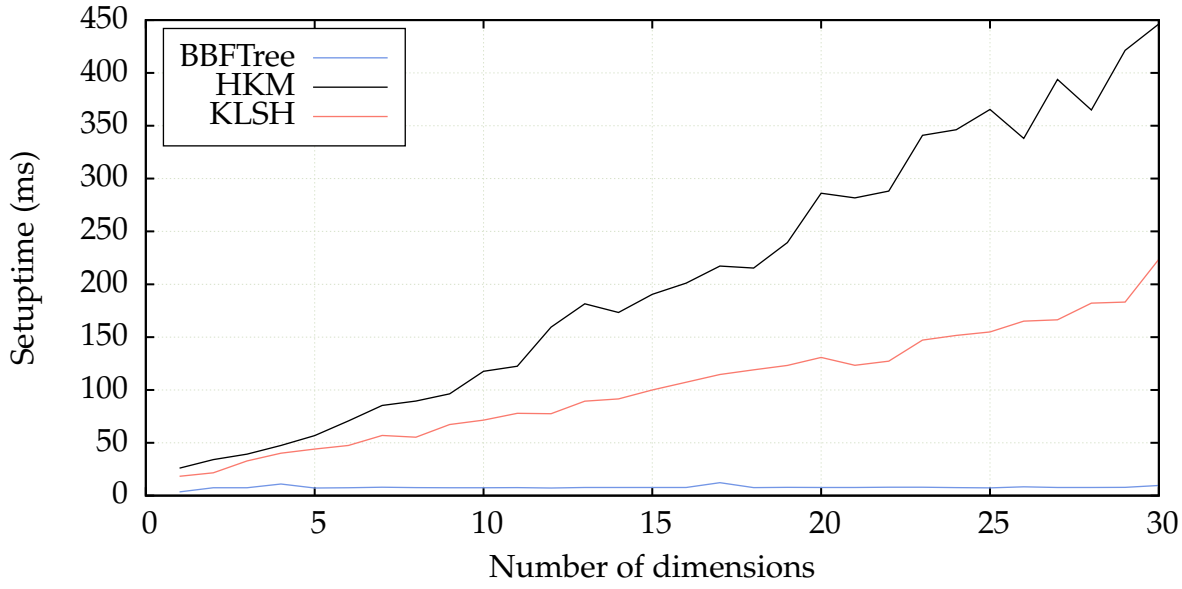



Figure 6.4: Feature vector index structure setup time with varying data point dimension

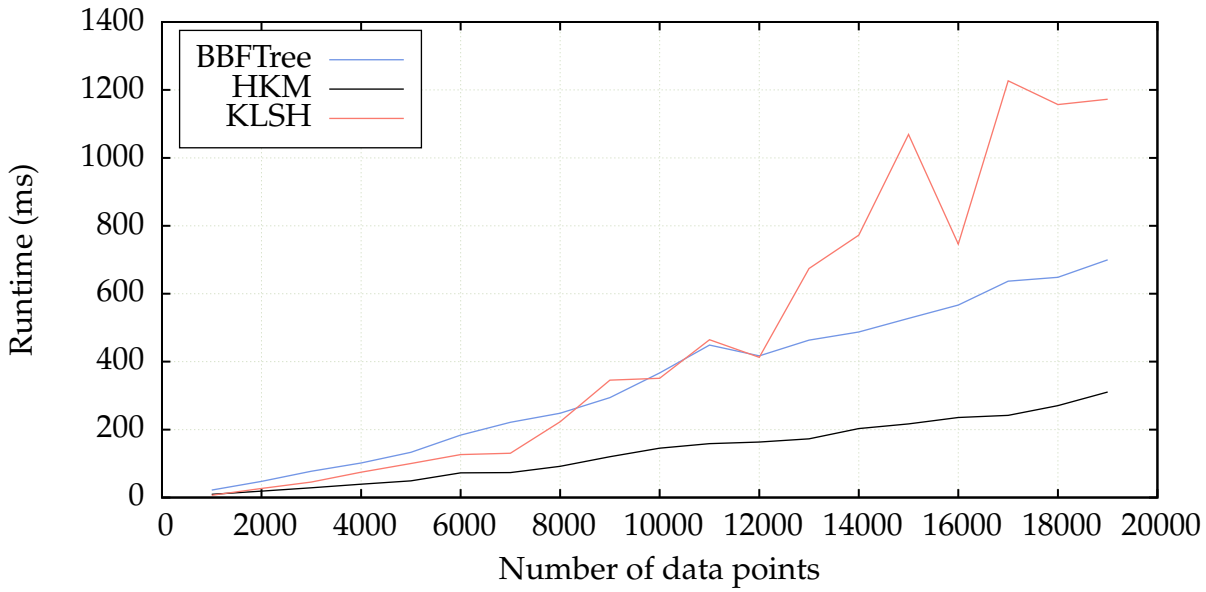



Figure 6.5: Feature vector index structure query runtime with varying number of data points

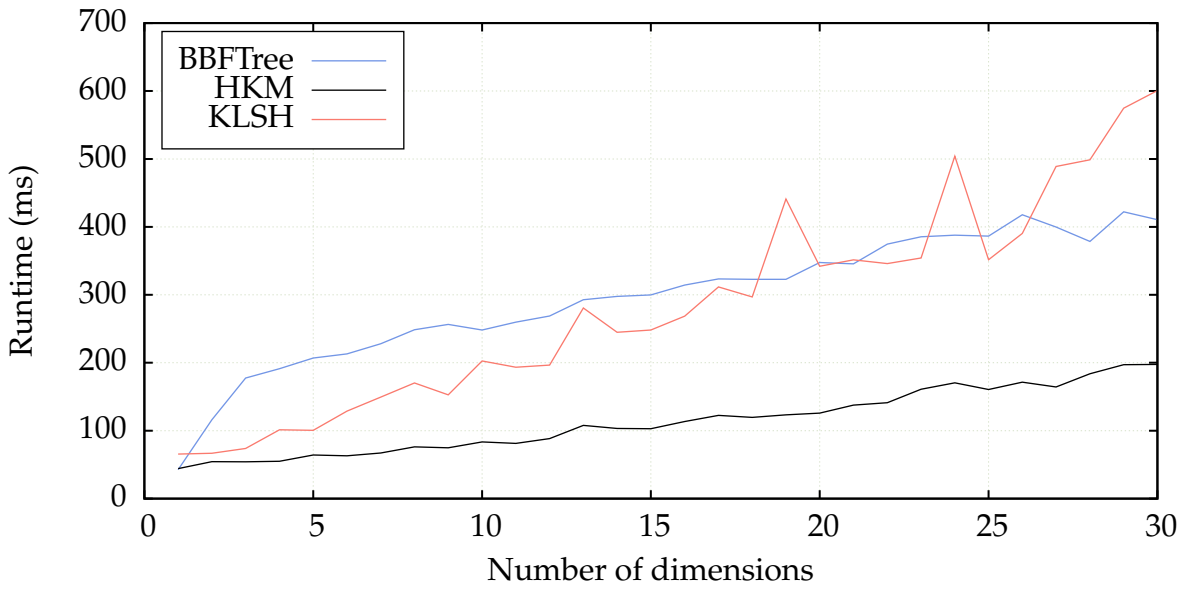



Figure 6.6: Feature vector index structure runtime of all queries with varying data point dimension

## 6.2 Feature Vector Matching Quality

In this section, we will evaluate the quality of the feature vector matching part. As described in Section 4.4 the feature vectors are constructed by first creating smaller p,q-gram trees, hashing those into a bag structure, then reducing the dimension of this bag structure and finally putting the feature vectors with smaller dimensionality into index structures.

### 6.2.1 Evaluation Method

To evaluate our method we will use both the real world data also used by Augsten et al. [3], by courtesy of the municipality of Bolzano, annotated with a correct matching, and synthetically generated test data, for which we now the correct matchings because existing trees are modified artificially. Since our algorithm includes an additional step which distinguishes false positives from positives, it is enough to test during the evaluation if the correct match is included in the set of the $k$ best matches. Because the other index structures have flaws, or have been worse than HKM in all cases, we only compared k-d tree/linear search, which returns the nearest neighbours, and HKM, which gives back only approximate nearest neighbours. One relevant parameter is the number of dimensions used. With an increasing number of dimensions, the number of correct matchings increases as more information about the trees can be encoded into the feature vectors. Because the maximal number of matches is limited, increasing the number of dimensions does have diminishing returns. If HKM is used as index structure, the disadvantage of having high dimensional data cancels this diminishing return at some point, i.e., the number of found correct matchings does not increase anymore.

### 6.2.2 Synthetically Generated Data

For benchmarking we used XML datasets found at [46], namely the *nasa1* data set, a document centric set of astronomical data, and the *orders* data set, a data centric set of order data converted to XML from the Transaction Processing Performance Council (TPC) benchmark. From those we extracted $1000$ random sub-trees with between $10$ and $100$ nodes. Those we saved in separate files. Then, we changed each of those trees randomly, once by randomly selecting one child of every node and changing its label (*one_child_change*), and once by randomly deleting, inserting, moving and copying 10 nodes within the tree (*const_10_change*).

### 6.2.3 Results

For the *Bolzano* data set – two versions of a list of streets with hierarchically organized house and apartment numbers annotated with the correct mappings – the maximal number of matches is 299. Performance of the exact (k-d tree/linear search) matching and matching using HKM with $k = 10$ for the *Bolzano* data set can be

found in Figure 6.7. It shows the number of correct matching with an increasing dimensionality of the feature vectors. A matching is correct, as in Section 6.1, if it is within the 10 nearest neighbours returned. The exact method returns the 10 nearest neighbours, HKM an approximate for those 10 nearest neighbours. The parameters for HKM are the same as before: $j = 10$ for j-means, subdivision for clusters greater than $200$ data points. The performance of HKM is in this case not much worse than the performance of the linear scan, because there are only $300$ data points.

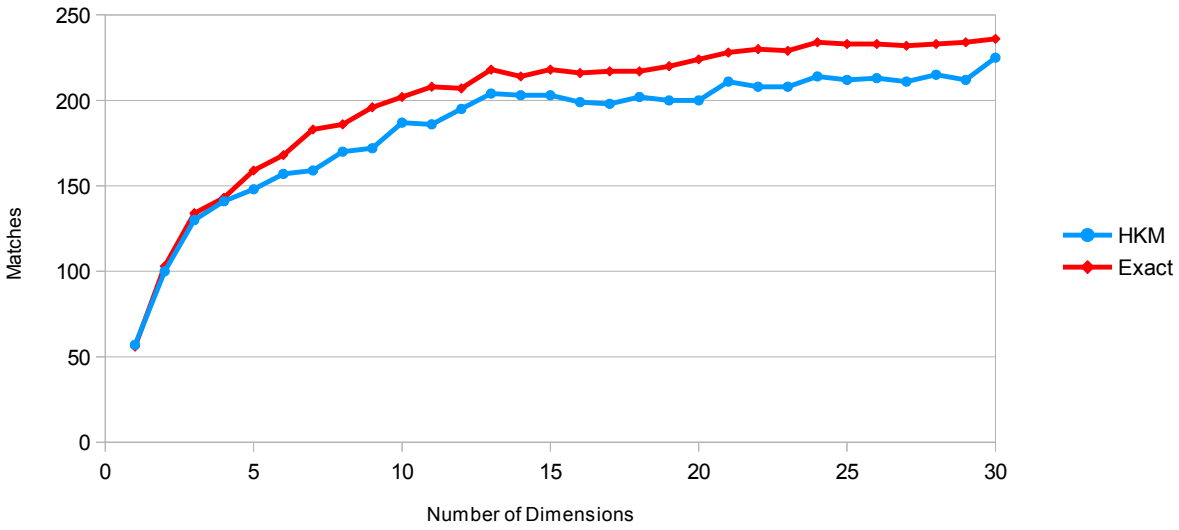



Figure 6.7: Results for the *Bolzano* dataset. Number of correct matches with increasing dimension of the feature vectors.

Figure 6.8 and 6.9 show the graphs for the *orders* and *nasa1* data set respectively. The parameters were the same. The test correspondences were generated using the method described in 6.2.2 – randomly changing one child of every node (*one_child_change*). In this case the disadvantage with regards to precision, caused by using an approximation of the k-nearest neighbours such as HKM, is more pronounced. The number of correct matches plateaus at about dimensionality $d = 20$ if HKM is used. Further increases seem to be not always beneficial. This is accentuated by the diminishing return for increasing dimensionality for the exact method, starting as soon as dimensionality $d = 10$.

Figure 6.10 and 6.11 show the graphs for the *orders* and *nasa1* data sets with the changes made by randomly inserting, deleting, moving and copying $10$ nodes within the trees
(*const_change_10_copy*).

