Algorithms for Web Scraping

Patrick Hagge Cording

Kongens Lyngby 2011

Abstract

Web scraping is the process of extracting and creating a structured representation of data from a web site. HTML, the markup language used to structure data on webpages, is subject to change when for instance the look-and-feel is updated. Since current techniques for web scraping are based on the markup, a change may lead to the extraction of incorrect data.

In this thesis we investigate the potential of using approximate tree pattern matching based on the tree edit distance and constrained derivatives for web scraping. We argue that algorithms for constrained tree edit distances are not suited for web scraping. To address the high time complexity of optimal tree edit distance algorithms, we present the lower bound pruning algorithm which, based on the data tree T_D and the pattern tree T_P, will attempt to remove branches of T_D that are not part of an optimal mapping. Its running time is $O\big(|T_D||T_P| \cdot \sigma(T_D, T_P)\big)$, where $\sigma(T_D, T_P)$ is the running time of the lower bound method used. Although it asymptotically is close to the approximate tree pattern matching algorithms, we show that in practice the total execution time is reduced in some cases. We develop several methods for determining a lower bound on the tree edit distance used for approximate tree pattern matching, and we see that our generalization of the q-gram distance from strings is the most effective with our algorithm. We also present a similar algorithm that use the HTML grammar to prune T_D, and some heuristics to guide the approximate tree pattern matching algorithm.

Preface

This master's thesis has been prepared at DTU Informatics from february 2011 to august 2011 under supervision by associate professors Inge Li Gørtz and Philip Bille. It has an assigned workload of 30 ECTS credits.

Source code for the software developed for the thesis is available from the following two mirrors.

`http://www.student.dtu.dk/~s062408/scpx.zip`
`http://iscc-serv2.imm.dtu.dk/~patrick/scpx.zip`

Acknowledgements. I would like to thank my supervisors for always being enthusiastic at our meetings. The individuals who have influenced my work include Henrik and Anne-Sofie of Kapow Software who took time out of their schedule to discuss web scraping, my friend Christoffer who brought my attention to the challenges faced in web scraping, friend and co-student Kristoffer who wrapped his head around the project in order to give criticism, and my girlfriend who has provided proof-reading. I am grateful to all.

Patrick Hagge Cording

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web scraping is the process of extracting and creating a structured representation of data from a web site. A company may for instance want to autonomously monitor its competitors product prices, or an enterprising student may want to unify information on parties from all campus bar and dormitory web sites and present them in a calendar on her own web site.

If the owner of the information does not provide an open API, the remedy is to write a program that targets the markup of the web page. A common approach is to parse the web page to a tree representation and evaluate an XPath expression on it. An XPath denotes a path, possibly with wildcards, and when evaluated on a tree, the result is the set of nodes at the end of any occurence of the path in the tree. HTML, the markup language used to structure data on web pages, is intended for creating a visually appealing interface for humans. The drawback of the existing techniques used for web scraping is that the markup is subject to change either because the web site is highly dynamic or simply because the look-and-feel is updated. Even XPaths with wildcards are vulnerable to these changes because a given change may be to a tag which can not be covered by a wildcard.

In this thesis we show how to perform web scraping using approximate tree pattern matching. A commonly used measure for tree similarity is the tree edit distance which easily can be extended to be a measure of how well a pattern can be matched in a tree. An obstacle for this approach is its time complexity, so we consider if faster algorithms for constrained tree edit distances are usable for web scraping, and we develop algorithms and heuristics to reduce the size of the tree representing the web page.

The aim of the project is to a develop a solution for web scraping that is

- tolerant towards as many changes in the markup as possible,
- fast enough to be used in e.g. a web service where response time is crucial, and
- pose no constraints on the pattern, i.e. any well-formed HTML snippet should be usable as pattern.

The rest of the report is organized as follows. Chapter 2 is a subset of the theory of the tree edit distance. We accentuate parts that have relevance to web scraping. In chapter 3 we describe how approximate tree pattern matching is used for web scraping and we discuss pros and cons of the algorithms mentioned in chapter 2. In chapter 4 we transform six techniques for approximation of the tree edit distance to produce lower bounds for the pattern matching measure. Chapter 5 presents an algorithm that uses the lower bound methods from chapter 4 to prune the data tree. We also present an algorithm and two heuristics that use knowledge of HTML to reduce the tree. In chapter 6 we conduct some experiments and chapter 7 is a discussion of our results.

Appendix A describes the software package developed for the thesis. It gives a brief overview of the design of the package and contains examples of how to use it.

1.1 Preliminaries and Notation

1.1.1 General

It is assumed that the reader has basic knowledge of HTML, is familiar with the string edit distance problem, and the concept of dynamic programming.

We use Smallcaps when referring to algorithms that have been implemented or described using pseudocode. `Typewriter` is used for URL's as well as modules and functions in the implementation chapter. Likewise, serif is used to refer to classes.

For pseudocode we use $\leftarrow$ for assignment and $=$ for comparison. We do not distinguish lists and sets. The union of two lists is the set of elements in the concatenation of the lists, e.g. $[x, y] \cup [x, x, z] = [x, y, z]$. We use $\oplus$ to denote concatenation, e.g. $[x, y] \oplus [x, x, z] = [x, y, x, x, z]$. A tuple is a list of fixed length. We use round parenthesis for tuples to distinguish them from lists. The concatenation operator also applies to tuples. The length of a list or tuple a is $|a|$.

1.1.2 Trees and Forests

T denotes a tree and F denotes a forest. Trees and forests are rooted and ordered if nothing else is stated. $|T|$ is the size of the tree T. $V(T)$ is the set of all nodes in T. $T(v)$ is the subtree rooted at the node v and $F(v)$ is the forest obtained from removing v from $T(v)$. When nothing else is stated the nodes are assigned postorder indices. Any arithmetic operation on two nodes is implicitly on their indices and the result is a number. $T[i]$ is the i^{th} node in T. The '$-$' operator on a tree and a node removes a subtree or a node from a tree, e.g. $T - T(v)$ removes the subtree rooted at v from T. The empty forest is denoted θ.

We define the following functions on trees.

$root : T \rightarrow V(T)$**.** Returns the root of the tree given as input.

$height : V(T) \rightarrow \mathbb{Z}$**.** Computes the height of the subtree rooted at the node given as input.