As mentioned in Section 6.1 using HKM implies a performance/quality tradeoff, which is not always appropriate for all applications using the overall method. By using an approximate nearest neighbour search, we sacrifice matching quality for the speed of the method. Building the HKM index for $n$ points and running $n$ queries has an average case runtime of $\mathcal{O}(n \log n)$, whereas finding the exact nearest

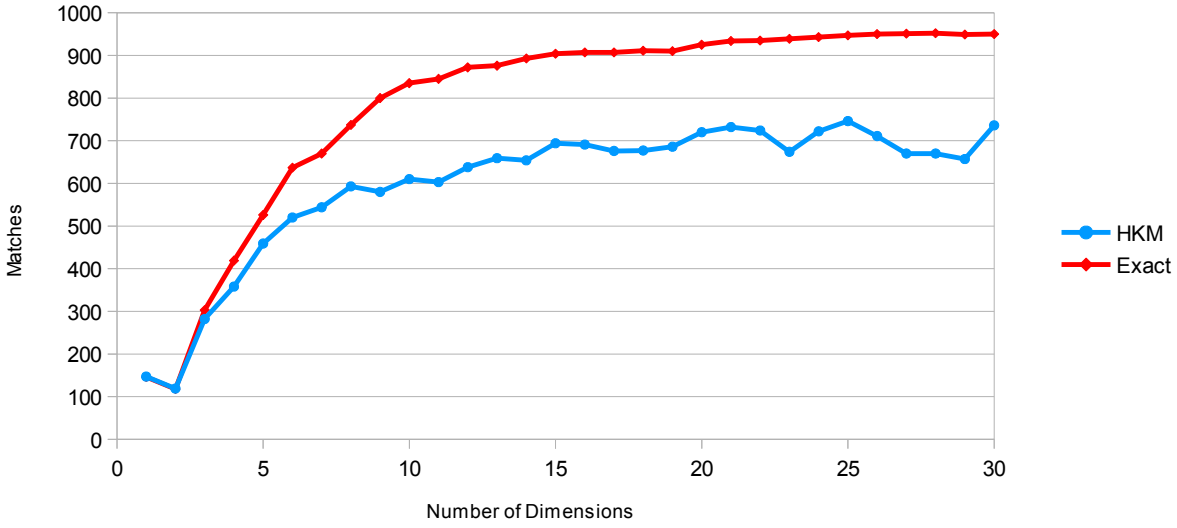



Figure 6.8: Results for the *orders/one_child_change* data set. Number of correct matches with increasing dimension of the feature vectors.

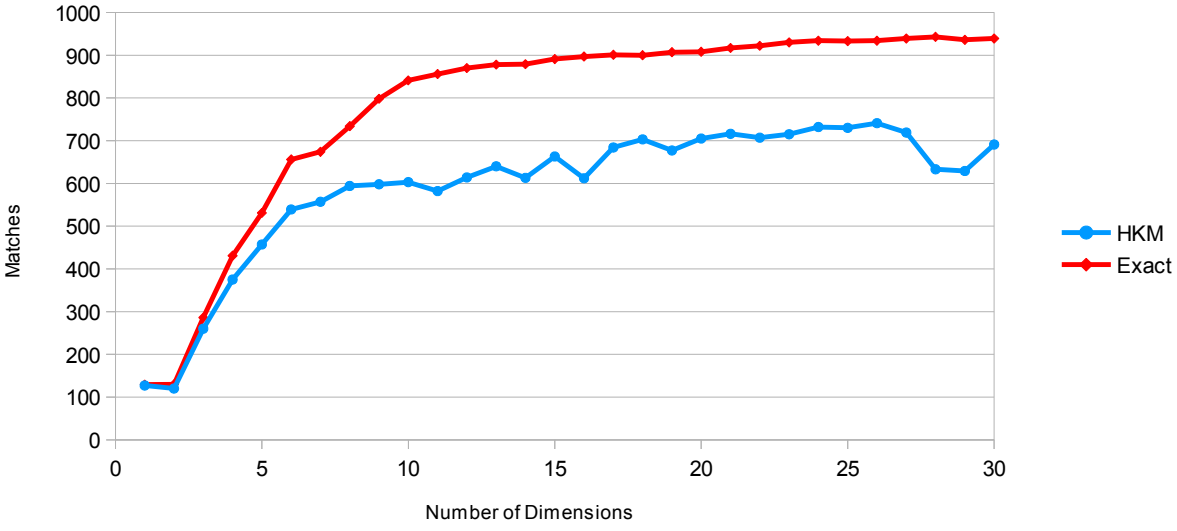



Figure 6.9: Matching the *nasa1/one_child_change* data set using feature vectors. Number of correct matches with increasing dimension of the feature vectors.

neighbours runs in, since we have high dimensional data points, $\mathcal{O}(n^2)$ time. As we deem a runtime of $\mathcal{O}(n^2)$ as unacceptable, we consider the use of an approximate nearest neighbour search a necessary evil. In Section 6.1 we showed that HKM is the best method to use for this trade-off. Figure 6.10 and 6.11 show that it is indeed a trade-off because we loose a lot of quality over the exact method. Since the alternative would be a runtime of $\mathcal{O}(n^2)$ in the matching step, this trade-off is still worthwhile in most cases. One has to consider as well that we do not have to always find the exact nearest neighbours – only the points that are reasonably close. When HKM returns nearest neighbours with only half of them being nearest neighbour in the exact method, it does not return arbitrary points. The approximate nearest neighbours returned are still near the query point and might still be correct mappings.

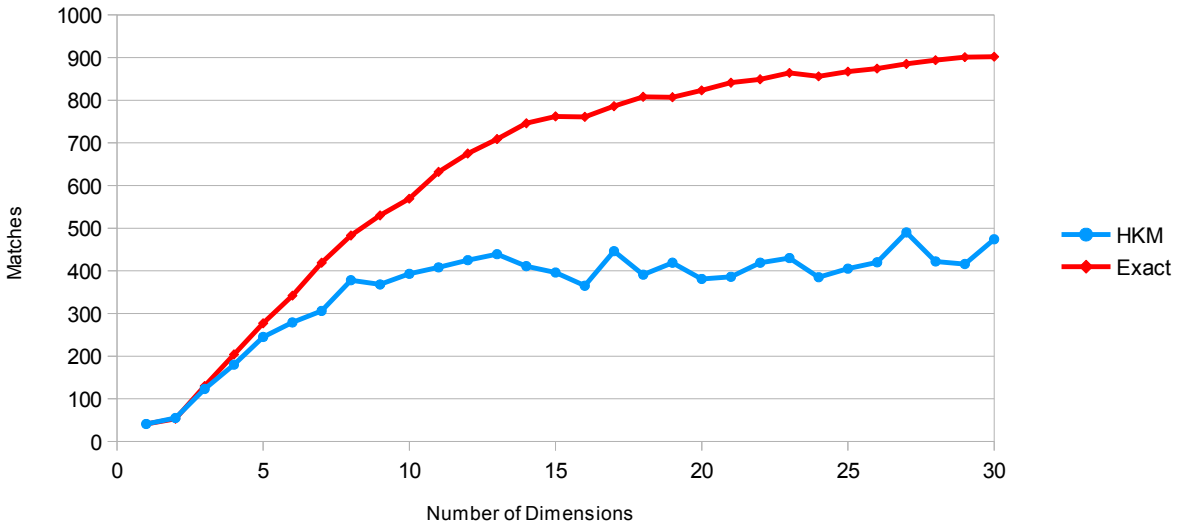



Figure 6.10: Matching the *orders / const_change_10_copy* data set using feature vectors. Number of correct matches with increasing dimension of the feature vectors.

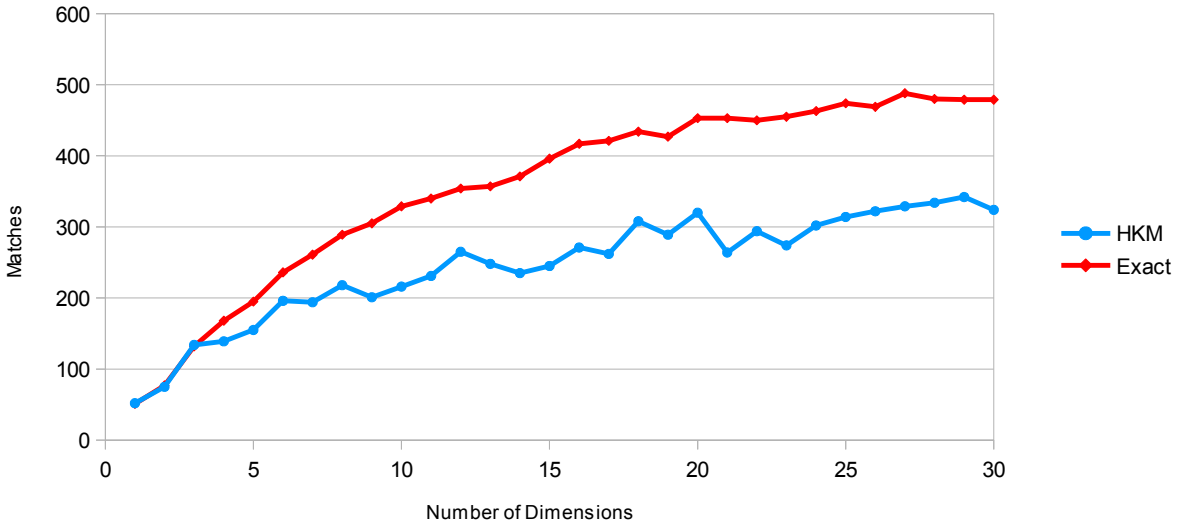



Figure 6.11: Matching the *nasa1 / const_change_10_copy* data set using feature vectors. Number of correct matches with increasing dimension of the feature vectors.

## 6.3 Runtime Evaluation

Next we take a look at how the algorithm behaves with a varying number of nodes as input. Again we tested with XML data found at [46], namely the *nasa1* and *orders* data sets. From these rather large files, the first $n$ nodes were extracted and then written in a separate file. Those smaller files were then changed randomly with different methods, namely:

1. *one_child_change_copy* / *one_child_change_nocopy*: Randomly changing the label of one child of every node. This change prevents the hash matching phase from finding any matches and should therefore test the performance of the approximate feature vector matching phase, which tries to find matches for all nodes unmatched by the hash matching.

2. *const_change_x_copy* / *const_change_x_nocopy*: Randomly deleting, inserting, moving and (if we allow a copy operator) copying $x$ nodes. Those changes should directly map to the edit operations found by the algorithm.

3. *incr_changes*: Randomly deleting, inserting, moving and copying an increasing amount of $20000$ nodes.

For every of the following runtime graphs, the number of nodes has been increased in $100$ node increments from $100$ nodes to $100000$ nodes. The XML file with this maximal number of nodes did has a size of about $2.5$ Megabytes for both data sets. The runtime is measured within the program. Only the relevant matching and edit operation generation parts are measured. It does thus not include the overhead introduced by reading the XML files or by writing the output. To smooth the runtime curve we draw the moving average of $20$ node increments and error bars showing the maximum and the minimum values of some of those averages. Because of this reason the first value of the curve is at $1000$ nodes.

All runtime evaluation has been done on Linux with an Intel Celeron G530 CPU and 6GB RAM.

Graphs with the number of nodes plotted against algorithm runtime for trees changed with the first change model and without *copy* operator (*one_child_change_nocopy*) can be seen for the *nasa1* dataset in Figure 6.12 and in Figure 6.14 for the *orders* data set. Figure 6.13 shows results for the *nasa1* dataset with a wider range of nodes. Changing one child of every node, means for the test data that the number of changes is proportional to the nodes. The output format needs to calculate at least one path (XPath) per change operation (see Section 5.3). Our tree implementation needs $\mathcal{O}(n)$ time for each path generation if the tree is relatively flat, which is, for example, the case for the *orders* dataset. This is why our method has super-linear runtime in Figure 6.14 and Figure 6.13. This could be avoided by using a better tree index for the XPath generation, which we did not investigate yet.

The same graphs with *copy* operation available (*one_child_change_copy*) can be seen in Figure 6.15 and 6.16. The algorithm is minimally slower if a *copy* operation is used because then sub-trees mapped by the hash matching or top-down matching

step, still have to be considered in the other matching steps, as they could be copied. This leads to more points in the HKM index structure during the feature vector matching step and more values in the hash tables in the hash matching step and thus increases the runtime of the algorithm.

Graphs with the second change model can be seen in Figure 6.17 with copy operation *const_change_10_copy* and Figure 6.18 without copy operation (*const_change_10_nocopy*) for the *nasa1* dataset. The same graphs for the *orders* data set are Figure 6.19 and Figure 6.20. The runtime increases proportionally to the number of nodes in those graphs. If the number of changes is constant, our method therefore has a runtime linear to the amount of nodes in the input trees.

Figure 6.21 and Figure 6.22 show the runtime with an increasing number of changes with a constant number of nodes (20000) in the input trees for the *nasa1* and the *orders* dataset. The number of edit operations our method produces is near 20000 with 10000 changes for both datasets. That means at this point it does remove everything and insert it again, i.e., it does not find any similarity. As investigating similarities needs computational power, this is the reason the runtime in Figure 6.21 is not proportional to the number of changes. This is also the reason the graph with the *orders* data set (Figure 6.22) does not have a runtime super-linear to the number of changes anymore.

Our method has, in conclusion, a runtime super-linear to the number of nodes and changes, because of the path generation. This is something that could be fixed by using a different, more efficient tree implementation, which supports a path generation proportional to the depth of the tree, or by choosing a different output format, which does not need the paths.

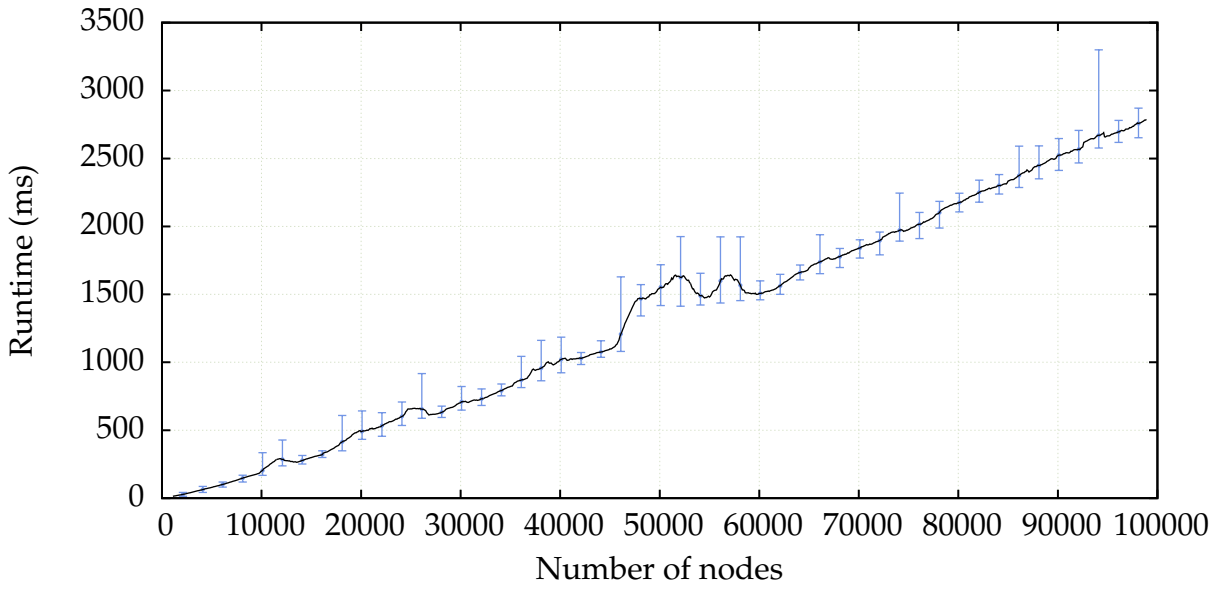



Figure 6.12: Runtime results for *nasa1/one_child_change_nocopy*.

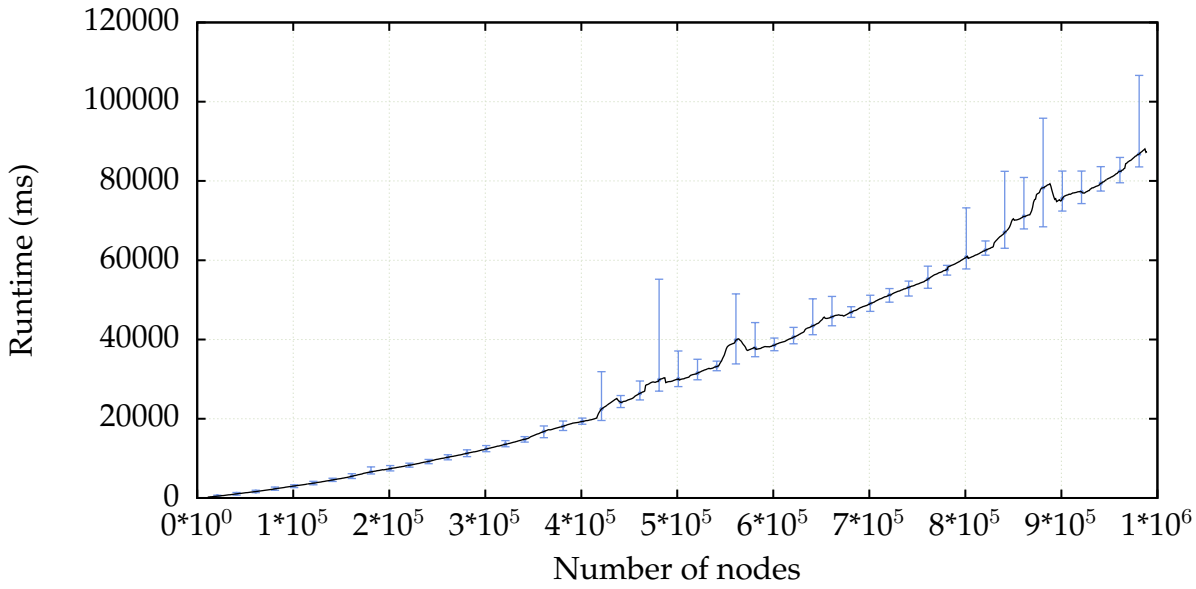



Figure 6.13: Runtime results for *nasa1/one_child_change_nocopy* with a larger number of nodes.

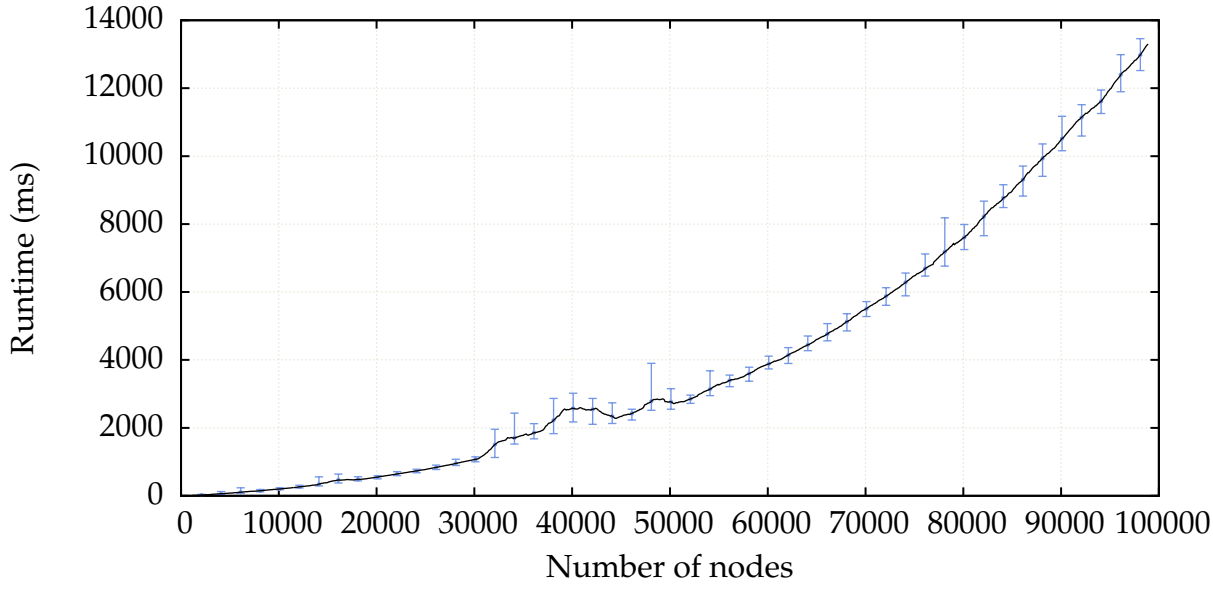



Figure 6.14: Runtime results for *orders/one_child_change_nocopy*.

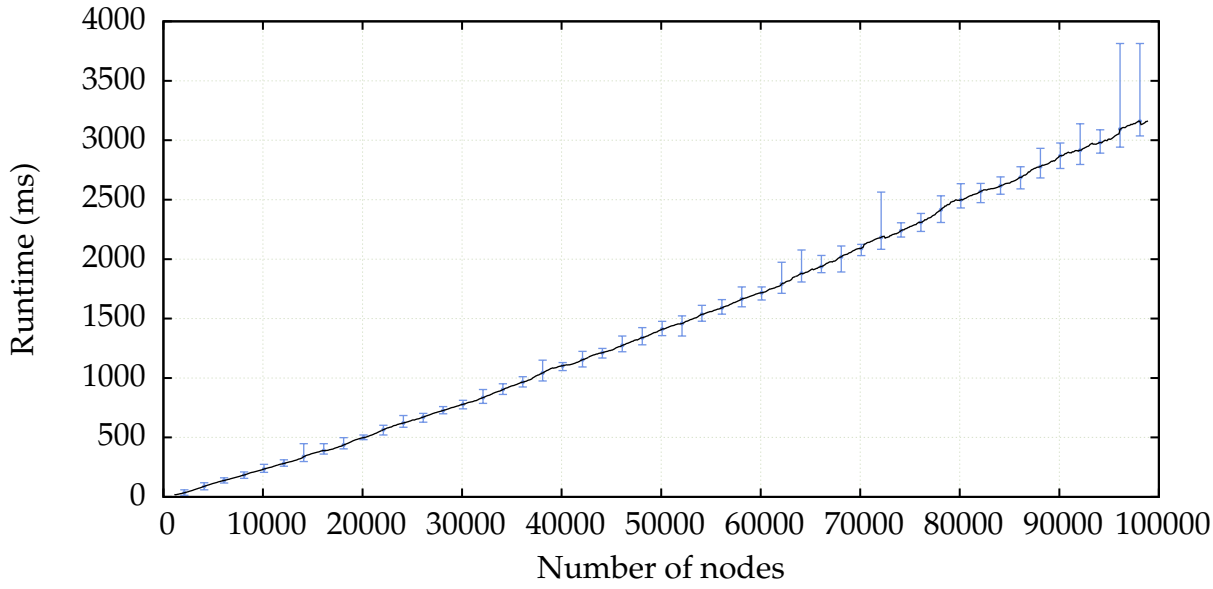



Figure 6.15: Runtime results for *nasa1/one_child_change_copy*.

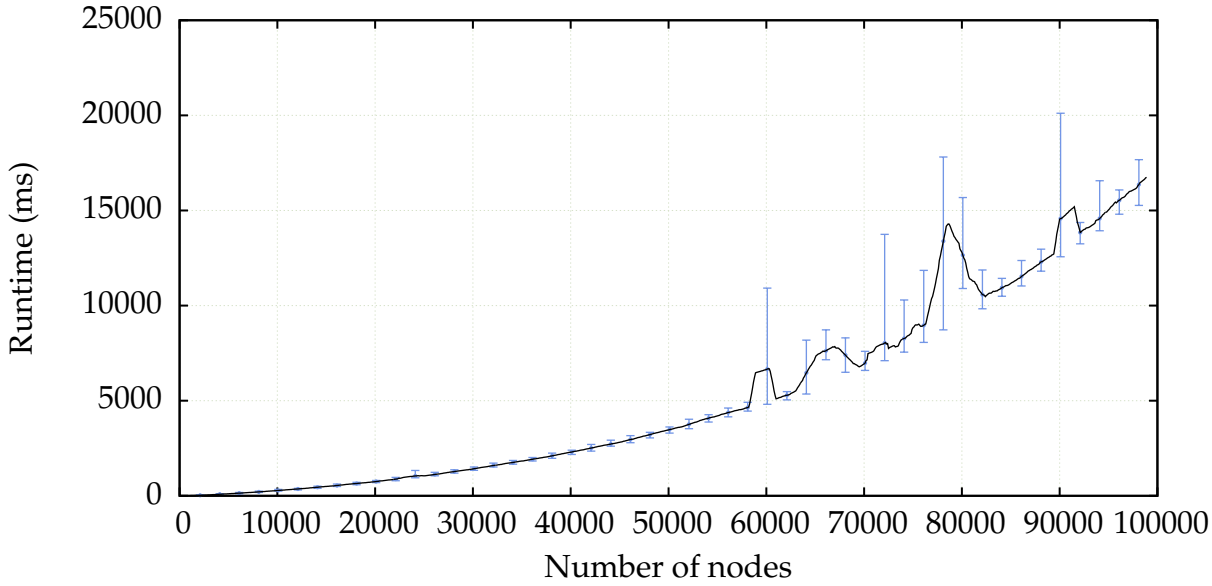



Figure 6.16: Runtime results for *orders/one_child_change_copy*.