$depth : V(T) \rightarrow \mathbb{Z}$**.** Returns the length of the path from the root of T to given node.

$lml : V(T) \rightarrow V(T)$**.** Finds the leftmost leaf of the tree rooted at the node given as input.

$nca : V(T) \times V(T) \rightarrow V(T)$**.** Finds the nearest common ancestor of the nodes.

$leaves : T \rightarrow \mathbb{Z}$**.** Returns the number of leaves in the tree.

$degree : T \rightarrow \mathbb{Z}$**.** Computes the max number of children of all nodes in T.

For simplicity we sometimes give a tree as input to a function expecting a node. In such cases the root of the tree is implicitly the input.

Finally, we define the anchor of a tree T as path p from the root of T to some node v where each node on p has exactly one child and v is either a leaf or has more than one child. If the root has more than one child, the length of the anchor is 1.

CHAPTER 2

Tree Edit Distance

In this chapter we review the theory of the tree edit distance which is needed for the rest of the thesis.

2.1 Problem Definition

Definition 2.1 (Tree Edit Distance) Let T_1 and T_2 be rooted, ordered trees and let $E = op_0, op_1, \ldots, op_k$ be an edit script (a sequence of operations on the trees) that transforms T_1 into T_2. Let γ be a cost function on operations, then the cost of an edit script is $\sum_{i=0}^{k-1} \gamma(op_i)$. The tree edit distance $\delta(T_1, T_2)$ is the cost of a minimum cost edit script.

The tree edit distance problem is to compute the tree edit distance and its corresponding edit script. In the general formulation of the problem, the edit operations are *insert*, *delete*, and *relabel*. A node v can be inserted anywhere in T. When inserted as the root, the old root becomes a child of v. If inserted between two nodes u and w, v takes the place of w in the left-to-right order of the children of u, and w becomes the only child of v. A node can also be inserted as a leaf. When deleting a node v, its children become children of the parent of v. Relabeling a node does not impose any structural changes on T. An example of the operations is shown in figure 2.1.

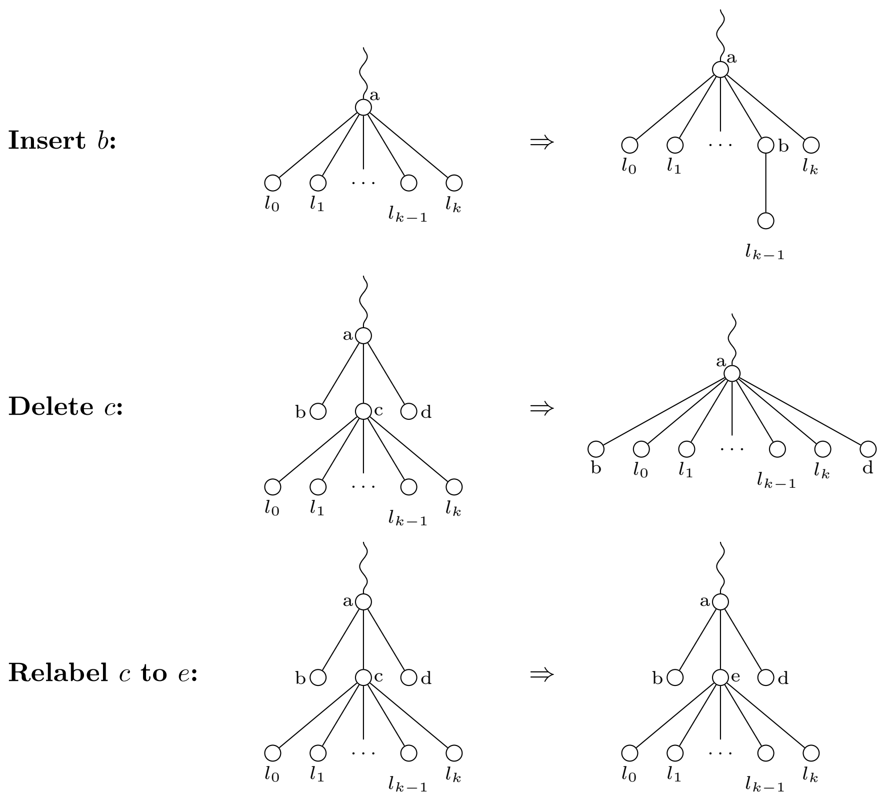


Figure 2.1: Example *insert*, *delete*, and *relabel* operations.

The tree edit distance can also be expressed in terms of a mapping. In the general formulation of the tree edit distance problem the following mapping is used.

Definition 2.2 (Mapping) Let T_1 and T_2 be rooted, ordered trees and M a set of tuples from $V(T_1) \times V(T_2)$. M is a tree edit distance mapping between T_1 and T_2 if, for any pair of tuples (v_1, w_1), $(v_2, w_2) \in M$, the following conditions are satisfied.

One-to-one $v_1 = v_2 \iff w_1 = w_2$.

Ancestor v_1 is an ancestor of $v_2 \iff w_1$ is an ancestor of w_2.

Sibling v_1 is to the left of $v_2 \iff w_1$ is to the left of w_2.

Mappings and edit scripts are interchangeable. A pair of nodes $(v, w) \in M$ corresponds to relabeling v to w. Any node in T_1 that is not in any tuple in M should be deleted, and any node in T_2 that is not in any tuple in M should be inserted. So if an edit script creates a mapping that violates one of the three conditions, it is not a solution to the tree edit distance problem. An example of a mapping is shown in figure 2.2. Intuitively, the mapping conditions are formalizations of what makes trees similar. If they are relaxed, the tree edit distance will no longer correspond to what we believe is similarity between trees. If they are augmented, the tree edit distance will correspond to a different perception of similarity between trees, and the complexity of computing it may be reduced.

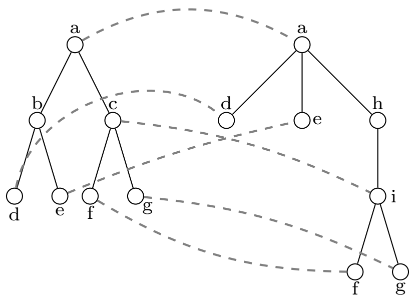


Figure 2.2: A mapping that corresponds to the edit script *relabel(a,a), delete(b), insert(h), relabel(c,i), relabel(d,d), relabel(e,e), relabel(f,f), relabel(g,g)*. We usually omit *relabel* operations in the edit script if their cost is zero, but they are included here to illustrate the correspondance to mappings.