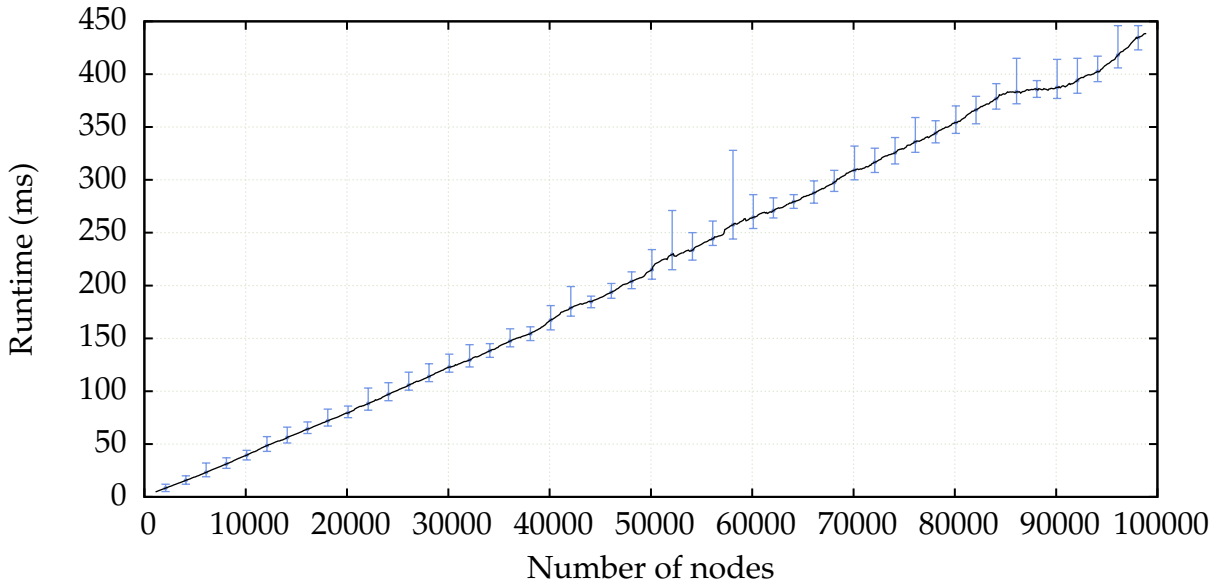



Figure 6.17: Runtime results for *nasa1/const_change_10_nocopy*.

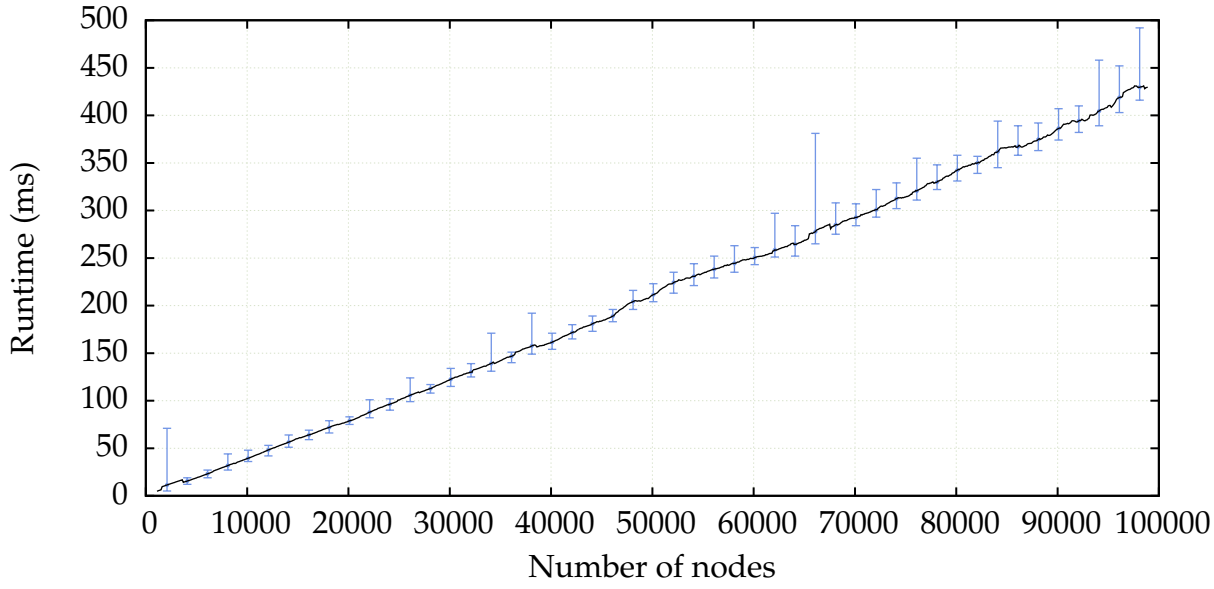



Figure 6.18: Runtime results for *nasa1/const_change_10_copy*.

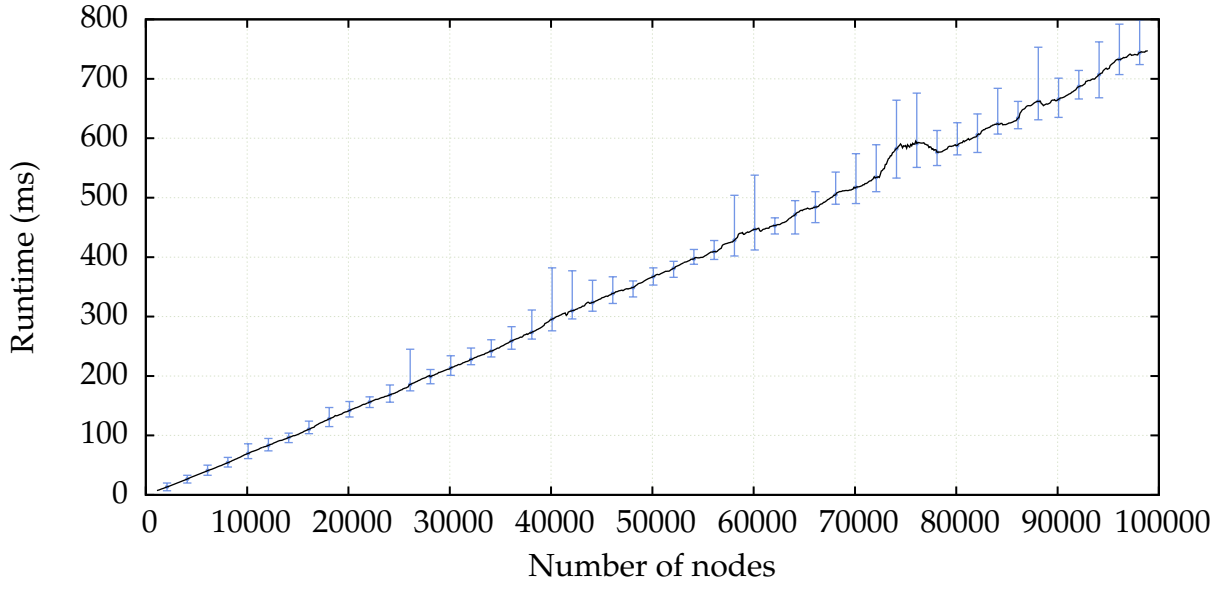



Figure 6.19: Runtime results for *orders/const_change_10_nocopy*.

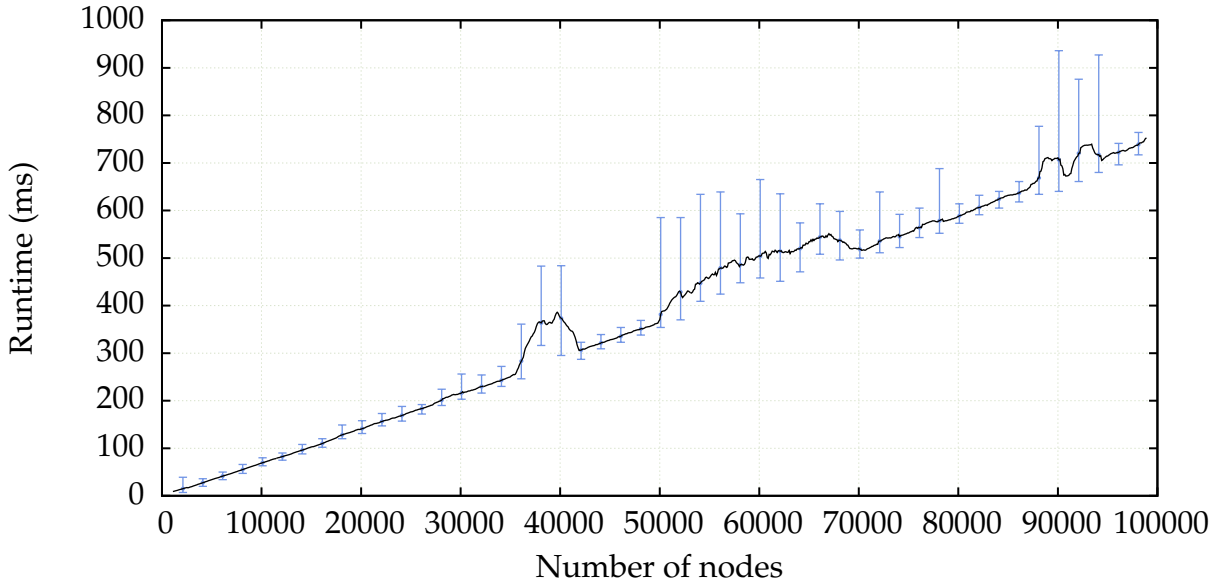



Figure 6.20: Runtime results for *orders/const_change_10_copy*.

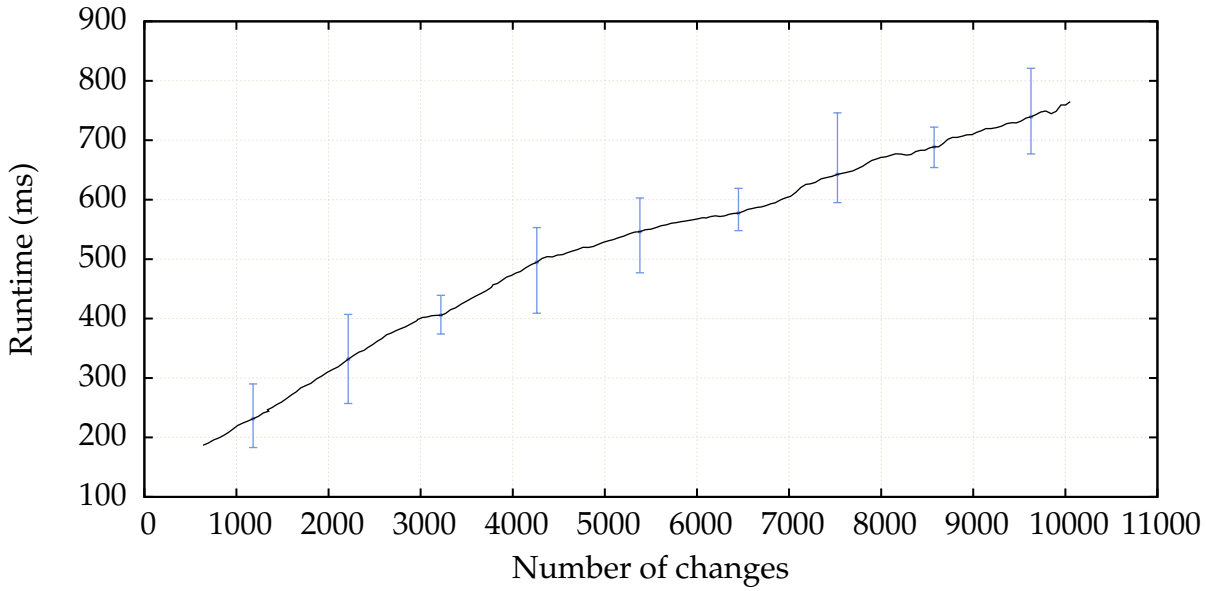



Figure 6.21: Runtime results for *nasa1/incr_changes*.

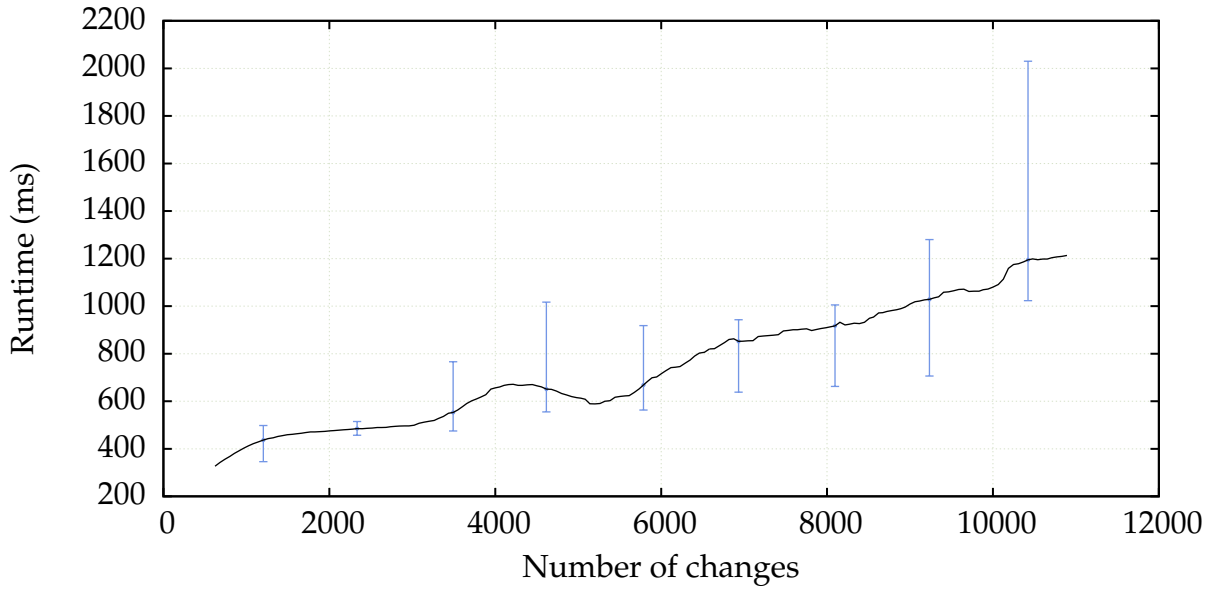



Figure 6.22: Runtime results for *orders/incr_changes*.

## 6.4 Comparison to Other Methods

To evaluate method's quality we compared the results with other tree differencing methods. The problem, as already mention in the related work (chapter 3), is that most methods have only a limited set of operations, i.e., no move operation or no copy operation or they work only on unordered or ordered data.
Even when they are comparable they often suffer from bugs, excessive runtime or enormous edit scripts for basic changes. Other comparative studies with real data (Open Office files and genome data) have found these shortcomings as well [40, 17]. Those studies have both also found that XyDiff [31] is the only serious contender. Unfortunately, the original homepage of XyDiff is not reachable anymore, but we still found it in the recesses of the internet and managed to compile the 10 year old piece of software. There is a Java version of XyDiff which is much easier to work with, but also runs about a hundred times slower than the original version written in C++.
A summary over all similar methods can be found in the related work chapter in Section 3.3. As mentioned there, X-Diff has no *move* support and is slow ($\mathcal{O}(n^2)$) anyways and we therefore not considered it further. The 3DM matching method produced wrong edit scripts and is not considered further either because it is therefore not fit for purpose. This leaves DiffXML and XyDiff with DiffXML having worst-case complexity $\mathcal{O}(n^2)$. We refrained from testing it on larger trees as the runtimes are relatively long, e.g., for 8000 nodes it already runs nearly ten seconds while XyDiff and our method runs in a few milliseconds.
To make the comparison fair we turned off sub-tree deletes as XyDiff does not support them. DiffXML theoretically uses them as well, but we granted that advantage. Both also work in ordered mode, where the order of children is important. Since this can introduce additional work and edit operations, we used our method in ordered mode as well, i.e., with the reorder step described in Section 5.2.1. Both XyDiff and DiffXML were modified such that they output the number of edit operations they produce and the time needed to calculate the edit script. The measured time did not include reading and parsing the XML trees or writing the edit script to a file.

### 6.4.1 Synthetic Data

This section shows the results of three different methods on data synthetically generated as before, i.e., by extracting an increasing amount of nodes from the *nasa1* and *orders* XML data sets, then modifying this extracted tree. The modification consisted of either random renames of one child of every node (*one_child_change*) or of random inserts, deletes, renames or moves within the tree (*const_change_x*). The data points are smoothed, as before, by calculating the moving average of $20$ points at a time. We show bars for some points to show the minima and maxima smoothed away. The range is again from $100$ to $100000$ nodes, but we had to stop DiffXML early, because of its increasing runtime.
Figure 6.23 shows the number of edit operations different methods produce on the a subset of the *orders* data set changed by modifying one child of every node. The

"Changes" line shows the optimal result a method can produce, namely simply renaming all the modified nodes. This should be a worst case scenario of all algorithms. Our approach consistently produces less edit operations than XyDiff. Figure 6.24 shows the runtimes of the different methods. Our approach is slower than XyDiff for smaller trees, but is faster for larger ones. We do not know why XyDiff is so slow with this data set, but the exponential slope for our method is caused by the XPath generation for the edit script. Since the number of changes is proportional to the number of nodes $n$, every change requires at least one XPath, and the XPath generation runs in $\mathcal{O}(n)$, because the *orders* data set is flat (the trees depth is small), our edit script generation is dominating the overall runtime and runs in $\mathcal{O}(n^2)$ here.

Figure 6.25 and Figure 6.26 show the number of changes the different methods produce and the runtime of those calculations with an increasing number of nodes from the *orders* data-set if the trees are modified by ten random tree edit operations (*const_change_10*). Since those are not a lot of changes, this can be seen as a best case scenario. Our approach nearly always produces exactly that amount of changes, i.e., ten and if not, the number of changes is only slightly higher. The maximum amount of changes of our method is $62$. XyDiff mostly finds the correct, low number of changes as well, but often (about every tenth time) it produces highly erratic results causing it to output more than $1000$ change operations. For our algorithm the average number of changes is $4.71$ with a standard deviation of $3.59$. XyDiff has an average number of changes of $250.85$ with standard deviation $707.86$. DiffXML has on average $1649.58$ changes with standard deviation $1050.83$.
Looking at the runtime of the algorithms our approach is still minimally faster and exhibits a runtime increasing linearly with the number of nodes, which shows that the edit script generation does indeed dominate the runtime and that the other steps in our change detection and correction method run in linear time.

Figure 6.27 and Figure 6.28 show the number of changes and the runtime of the different methods with an increasing amount of changes and a constant amount of nodes ($20000$) in both versions of the trees for the *orders* data set. The amount of changes our method produces consistently higher, than the amount of changes XyDiff produces. This is the case, because the amount of information the feature vector matching can use is low for the *orders* data set. It is an XML version of a single database table and thus essentially a list. The feature vector matching takes the hierarchical structure into account which does not exist in the *orders* dataset. Increasing the amount of changes destroys this information further, until the feature vector matching has next to no information to work with, that is, the result is as good as without using this step. XyDiff then has the advantage, because it focuses on the hash matching. The other change models, e.g., the one shown in Figure 6.23, do not exhibit this not enough information problem, because there the number of nodes is increased with the number of changes. The runtime difference can be explained by the different output formats. XyDiff produces output that shows the change operations in-place while our method produces a difference file, i.e., a list of operations. Thus, when XyDiff does remove tree $A$ and inserts the whole tree $B$ it just has to write tree $B$, while our method has to generate at least $|V(A)| + |V(B)|$ paths, because we disabled sub-tree removes.

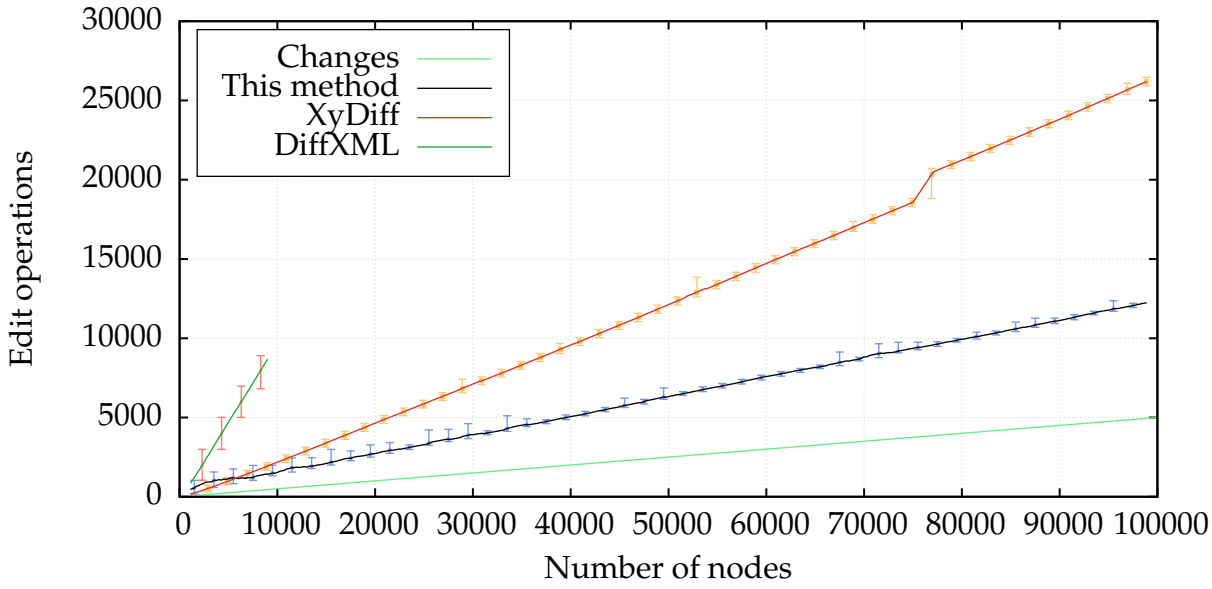



Figure 6.23: Edit operations for the *orders/one_child_change* data set.

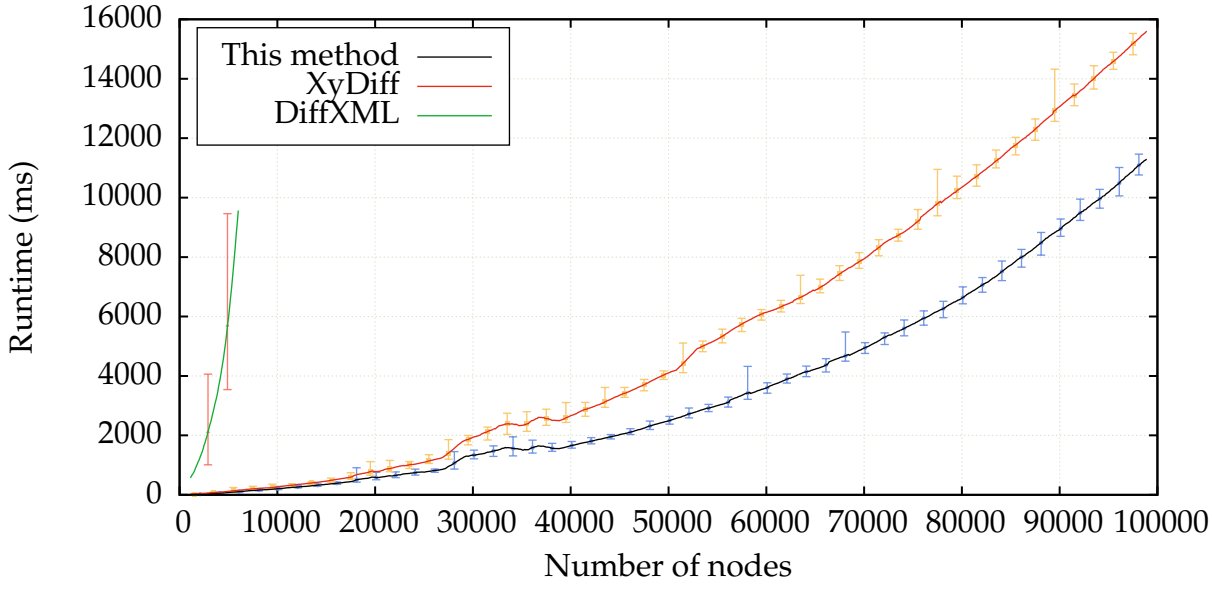



Figure 6.24: Runtime for the *orders/one_child_change* data set.