The cost function γ is a function on nodes and a special blank character λ. Formally, we define it as $\gamma : (V(T_1) \cup \lambda \times V(T_2) \cup \lambda) \backslash (\lambda \times \lambda) \rightarrow \mathbb{R}$. A common way of distinguishing nodes is on labels. A unit cost function is one where the costs do not depend on the nodes. We define the simplest unit cost function γ_0 to be

$$\begin{aligned} \gamma_0(v \rightarrow \lambda) &= 1 \\ \gamma_0(\lambda \rightarrow w) &= 1 \\ \gamma_0(v \rightarrow w) &= \begin{cases} 0 & \text{if } v = w \\ 1 & \text{otherwise} \end{cases} \end{aligned} \qquad \forall (v, w) \in V(T_1) \times V(T_2) \tag{2.1}$$

The tree edit distance is a distance metric if the cost function is a distance metric. The following are the requirements for a cost function to satisfy to be a distance metric.

1. $\delta(T_1, T_1) = 0$
2. $\delta(T_1, T_2) \geq 0$
3. $\delta(T_1, T_2) = \delta(T_2, T_1)$ (symmetry)
4. $\delta(T_1, T_2) + \delta(T_2, T_3) \geq \delta(T_1, T_3)$ (triangle inequality)

All the bounds on tree edit distance algorithms given in the following section are symmetric because the tree edit distance is a distance metric.

2.2 Algorithms

2.2.1 Overview

This section will present the main results found in the litterature for the tree edit distance problem chronologically and relate them to eachother.

K. C. Tai, 1979 [15] This paper presents the first algorithm to solve the tree edit distance problem as it is defined in this report. It is a complicated algorithm and is considered impractical to implement. Its time complexity is

$$O\big(|T_1||T_2| \cdot height(T_1)^2 \cdot height(T_2)^2\big)$$

which in the worst case is $O\big(|T_1|^3|T_2|^3\big)$.

Zhang and Shasha, 1989 [22] The authors formulate a dynamic program and show how to compute a solution bottom-up. They reduce space requirements by identifying which subproblems that are encountered more than once and discard of those that are not. The time complexity is reduced because the algorithm exploits that the solution to some subproblems is a biproduct of a solution to another. The algorithm runs in

$$O\big(|T_1||T_2| \cdot \min\big(leaves(T_1), height(T_1)\big) \cdot \min\big(leaves(T_2), height(T_2)\big)\big)$$

time (worst case $O\big(|T_1|^2|T_2|^2\big)$) and $O\big(|T_1||T_2|\big)$ space.

The algorithm is referred to throughout the report, so an extensive description is given in section 2.2.2.

Shasha and Zhang, 1990 [14] This paper presents several (sequential and parallel) algorithms where the authors speed up their previous algorithm assuming a unit cost function is used. The main result is the sequential unit cost algorithm (from now on referred to as Shasha and Zhang's unit cost algorithm or just the unit cost algorithm), in which subproblems are ruled out based on a threshold k on the tree edit distance supplied as input to the algorithm. By doing so they achieve a

$$O\big(k^2 \cdot \min(|T_1|, |T_2|) \cdot \min(leaves(T_1), leaves(T_2)\big)$$

time bound and maintain the $O\big(|T_1||T_2|\big)$ space bound.

The algorithm is described in detail in section 5.1 on page 37 where it is used as a launch pad for further work.

Philip Klein, 1998 [9] Based on the same formulation of a dynamic program as Zhang and Shasha's algorithm, the author propose an algorithm that requires fewer subproblems to be computed in the worst case. In a top-down implementation, the algorithm alternates between the formulation by Zhang and Shasha and its symmetric version based on the sizes of the trees in the subforest of T_1, and the author shows that this leads to a time complexity of

$$O\big(|T_1|^2|T_2| \cdot \log(|T_2|)\big)$$

while maintaining a $O\big(|T_1||T_2|\big)$ space bound [3].

Demaine *et al.*, 2009 [6] The authors present an algorithm that alternates between the two formulations of the dynamic program similarly to Klein's algorithm. However, the conditions for chosing one over the other, i.e. the *recursion strategy*, are more elaborate. The algorithm runs in

$$O\big(|T_1|^2|T_2|(1 + \log\frac{|T_2|}{|T_1|})\big)$$

time (worst case $O\big(|T_1|^2|T_2|)\big)$ and $O\big(|T_1||T_2|\big)$ space.

The authors prove that the time bound is a lower bound for algorithms based on possible recursion strategies for the dynamic program formulation of the tree edit distance problem by Zhang and Shasha.

2.2.2 Zhang and Shasha's Algorithm

This section describes Zhang and Shasha's algorithm. It is structured to show how to go from a naive algorithm to the space bound improvement, and then further on to the time bound improvement.

The algorithm computes the tree edit distance using the following lemma.

Lemma 2.3 (Tree edit distance [3]) *Let F_1 and F_2 be ordered forests, γ a distance metric cost function on nodes, and v and w the rightmost nodes of F_1 and F_2, respectively. The tree edit distance δ is found from the recursion:*

$$\begin{aligned}
\delta(\theta,\theta) &= 0\\
\delta(F_1,\theta) &= \delta(F_1-v,\theta)+\gamma(v\to\lambda)\\
\delta(\theta,F_2) &= \delta(\theta,F_2-w)+\gamma(\lambda\to w)\\
\delta(F_1,F_2) &= \min\begin{cases}\delta(F_1-v,F_2)+\gamma(v\to\lambda)\\ \delta(F_1,F_2-w)+\gamma(\lambda\to w)\\ \delta(F_1(v),F_2(w))+\delta(F_1-T_1(v),F_2-T_2(w))\\ \quad+\gamma(v\to w)\end{cases}
\end{aligned}$$

The intuition behind lemma 2.3 is the following. We always compare the rightmost nodes v and w of the forests*. When comparing the nodes there are three cases—delete v, insert w, and relabel v to w—which have to be investigated, so we branch for each case. In the delete-branch we remove v from its forest because it is now accounted for. Similarly w is removed from its forest in the insert-branch. When nodes are relabeled we branch twice and the pair of relabeled nodes becomes a part of the mapping. This means that in order to adhere to the mapping restrictions, nodes descending from v can only map to nodes descending from w. Consequently, the left forest of v must be compared to the left forest of w.

The lemma states that the tree edit distance can be found by composing results from subproblems, so the algorithm employs dynamic programming. It computes the result bottom up, so we need a table entry for each possible subproblem. The forests are given postorder indices so the nodes of the subproblems always have consecutive indices. Thus, the set of possible subforests per forest is

$$S_1=\{v_i,v_{i+1},\dots,v_j\mid 0\le i\le j<|F_1|\}$$

If we count these we get

$$|S_1|=\sum_{i=0}^{i<|F_1|}|F_1|-i\in O\big(|F_1|^2\big)$$

*The recursion is symmetric so it is also possible to compare the leftmost nodes.

subproblems. Since we require $|F_1|^2|F_2|^2$ subproblems to be computed we have now established that a naive algorithm can compute the tree edit distance in $O\big(|T_1|^2|T_2|^2\big)^{\dagger}$ time and space.

We now show how the space bounds can be improved. We observe from the recursion that it is either the rightmost node of one of the forests that is removed or all but the rightmost tree. Therefore, it would suffice to have a $|F_1||F_2|$ dynamic programming table if the solution to any subproblem consisting of two trees was already known.

This is utilized by the algorithm as follows. We maintain a permanent table of size $|T_1||T_2|$ for all subproblems that consists of two trees. We solve each subproblem from the permanent table in the function TREEDIST. In TREEDIST we create a temporary table of at most size $|T_1||T_2|$ to hold the subproblems that are needed to solve the subproblem from the permanent table. If we need a subproblem to solve another subproblem that is not present in the temporary table while in TREEDIST, it is because it consists of two trees, and the solution can thus be read from the permanent table. Since we maintain one table of size $|T_1||T_2|$ and one of at most size $|T_1||T_2|$, the space bound has been improved to $O\big(|T_1||T_2|\big)$.

The time bound can also be improved. Occasionally, when in TREEDIST, we solve a subproblem which consist of two trees. This happens for all pairs of subtrees where at least one tree is rooted on the path from the root of a tree to its leftmost leaf. Such a subproblem has already been solved in another invocation of TREEDIST.

To take advantage of this, we define the notion of a keyroot. A keyroot is a node that has one or more siblings to the left. Then we only invoke TREEDIST on subproblems from the permanent table where both trees have keyroots as roots. In TREEDIST we save the result of a subproblem in the permanent table if it consists of two trees where at least one of them is not a rooted at a keyroot. Zhang and Shasha show that there is $\min\big(leaves(T), height(T)\big)$ keyroots in a tree T, so the running time of the algorithm is

$$O\big(|T_1||T_2| \cdot \min\big(leaves(T_1), height(T_1)\big) \cdot \min\big(leaves(T_2), height(T_2)\big)$$

The algorithm ZHANGSHASHA and its subprocedure TREEDIST are given in the following pseudocode.

[†] For simplicity we will state the time and space bounds as functions of input trees. The preceeding derivation uses the size of the forests because it has its starting point in the recursion. Subsequent tree edit distance algorithms do not accept forests as input.

Algorithm 1 ZhangShasha(T_1, T_2)

```
1: Let $D$ be the permanent dynamic programming table (which is global)
2: for each node $v \in$ Keyroots($T_1$) do
3:   for each node $w \in$ Keyroots($T_2$) do
4:     Treedist($v, w$)
```

Algorithm 2 Treedist(T_1, T_2)

```
1: Let $F$ be the local dynamic programming table
2: $F[0,0] \leftarrow 0$
3: for $i = 1$ to $|T_1|$ do
4:   $F[i,0] \leftarrow F[i-1,0] + \gamma(T_1[i] \rightarrow \lambda)$
5: for $j = 1$ to $|T_2|$ do
6:   $F[0,j] \leftarrow F[0,j-1] + \gamma(\lambda \rightarrow T_2[j])$
7: for $i = 1$ to $|T_1|$ do
8:   for $j = 1$ to $|T_2|$ do
9:     if $lml(T_1) = lml(T_1[i])$ and $lml(T_2) = lml(T_2[j])$ then
10:      $F[i,j] \leftarrow \min\big($
           $F[i,j-1] + \gamma(T_1[i] \rightarrow \lambda)$,
           $F[i-1,j] + \gamma(\lambda \rightarrow T_2[j])$,
           $F[i,j] + \gamma(T_1[i] \rightarrow T_2[j])\,\big)$
11:      $D[lml(T_1)+i, lml(T_2)+j] \leftarrow F[i,j]$
12:    else
13:      $F[i,j] \leftarrow \min\big($
           $F[i,j-1] + \gamma(T_1[i] \rightarrow \lambda)$,
           $F[i-1,j] + \gamma(\lambda \rightarrow T_2[j])$,
           $F[lml(T_1[i]), lml(T_2[j])] + D[lml(T_1)+i, lml(T_2)+j]\,\big)$
```

2.3 Constrained Mappings

The high complexity of the algorithms for the tree edit distance has led people to study other formulations of the problem. By imposing an additional constraint on the mapping, the problem becomes easier to solve.

Constrained mappings are interesting because the algorithms are approximations to the tree edit distance and their running times are faster. The cost of a constrained mapping is always greater than or equal to the tree edit distance because the contraints are added to the original mapping requirements. In some contexts the approximation may be so good that an algorithm producing a contrained mapping is favorable.

Valiente [18] and Bille [3] give surveys of the different mappings found in the litterature. In this section we will describe the top-down and isolated-subtree mappings.

2.3.1 Top-down

Definition 2.4 (Top-down mapping) Let T_1 and T_2 be rooted, ordered trees and M a set of tuples from $V(T_1) \times V(T_2)$. A top-down mapping M is a mapping (definition 2.2 on page 6) for which it holds that if some pair $(v, w) \in M \backslash (root(T_1), root(T_2))$ then $(parent(v), parent(w)) \in M$.

An example is shown in figure 2.3. The top-down mapping corresponds to restricting insertions and deletions to leaves.