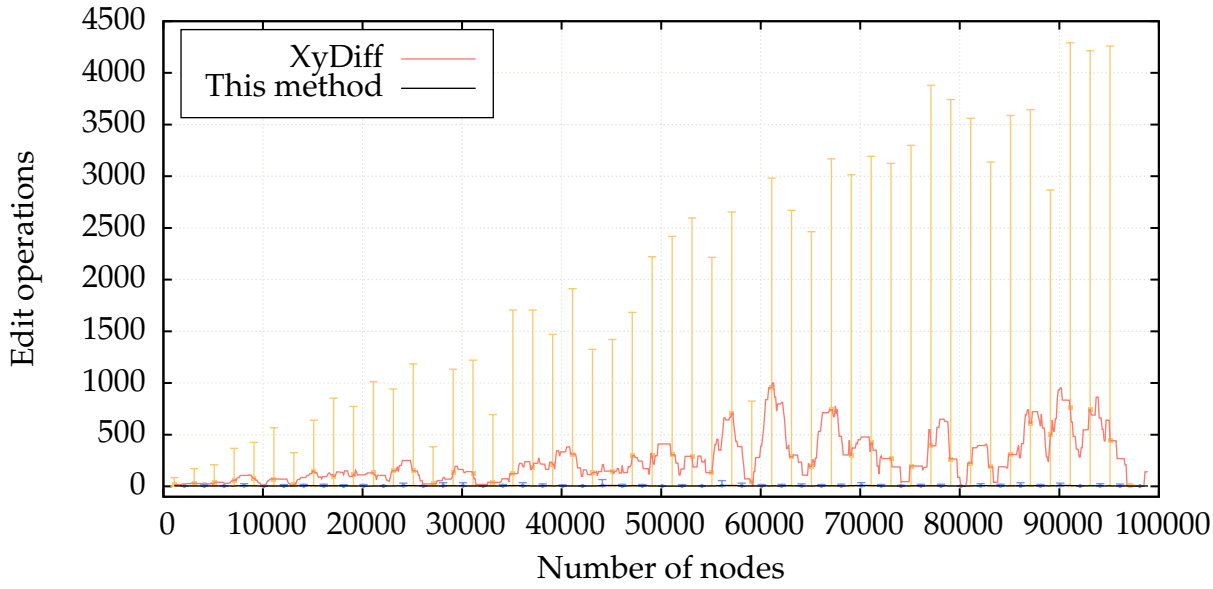



Figure 6.25: Edit operations for the *orders/const_change_10* data set.

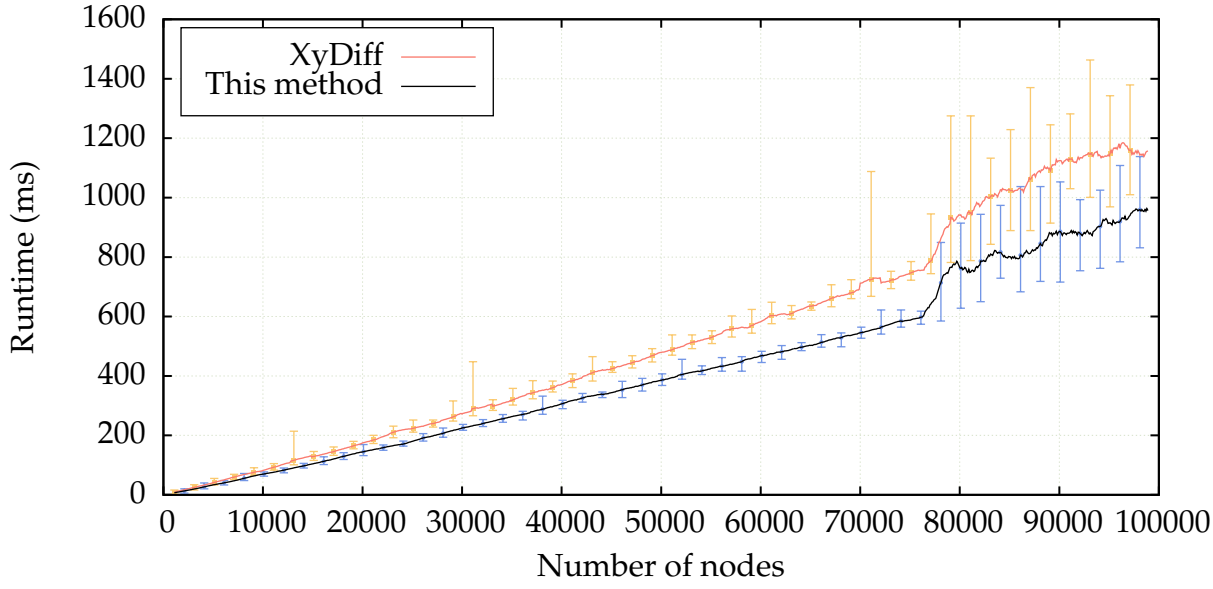



Figure 6.26: Runtime for the *orders/const_change_10* data set.

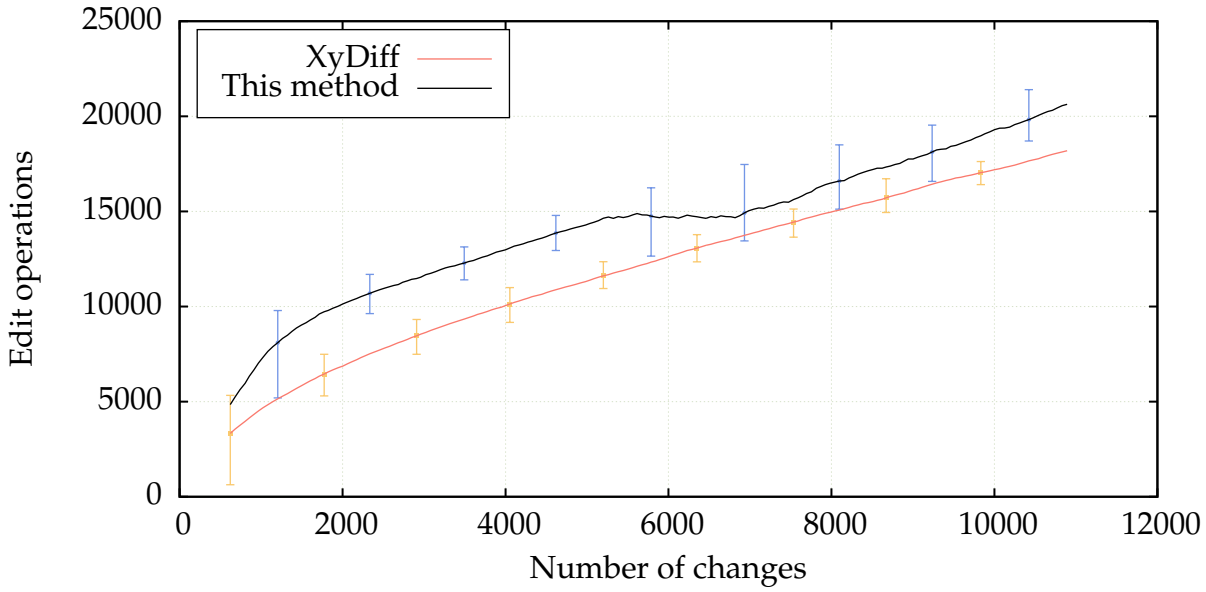



Figure 6.27: Edit operations for the *orders / incr_changes* data set.

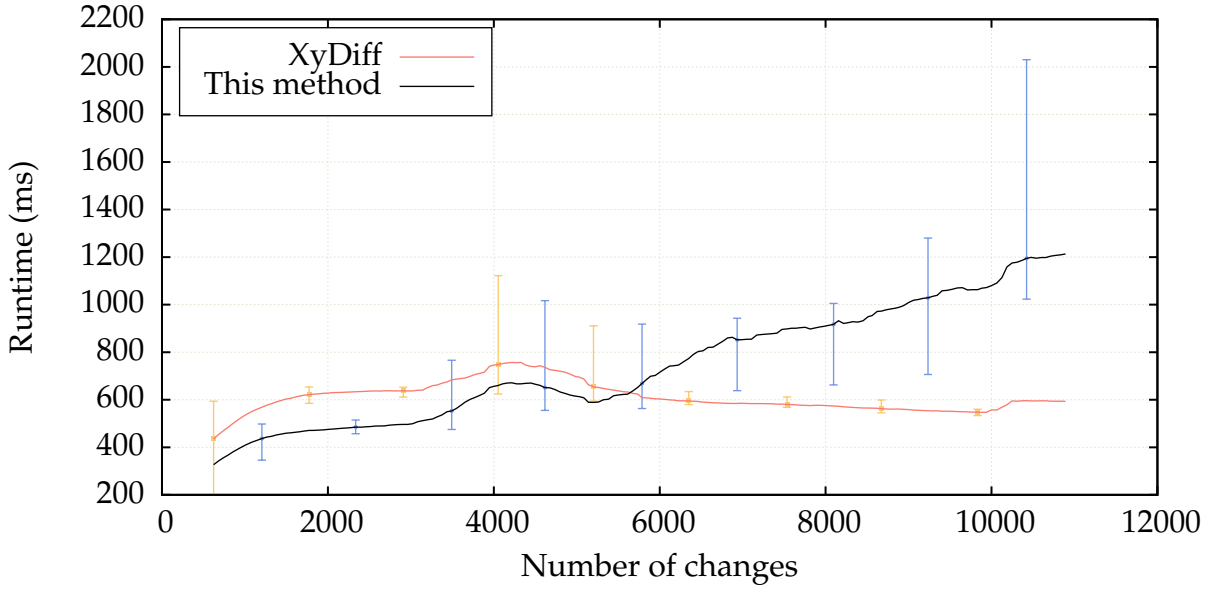



Figure 6.28: Runtime for the *orders / incr_changes* data set.

The same benchmarks for the *nasa1* data set show a similar picture, except that now the runtime of XyDiff is better when one random child node of every node is renamed (See Figure 6.30; about two times faster). This is caused by the feature vector generation, their indexing, and the nearest neighbour queries. All those steps run in $\mathcal{O}(n \log n)$ time, but they are something XyDiff does not have to do. Given that our method compared to XyDiff needs about half the number of edit operations (Figure 6.29) to express the same changes, we consider this additional amount of work worthwhile. The number of changes our method produces is consistently better when one random child per node is renamed. This is not the case anymore when ten random edits are done (*const_change_10*). XyDiff produces better results there in a few cases. Still, our algorithm produces on average $8.95$ edit operations with standard deviation $25.60$. XyDiff produces $24.12$ changes on average with standard deviation $201.70$. This can also be seen in Figure 6.31. Figure 6.32 shows the runtime of the two methods. Our runtime is moderately higher than the runtime of XyDiff. Our method thus produces a $2.69$ times better result, so the runtime spent on the feature vector matching step is worthwhile here as well.

Figure 6.33 and Figure 6.34 show the number of changes and the runtime of the different methods with an increasing amount of changes and a constant amount of nodes ($20000$) in both versions of the trees for the *nasa1* data set. This time the number of changes our method produces is significantly lower than the number of changes XyDiff produces. This can be explained by the document centric *nasa1* data set having more tree structure information. This information about the hierarchical relationships of nodes in the trees is used in the feature vector matching step, which then can find similarities which XyDiff does not discover. The increasing runtime, compared to the decreasing runtime of XyDiff, can again be explained by the different output formats.

The matching using feature vectors, in conclusion, improves the generated edit script in most cases significantly, while the speed of the overall method still remains reasonable. Compared to the other methods this often means a consistently better tree edit script quality. The XyDiff approach sometimes beats the runtime of our approach, but it is always comparable, i.e., our method does not exhibit a runtime like DiffXML. Figure 6.25 and 6.31 also show that XyDiff produces a lot of changes, seemingly at random, even if the real number of changes is small. Our approach does exhibit this behaviour to a much smaller degree as it is caused by disabling mappings which an approximation method cannot avoid altogether. But contrary to XyDiff, it does not select those disabling mappings as often and does not produce as many changes even if it selects a disabling mapping. As this behaviour is always undesirable, the runtime of our approach is comparable to the runtime of XyDiff, and the number of edit operations our method produces is lower than XyDiff or DiffXML in most cases, our approach is better than XyDiff and certainly better than DiffXML.

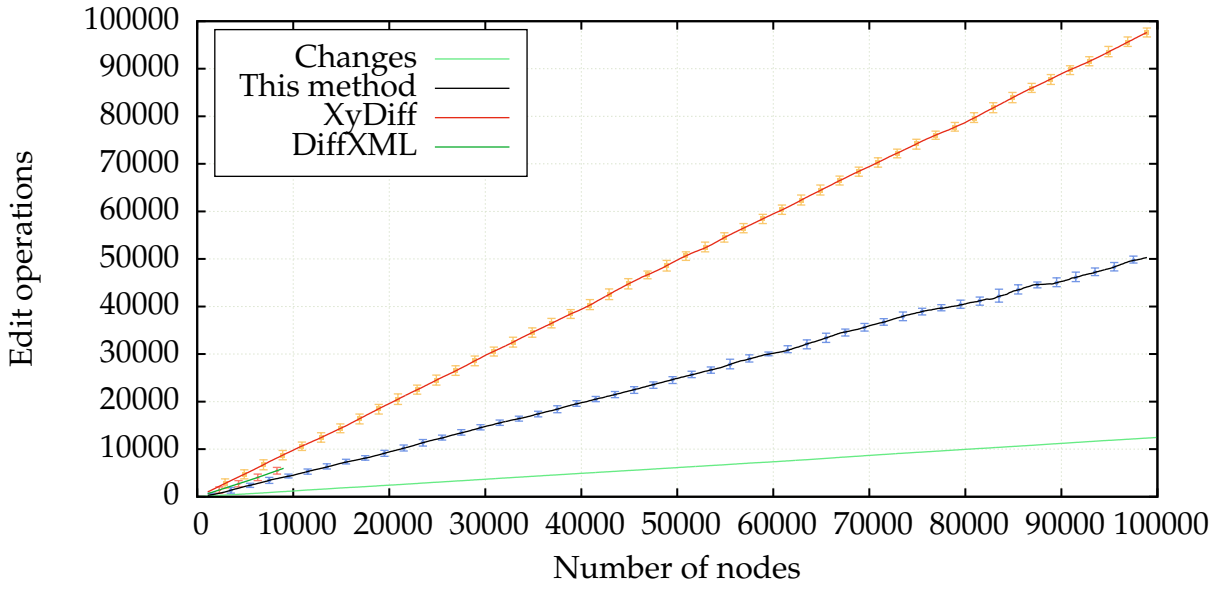



Figure 6.29: Edit operations for the *nasa1 / one_child_change* data set.