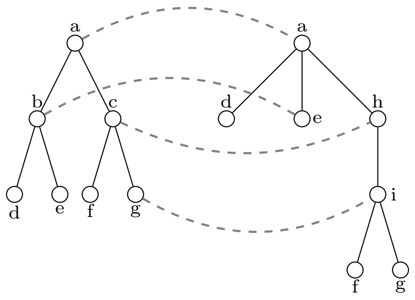


Figure 2.3: A top-down mapping that corresponds to the edit script *relabel(a,a), relabel(b,e), relabel(c,h), delete(d), delete(f), delete(e), relabel(g,i), insert(f), insert(g).*

Top-down mappings are useful for comparing hierarchical data, e.g. instances of an XML database. Changes to an XML database is often insertion or deletion of one or more entries, which in either case affects a leaf or a an entire subtree.

Selkow [13] gave the first algorithm for computing a top-down mapping. It is a simple dynamic programming algorithm which runs in $O(|T_1||T_2|)$ time and space.

Other algorithms have been proposed but they are mostly variants tailored for a specific context [21] or require even further contraints on the mapping [11] and have the same time and space complexity.

2.3.2 Isolated-subtree Mapping

Definition 2.5 (Isolated-subtree mapping) Let T_1 and T_2 be rooted, ordered trees and M a set of tuples from $V(T_1) \times V(T_2)$. An isolated-subtree mapping M is a mapping (definition 2.2 on page 6) for which it holds that for any three pairs $(v_1, w_1), (v_2, w_2), (v_3, w_3) \in M$ then $nca(v_1, v_2) = nca(v_1, v_3)$ iff $nca(w_1, w_2) = nca(w_1, w_3)$.

The intuition of this mapping is that subtrees must map to subtrees. In the example shown in figure 2.4 we see that the pair (d, d) is not in the mapping M as it was in the non-contrained mapping because $nca(e, d) \neq nca(e, f)$ in the left tree whereas $nca(e, d) = nca(e, f)$ in the right tree. In other words, e and d have become part of two different subtrees in the right tree.

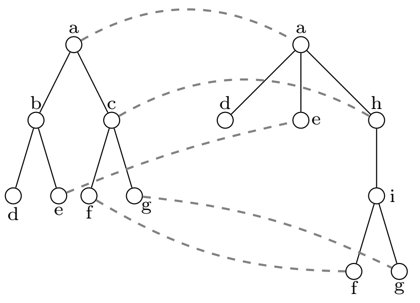


Figure 2.4: An isolated-subtree mapping that corresponds to the edit script *relabel(a,a), delete(b), delete(d), relabel(e,e), relabel(c,h), relabel(f,f), relabel(g,g), insert(d), insert(i).*

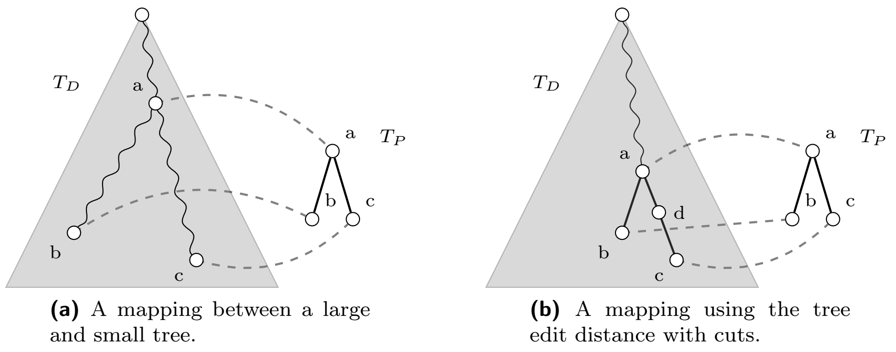
For some applications of tree edit distance there may not be any difference between this and the original mapping. If we know that changes in the data is only relevant to the subtree they occur in, this mapping may even produce more useful results.

Zhang [23] presents a $O(|T_1||T_2|)$ time and space algorithm. Because it is a dynamic programming algorithm, the time complexity is valid for both best, average, and the worst case. However, it relies on a heavily modified version of the recursion from lemma 2.3 on page 10 which makes it quite complex in practice. Richter [12] presents an algorithm very similar to Zhangs but with a different $O\big(degree(|T_1|) \cdot degree(|T_2|) \cdot |T_1||T_2|\big)/O\big(degree(T_1) \cdot height(T_1) \cdot |T_2|\big)$ time/space tradeoff. The worst case of this algorithm is of course when the trees have a very high degree.

2.4 Approximate Tree Pattern Matching

A tree edit distance algorithm based on lemma 2.3 on page 10 can be modified to be used for approximate tree pattern matching. In tree pattern matching we denote the data tree T_D and the pattern tree T_P.

The cut operation was introduced in [22] and enables the algorithm to remove entire subtrees without a cost. This has the effect that the algorithm finds the best mapping among the subtrees of T_D and the pattern rather than finding the best mapping between T_D and T_P as illustrated in figure 2.5.



(a) A mapping between a large and small tree.

(b) A mapping using the tree edit distance with cuts.

Figure 2.5: The effect from introducing the cut operation.

To implement the cut operation, the tree edit distance algorithm must be modified to comply with the following version of the recursion from lemma 2.3. Zhang and Shasha [22] show how to modify their algorithm for the new recursion. The implementation is straightforward. We denote the tree edit distance with cuts used for approximate tree pattern matching for δ_c.

$$
\begin{aligned}
\delta_c(\theta,\theta) &= 0\\
\delta_c(F_1,\theta) &= 0\\
\delta_c(\theta,F_2) &= \delta_c(\theta,F_2-w)+\gamma(\lambda\to w)\\
\delta_c(F_1,F_2) &= \min\begin{cases}\delta_c(\theta,F_2)\\ \delta_c(F_1-v,F_2)+\gamma(v\to\lambda)\\ \delta_c(F_1,F_2-w)+\gamma(\lambda\to w)\\ \delta_c(F_1(v),F_2(w))+\delta_c(F_1-T_1(v),F_2-T_2(w))\\ \quad+\gamma(v\to w)\end{cases}
\end{aligned}
$$

In mathematical terms, the tree edit distance with cuts is a pseudoquasimetric which means $\delta_c(T_1, T_2)$ can be 0 when $T_1 \neq T_2$ and $\delta_c(T_1, T_2)$ can differ from $\delta_c(T_2, T_1)$ [27]. The proviso from this is that we consistently must give the data tree as first argument to the algorithm.

The tree edit distance with cuts reflects changes to the subtree mapped to the pattern as well as the depth of the mapping. Consequently, the deeper the mapping is the less errors we allow within the actual pattern. To deal with this, Zhang, Shasha and Wang [19] present an algorithm which takes variable length don't cares (abbreviated VLDC and also commonly referred to as wildcards) in the pattern into account. It resembles Zhang and Shasha's algorithm and has the same time and space complexity.

Chapter 3

Web Scraping using Approximate Tree Pattern Matching

In this chapter it is shown how to apply approximate tree pattern matching to web scraping. We then discuss the algorithms from the previous chapter in relation to web scraping and give an overview of related work from the litterature.

3.1 Basic Procedure

The layout of a web site, i.e. the presentation of data, is described using Hypertext Markup Language (HTML). An HTML document basically consists of four type of elements: document structure, block, inline, and interactive elements. There is a Document Type Definition (DTD) for each version of HTML which describes how the elements are allowed to be nested*. It is structured as a grammar in extended Backus Naur form. There is a strict and a transitional version of the DTD for backward compatability. The most common abstract model for

*The DTD also describes optional and mandatory attributes for the elements, but this is irrelevant to our use.

HTML documents are trees. An example of a HTML document modelled as a tree is shown in figure 3.1.

```
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <h1>Headline</h1>
    <table>
      <tr>
        <td>a</td>
        <td>b</td>
      </tr>
      <tr>
        <td>c</td>
        <td>d</td>
      </tr>
    </table>
  </body>
</html>
```

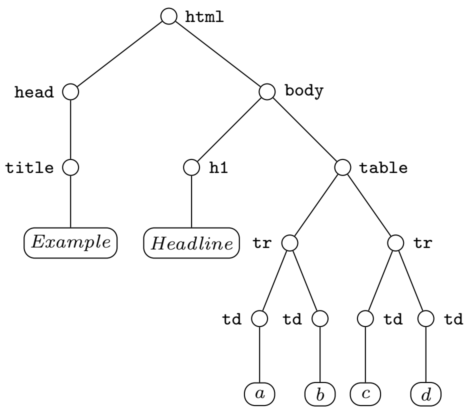


Figure 3.1: A HTML document and its tree model.

Changes in the HTML document affects its tree model so a tree edit distance algorithm can be used to identify structural changes. Furthermore, approximate tree pattern matching can be used to find matchings of a pattern in the HTML tree. For this purpose, the pattern is the tree model of a subset of a HTML document.

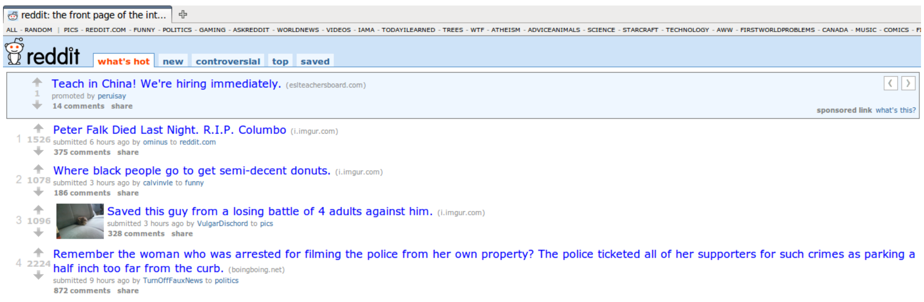
Utilizing the above for web scraping is straightforward. First we have to generate[†] or manually define a pattern. The pattern must contain the target nodes, i.e. the nodes from which we want to extract data. Typically, the pattern could be defined by extracting the structure of the part of the web site we wish to scrape the first time it is visited. Then we parse the HTML and build a data tree. Running the algorithm with the data and pattern trees gives a mapping. We are now able to search the mapping for the target nodes and extract the desired data.

[†]Automatic generation of a pattern is more commonly referred to as *learning* a pattern. Given a set of pages from a web site, a learning algorithm can detect the similarities and create a pattern. Learning is beyond the scope of this project.

3.2 Pattern Design

When using approximate tree pattern matching, the pattern plays an important role to the quality of the result produced by an algorithm. We now discuss how to design patterns for the purpose of web scraping.

We will use Reddit[‡] (`www.reddit.com`) as the running example in our discussion. Consider the screenshot of the front page of Reddit in figure 3.2. It shows a small topic toolbar followed by the Reddit logo and the highest rated entries. Notice that the entry encapsulated in a box with a light blue background is a sponsored entry, as opposed to the other entries, which are submitted by Reddit users.



reddit: the front page of the int...

ALL - RANDOM | PICS - REDDIT.COM - FUNNY - POLITICS - GAMING - ASKREDDIT - WORLDNEWS - VIDEOS - IAMA - TODAYILEARNED - TREES - WTF - ATHEISM - ADVICEANIMALS - SCIENCE - STARCRAFT - TECHNOLOGY - AWW - FIRSTWORLDPROBLEMS - CANADA - MUSIC - COMICS - FI

reddit | what's hot | new | controversial | top | saved

1 Teach in China! We're hiring immediately. (eslteachersboard.com)
promoted by peruisay
14 comments share
sponsored link what's this?

1 1526 Peter Falk Died Last Night. R.I.P. Columbo (i.imgur.com)
submitted 6 hours ago by ominus to reddit.com
375 comments share

2 1078 Where black people go to get semi-decent donuts. (i.imgur.com)
submitted 3 hours ago by calvinvle to funny
186 comments share

3 1096 Saved this guy from a losing battle of 4 adults against him. (i.imgur.com)
submitted 3 hours ago by VulgarDischord to pics
328 comments share

4 2224 Remember the woman who was arrested for filming the police from her own property? The police ticketed all of her supporters for such crimes as parking a half inch too far from the curb. (boingboing.net)
submitted 9 hours ago by TurnOffFauxNews to politics
872 comments share

Figure 3.2: A selection of entries from the frontpage of *Reddit* (2011-06-25).