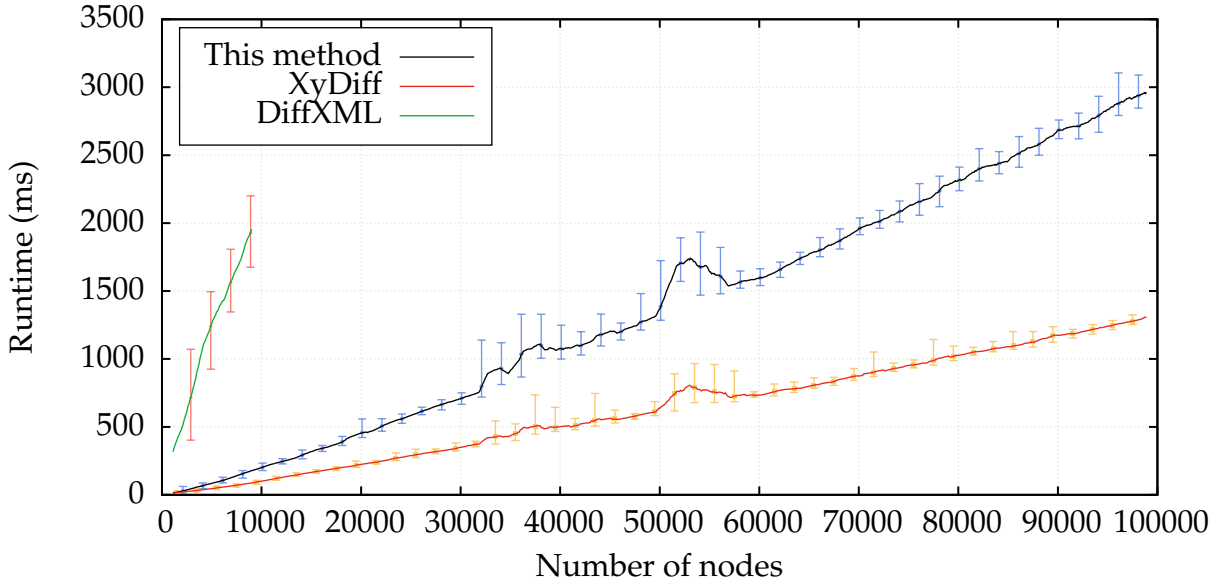



Figure 6.30: Runtime for the *nasa1 / one_child_change* data set.

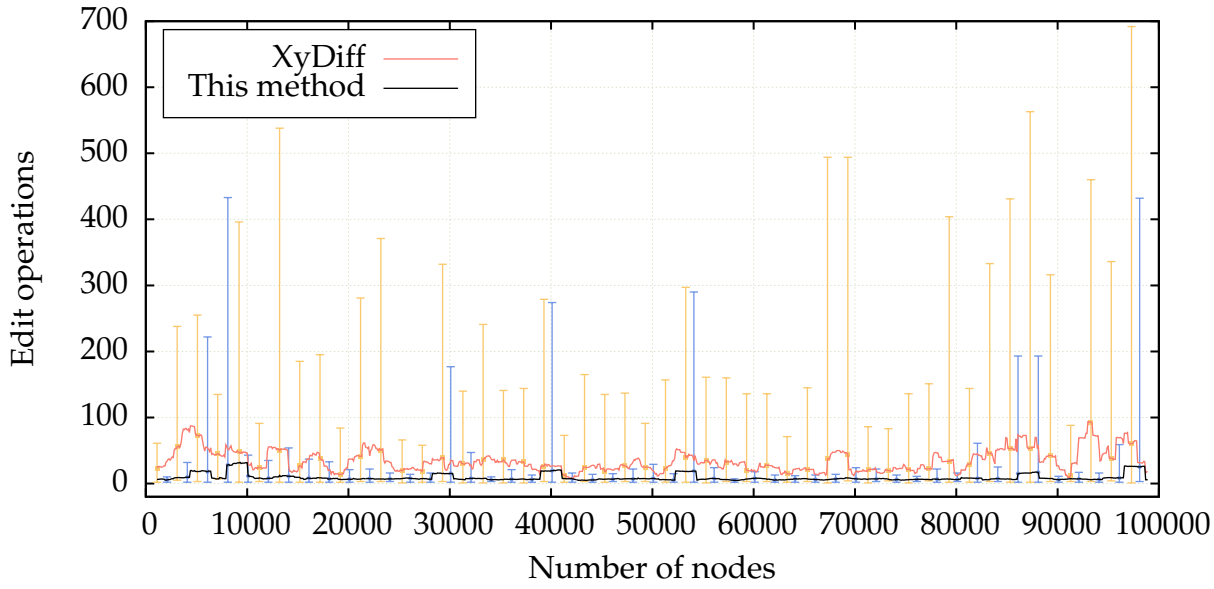



Figure 6.31: Edit operations for the *nasa1 / const_change_10* data set.

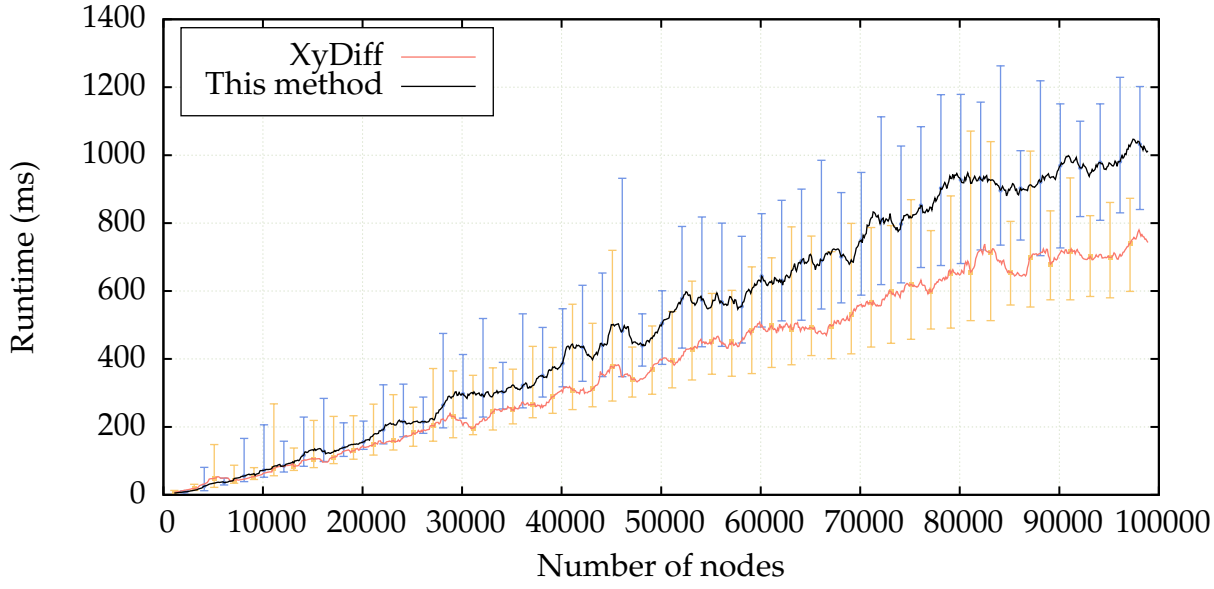



Figure 6.32: Runtime for the *nasa1 / const_change_10* data set.

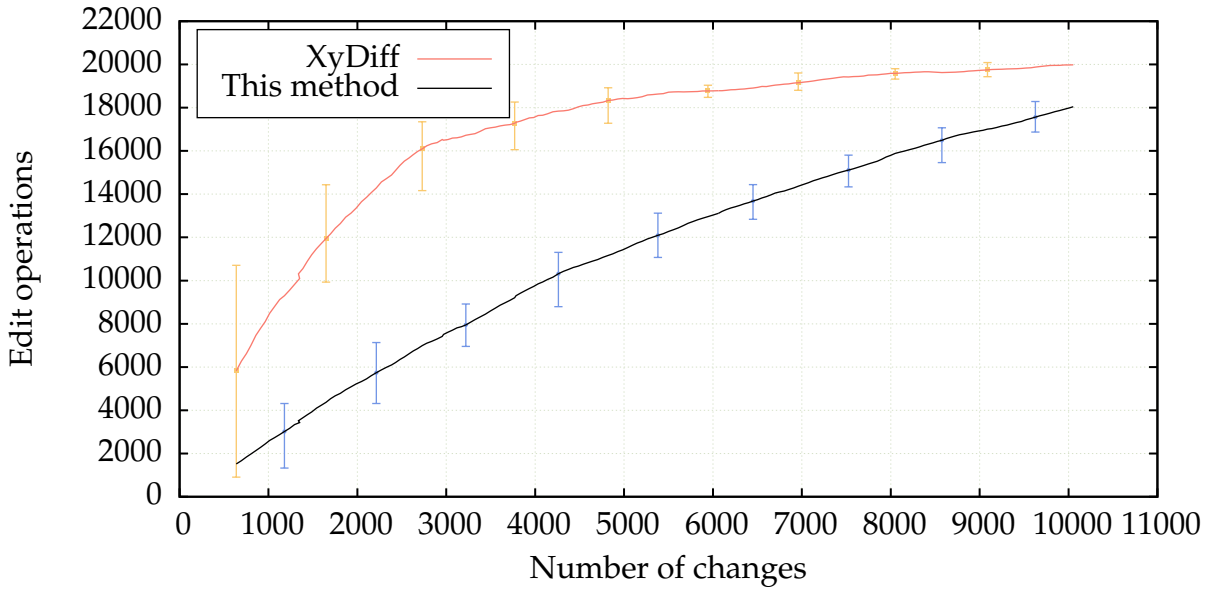



Figure 6.33: Edit operations for the *nasa1 / incr_changes* data set.

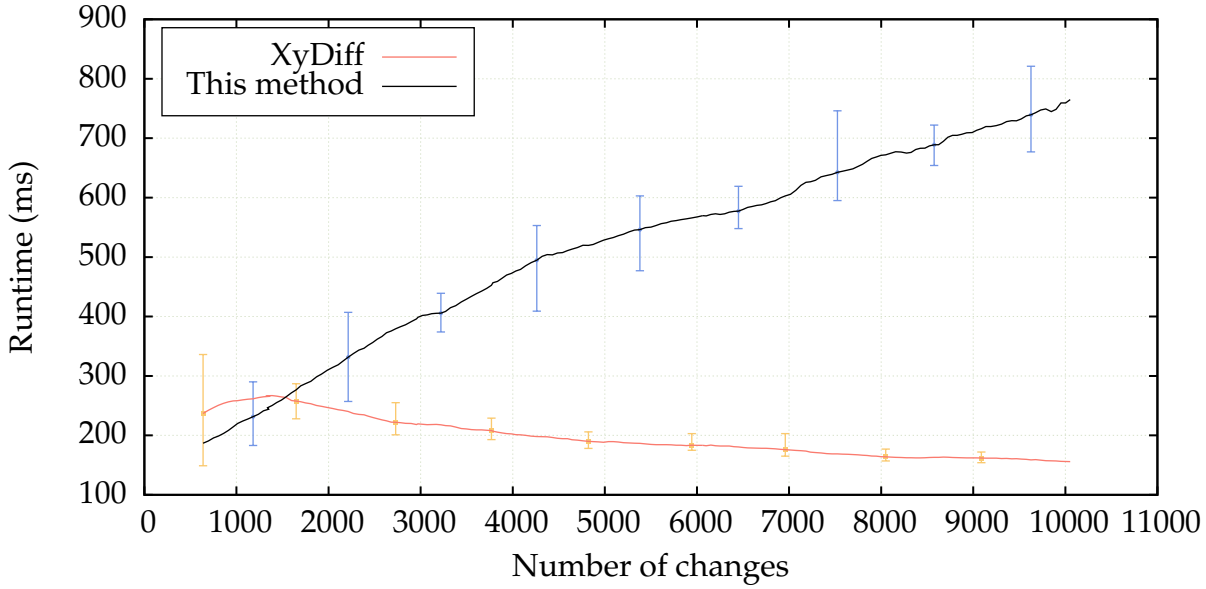



Figure 6.34: Runtime for the *nasa1 / incr_changes* data set.