Say we want to extract the title, URL to the image, and time of submission of the first entry. Then it suffices to use the substructure (shown in figure 3.3 on the following page) of the entries as pattern. This pattern will actually match the first entry that has a picture.

We may not be interested in extracting the sponsored entry. Conveniently, the sponsored entry and the user submitted entries reside in separate `<div>` containers. So to exclusively target the user submitted entries we have to modify the pattern to include an anchor that is long enough for the algorithm to distinguish between the sponsored entry and the user submitted entries. In this case the pattern is given an anchor consisting of two `div` elements. The second

[‡]Reddit is a user driven community where users can submit links to content of the Internet or self-authored texts. The main feature of the site is the voting system which bring about that the best submissions are on the top of the lists.

```html
<div><!-- Entry -->
  <a><img /><!-- Image --></a>
  <div>
    <p>
      <a><!-- Title --></a>
    </p>
    <p>
      <time><!-- Time --></time>
    </p>
  </div>
</div>
```

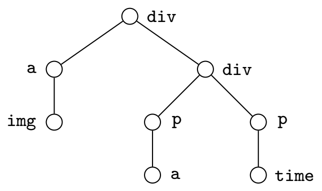


Figure 3.3: A pattern for a Reddit entry and its tree model.

element is given an id attribute to distinguish the container for the sponsored entries from the container for user entries. It is shown in figure 3.4.

```html
<div>
  <div id="siteTable">
    <div><!-- User entry -->
      <a><img /><!-- Image --></a>
      <div>
        <p>
          <a><!-- Title --></a>
        </p>
        <p>
          <a><!-- Time --></a>
        </p>
      </div>
    </div>
  </div>
</div>
```

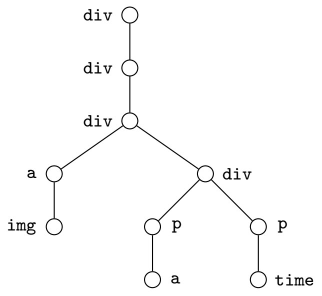


Figure 3.4: A pattern with an anchor for a Reddit entry and its tree model.

The main disadvantage of adding an anchor is that the pattern becomes larger which affects the execution time of the algorithm. Using attributes in the pattern also faces the risk that the value may be changed by the web designer. If the id is changed in this example, the pattern will match the sponsored entry just as well.

3.3 Choosing an Algorithm

Having decided on using approximate tree pattern matching for web scraping, there is still a selection of algorithms to choose from.

The most suitable algorithm for the tree edit distance with cuts to use for HTML trees is Zhang and Shasha's algorithm because its running time depends on the height of the trees, which commonly is low compared to the total number of nodes for HTML trees. Some samples of this is shown in table 3.1.

web site	Type	# nodes	Height
`google.com`	Simple	142	13
`pol.dk`	News	3414	19
`berlingske.dk`	News	3151	21
`reddit.com`	User driven news	1935	13
`python.org`	Simple dynamic	259	10
`blog.ianbicking.org`	Custom blog	1105	14
`crossfitmobile.blogspot.com`	Blogspot blog	1286	26
`rjlipton.wordpress.com`	Wordpress blog	842	11

Table 3.1: The number of nodes and the height of the DOM trees of selected web sites (2011-07-17).

Zhang, Shasha, and Wang's VLDC algorithm allows wildcards in the pattern which can be an advantage. When using the tree edit distance with cuts for approximate tree pattern matching, the depth of the match in the data tree influence the result. If the pattern is small, Zhang and Shasha's algorithm may find a match in the top of the data tree by relabeling a lot of nodes. To circumvent this, we can add a wildcard as the root of the pattern and use the VLDC algorithm for matching. Wildcards can also be used to develop more advanced patterns. The reason we choose not to work with this algorithm is because wildcards eliminate the possibility of applying the methods we later derive to speed up the algorithms.

Algorithms for the top-down mapping are unfit because they require the pattern to start at the `<html>` tag and if there are changes to the part of the HTML tree that matches the internal nodes of the pattern, the algorithm will not produce a useful mapping.

The isolated-subtree algorithm will not locate and match a subtree if it is split into several subtrees or merged into one. Consider the example from Reddit.

Assume that a new visualization of the entries, where the second `div` is superflous, is introduced. This corresponds to merging the two subtrees of the root of the pattern into one. As shown in figure 3.5 on the next page, the `img` tag will not be located when using the isolated-subtree mapping.

3.4 Extracting Several Matches

Until now we have assumed that we only want to extract the optimal match. However, it is common that we want to extract several matches. Continuing with Reddit as example we may want to extract all entries from the front page. In this section we discuss how to achieve this using approximate tree pattern matching.

If we know how many entries we want to extract, one approach is to create a pattern that matches the required number of entries. This also applies if we want to extract the n^{th} entry. Then we create a pattern matching the first n entries and discard the results for the first $n-1$ entries. The drawback of this approach is that the pattern quickly becomes big and slows down the algorithm. Also, when creating a pattern for n entries we are not guaranteed that it will match the first n entries. It will match some n entries.

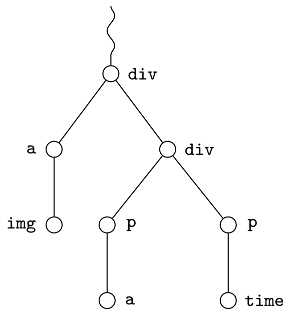
The above mentioned method is not applicable if we want to extract all entries without knowing the exact number of entries. An alternative approach is to create a pattern matching one entry. After matching the pattern we remove the nodes in the mapping from the data tree. This is repeated until the cost of the mapping exceeds some predefined threshold.

The main disadvantage of this approach is that the algorithm has to be invoked several times. Furthermore, if the goal is to extract the first n entries we cannot guarantee that the pattern matches the first n entries in n invocations of the algorithm.