### 6.4.2 Website Data

We downloaded the news websites `http://www.bbc.co.uk/news/` and `http://www.tagesschau.de/` in 20 minute intervals. They were selected because they change frequently. After collecting 900 different versions of those websites, we let our approach, XyDiff and DiffXML calculate the difference between each consecutive pair of versions. Again no sub-tree delete or copy was used, and the differencing worked with the ordered data model. DiffXML was sometimes not up to the task and aborted with an exception; the following averages thus only considered the runtimes and results where it did not crash.
This is the result for the *tagesschau* data set:

| Method | Average number of edit operations | Standard deviation of the number of edit operations | Average run-time (ms) | Standard deviation of run-time |
|---|---|---|---|---|
| This method | 81.41 | 42.41 | 120.93 | 9.61 |
| XyDiff | 960.99 | 366.99 | 37.12 | 6.32 |
| DiffXML | 359.12 | 278.37 | 2649.49 | 244.99 |

Since the trees are relatively small, the runtime of XyDiff is better. It is about 5 times faster. Sometimes XyDiff even produces less edit operations than our method. This happens when there is a relatively low number of changes. For example, XyDiff produced 54 edit operations once while our method produced 94. This is offset by times XyDiff produces a lot of changes, i.e., when our method produces 52 operations and XyDiff produces 1615. The standard deviation of the number of operations required by XyDiff is $596.29$ while it is $88.10$ for our method. This shows that our method is much more stable, in that it does not have such an extreme worst-case and that it does not run into that worst-case so often.

The *bbc* data set only confirms this view:

| Method | Average number of edit operations | Standard deviation of the number of edit operations | Average run-time (ms) | Standard deviation of run-time |
|---|---|---|---|---|
| This method | 61.32 | 87.37 | 126.58 | 10.76 |
| XyDiff | 305.16 | 596.29 | 25.15 | 8.39 |
| DiffXML | 313.39 | 299.90 | 1399.85 | 309.64 |

The additional time our method needs to compute tree edit scripts is, in conclusion, more than offset by the improved quality of the edit script. The number of changes our method creates is several times lower ($11.8$ times and $5.0$ times) than the number of changes produced by XyDiff. Additionally the number of changes our method produces is consistently low, and not, like XyDiff, sometimes low and sometimes

very high. DiffXML, having a quadratic runtime, is slow even for such small website data. Our method is, in conclusion, the best method to calculate changes between different versions of websites.

# 7 Conclusion

In this work, we first introduced how to model evolving hierarchical data in general and discussed under which circumstances having a change model would be beneficial for applications which work with this hierarchical data. We then moved on to state the problem itself in a flexible way which allows many such applications to use the results of this method, followed by a comprehensive overview of existing methods which work on hierarchical data. Introducing our own method we improved upon this state of the art by combining several approaches with our own effort, which we embedded in a flexible and performant framework which allows eventual applications to tweak the method as they desire and also allowed us to test, verify, and view the output.
Next, we evaluated which data structures and parameters are best for individual parts of the algorithm. We compared the different data structures and evaluated their usefulness. Comparing our method with others, we found that the quality of the output of our approach is nearly always better while the runtime is comparable to that of the best competitor's runtime.

## 7.1 Future Work

As mentioned in Section 6.4.1 the runtime complexity of the overall method is sometimes bounded by the path generation during the edit script creation. Improving this by introducing index structures for fast path generation would speed up the creation of the edit script, if there are a lot of changes and the trees are relatively flat. Another way to fix this would be to use a different output format, where the change information is embedded into the trees.
Another area for future work could be to embed the change detection and correction method developed in this thesis into a larger framework allowing merges/reconciliation of hierarchical data. Others have already tried to do that [27, 23], but they used tree differencing algorithms considerably worse than the algorithm introduced in this thesis.

## 7.2 Method Summary

The method worked by first doing a bottom-up hash matching. There, we calculated hashes for every node in the tree. Those hashes represented the whole subtree rooted in those nodes – one node in this sub-tree differs and the hash changes. We then mapped sub-trees to sub-trees with same hashes starting from the largest

ones. In cases where more than one sub-tree with the same hash was found we defined rules which improved the mapping quality. We also took a look at the top-down matching used and how cases in which nodes with the same label exist can be handled. We then showed how those two matching methods are used in a pre-processing step.

Then we looked at cases where those basic matching methods failed and introduced the concept of feature vectors to improve the matching quality in those worst-case scenarios. We showed different ways to construct those feature vectors on different levels. First we showed different ways to build grams for trees and finally settled on p,q-grams. Then we took a look how to best calculate the distance between grams of trees. Afterwards, we introduced two different methods to reduce those grams to feature vectors with a fixed number of dimension. The first possibility was putting each of the grams into one of $n$ frequencies, resulting in a histogram of the hash distribution. Since same grams increase the same frequencies and different grams increase different frequencies with a high probability, comparing the histograms gave us a similarity measure. The second method mapped each gram to a point on an $n$-dimensional unit sphere. By adding all grams of a tree, we then get a point in $n$-dimensional space representing the tree. Since the point on the unit sphere of identical grams is the same and points of different grams differ, the probability that different trees are in each others vicinity is smaller than the probability that similar trees are near each other. Looking closer at those two methods, we discovered that their performance is the same for large $n$ and a large number of grams. For other cases the spherical distance should be preferred. Afterwards, we looked at how to improve the distance measures by weighing grams with their frequency with which they occur in the two versions of the hierarchical data.

Having reduced the trees to points in $n$-dimensional space, we looked for efficient data structures which accelerate $k$-nearest neighbour queries. We considered k-d trees with best bin first (BBF) search, Hierarchical K-Means (HKM) and K-Means Locale Shared Hashing (KLSH) and described how they are build and how the $k$-nearest neighbour queries can be performed on those data structures.

Having completed a description of the central matching method we described how this method is embedded in a general framework that can handle XML, HTML and JSON as input, uses an efficient tree implementation and can generate and output the edit script in the form of a set of operations in both XML and JSON. It is also explained how the nodes are reordered if their order has changed, if so desired.

## 7.3 Results

We analysed and compared the performance of the different data structures accelerating the $k$-nearest neighbour queries. HKM was shown to be the most consistent such data structure with a reasonable performance/quality trade-off, and was selected as default for our method. We tested the feature vector matching quality using HKM and thus gained insight into how much precision is lost by using such an approximate nearest neighbour method.

Then we took a look at how the runtime of the whole algorithm behaves with an increasing amount of data and found that it scales well with the amount of input data.
Finally, we compared our method with other available methods on natural and synthetic data, finding that our method almost always improves the edit script quality with only a moderate increase in runtime, compared to other, simpler methods.

Our approach has with the optional *copy* or *remove* operations on sub-trees a more expressive set of change operations and can therefore produce tree edit scripts, describing the difference between versions of hierarchical data, that use fewer operations than previous methods. Our approach also works on both, hierarchical data where the order of siblings is important, and on hierarchical data where it is not important. Previous methods only worked on either ordered or unordered hierarchical data. Even if our approach is not allowed to produce *copy* operations, it generates fewer operations than all other comparable methods in most cases. This improvement is accomplished using a novel matching method using feature vectors for trees and data structures for fast nearest neighbour queries in high dimensional spaces. Even though we introduced this additional matching method, the overall runtime of the algorithm producing the changes is only insignificantly higher, or sometimes even lower, than the runtime of comparable methods. The approach presented in this thesis therefore constitutes a significant improvement upon those existing, comparable methods.

# Bibliography


[1] S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, et al. Xyleme, a dynamic warehouse for xml data of the web. In *International Symposium on Database Engineering & Applications*, pages 3–8. IEEE, 2001.

[2] Nikolaus Augsten, Michael Bohlen, Curtis Dyreson, and Johann Gamper. Approximate joins for data-centric XML. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 814–823, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35(1):4:1–4:36, February 2008.

[4] D.T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. *Technical Report*, pages 95–372, 1995.

[5] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, pages 1000–, Washington, DC, USA, 1997. IEEE Computer Society.

[6] Laurent Boyer, Amaury Habrard, and Marc Sebban. Learning metrics between tree structured data: Application to image recognition. In *Proceedings of the 18th European conference on Machine Learning*, ECML '07, pages 54–66, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 90–101, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and querying changes in semistructured data. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 4–13, Washington, DC, USA, 1998. IEEE Computer Society.

[9] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Rec.*, 26(2):26–37, June 1997.

[10] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[12] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):2:1–2:19, December 2009.

[13] Serge Dulucq and Hélène Touzet. Analysis of tree edit distance algorithms. In *Proceedings of the 14th annual conference on Combinatorial pattern matching*, CPM'03, pages 83–95, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29 – 35, 1975.

[15] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[16] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 287–298, New York, NY, USA, 2002. ACM.

[17] Cornelia Hedeler and Norman W. Paton. A comparative evaluation of XML difference algorithms with genomic data. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, SSDBM '08, pages 258–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Piotr Indyk. Nearest neighbors in high-dimensional spaces. In *Handbook of Discrete and Computational Geometry*. CRC Press, 2004.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[20] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees: an alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, May 1995.

[21] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001. `http://www.scipy.org/`.

[22] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 91–102, London, UK, UK, 1998. Springer-Verlag.

[23] K. Komvoteas and J. Wells. XML diff and patch tool. Master's thesis, Heriot-Watt University, Edinburgh, Scotland, 2003.

[24] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 63–74, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[25] Kyong-Ho Lee, Yoon-Chul Choy, and Sung-Bae Cho. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):965–979, August 2004.

[26] Tancred Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, MobiDe '03, pages 93–97, New York, NY, USA, 2003. ACM.

[27] Tancred Lindholm. A three-way merge for XML documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[28] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[29] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.

[30] DeltaXML Ltd. DeltaXML - Change control for XML, 2012. `http://www.deltaxml.com/`.

[31] Amelie Marian. Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 41–52, Washington, DC, USA, 2002. IEEE Computer Society.

[32] A.W. Moore. An intoductory tutorial on kd-trees. *Extract from Andrew Moore's PhD Thesis: Effcient Memory based Learning for Robot Control*, 1991.

[33] Adrian Mouat. XML diff and patch utilities. Master's thesis, Heriot-Watt University, Edinburgh, Scotland, June 2002.

[34] Adrian Mouat. Delta update language specification, November 2012. `http://diffxml.sourceforge.net/dul/dul.html`.

[35] David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 2161–2168, Washington, DC, USA, 2006. IEEE Computer Society.

[36] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, August 2010.

[37] Mateusz Pawlik and Nikolaus Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, December 2011.

[38] Dave Raggett. HTML tidy. http://tidy.sourceforge.net/, 2004. `http://tidy.sourceforge.net/`.

[39] Mervi Ranta and Tancred Lindholm. A 3-way merging algorithm for synchronizing ordered trees - the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, 2001.

[40] Sebastian Rönnau, Jan Scheffczyk, and Uwe M. Borghoff. Towards XML version control of office documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, DocEng '05, pages 10–19, New York, NY, USA, 2005. ACM.

[41] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, August 1988.

[42] Stanley M. Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.

[43] C. E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[44] D. Shasha, J. T. L. Wang, Kaizhong Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):668–678, 1994.

[45] Yang Song and Sourav S. Bhowmick. BioDIFF: an effective fast change detection algorithm for genomic and proteomic data. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 146–147, New York, NY, USA, 2004. ACM.

[46] Dan Suciu. XML data repository, 2012. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[47] Gabriel Valiente. An efficient bottom-up distance between trees. In *SPIRE*, pages 212–219, 2001.

[48] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *19th International Conference on Data Engineering*, ICDE, pages 519–530, 2003.

[49] Haiyuan Xu, Quanyuan Wu, Huaimin Wang, Guogui Yang, and Yan Jia. KF-Diff+: Highly efficient change detection algorithm for XML documents. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 1273–1286, London, UK, 2002. Springer-Verlag.

[50] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 754–765, New York, NY, USA, 2005. ACM.

[51] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, December 1989.

[52] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Combinatorial Pattern Matching*, CPM, pages 395–407. Springer, 1995.

[53] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463 – 474, 1995.

[54] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15:205–222, 1996. 10.1007/BF01975866.

[55] Kaizhong Zhang and Tao Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *Information Processing Letters*, 49(5):249 – 254, 1994.