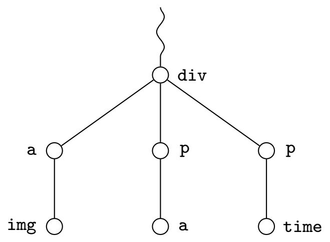
3.5 Related Work

The litterature on information extraction from web sites is vast. However, the focus is mostly on learning patterns and information extraction[§] from data. We

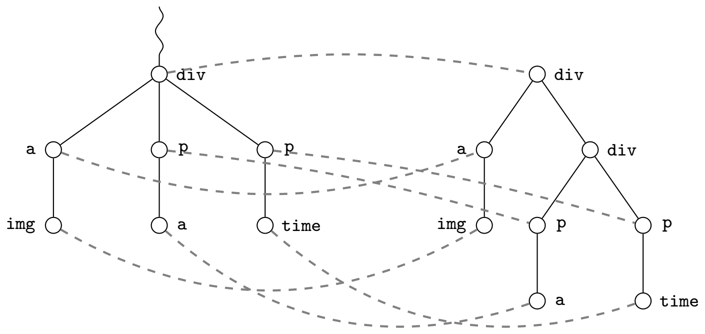
[§]Information extraction is extraction of meaningful content from web sites without explicitly specifying where the information reside.



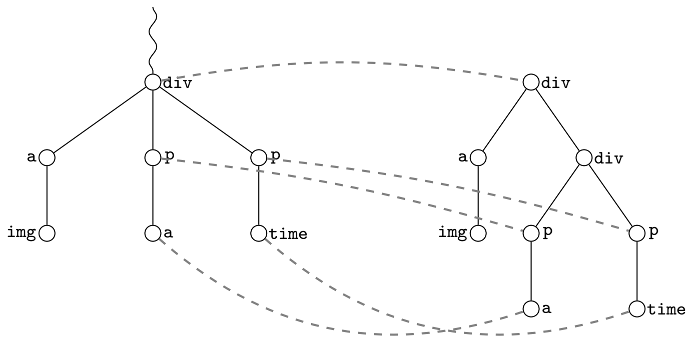
(a) A Reddit entry before the change. Subtree is used as pattern.



(b) A Reddit entry after the change.



(c) The optimal mapping as produced by e.g. Zhang and Shasha's algorithm.



(d) The isolated-subtree mapping.

Figure 3.5: Example of the difference in outcome from using an original mapping and an isolated-subtree mapping.

will review the known results from using approximate tree pattern matching in this context.

Xu and Dyreson [20] present an implementation of XPath which finds approximate matches. The algorithm is based on the Apache Xalan XPath evaluation algorithm, but the criteria for the algorithm is that the result must be within k edits of the HTML tree model. The algorithm runs in $O\big(k \cdot |T_P||T_D| \log |T_D|\big)$ time, but bear in mind that this is for evaluating a path. To simulate matching a tree pattern we need several paths [28] and the running time becomes $O\big(k \cdot |T_P|^2|T_D| \log |T_D|\big)$ (worst case), which is worse than Zhang and Shasha's algorithm. Also, it matches each path of the pattern separately so it does not obey the mapping criterias we know from the tree edit distance.

Reis *et al.* [11] presents an algorithm for extraction of information from sets of web sites. Given a set of web sites they generate a pattern with wildcards. Using the pattern their algorithm computes a top-down mapping, but unlike other algorithms for the top-down mapping it does not compute irrelevant subproblems based on a max distance k given as input. The worst case running time is $O\big(|T_1||T_2|\big)$ but due to the deselection of irrelevant subproblems, the average case is a lot faster. Their algorithm works well because their pattern generation outputs a pattern which is sufficiently restricted to be used with a top-down algorithm.

Based on the claim that information extraction or web scraping using tree pattern matching is flawed, Kim *et al.* [8] suggests a cost function that takes the rendered size of tags into account when comparing them. They use the cost function with an algorithm for the top-down mapping.

3.6 Summary

The most suited algorithm for web scraping is Zhang and Shasha's algorithm, but we anticipate that it will be slow on large webpages. The Reddit example shows that there is a case where Zhang's algorithm for the isolated-subtree mapping fails to locate the targeted data.

The speed issue is not adressed directly in the litterature because sub-optimal algorithms are used in return for pattern constraints. The remainder of this thesis will focus on speeding up the algorithm aiming at attaining a more versatile solution than found in the litterature.

Chapter 4

Lower Bounds for Tree Edit Distance with Cuts

In this chapter we describe six methods for approximating the tree edit distance with cuts. The methods are all lower bounds for the tree edit distance with cuts. The methods are derived from attempts to approximate the tree edit distance (without cuts) in the litterature.

The methods are all based on the assumption that a unit cost function is used, and we will use D to denote the approximations of the tree edit distance with cuts.

4.1 Tree Size and Height

When a unit cost function is used, the edit distance between two trees is bounded from below by the difference in the size of the trees. Consider two trees T_1 and T_2 where $|T_1| > |T_2|$. To change T_1 to T_2 at least $|T_1| - |T_2|$ nodes must be removed from T_1. Similarly, if T_2 is a larger tree than T_1, at least $|T_2| - |T_1|$ nodes must be inserted in T_1 to change it to T_2.

When using the tree edit distance for approximate tree pattern matching we include the cut operation, so some nodes may be deleted from T_1 without affecting the cost. Therefore, it is impossible to determine, based on the size of the trees, if T_2 can be obtained by using only cut on T_1. As a result, the lower bound D_{size} is:

$$D_{size}(T_1, T_2) = \max\bigl(0, |T_2| - |T_1|\bigr) \tag{4.1}$$

Assuming the nodes of a tree T have postorder indices, then the nodes of any subtree $T(v)$ is a consecutive sequence of indices starting from the index of the leftmost leaf of the subtree to the index of the root of the tree. So the size of a tree can be found from $|T(v)| = v - lml(v)$, which can be found in constant time after a $O(|T|)$-preprocessing of the tree.

Alternatively, the height of the trees can be used as a lower bound as shown below. This can produce a tighter lower bound for trees that may be very similar in terms of size but otherwise look very different.

$$D_{height}(T_1, T_2) = \max\bigl(0, height(T_2) - height(T_1)\bigr) \tag{4.2}$$

4.2 Q-gram Distance

The q-gram is a concept from string matching theory. A q-gram is a substring of length q. The q-gram distance between two strings x and y is defined as follows. Let Σ be the alphabet of the strings and Σ^q all strings of length q in Σ. Let $G(x)[v]$ denote the number of occurrences of the string v in the string x. The q-gram distance for strings d_q is obtained from the following equation [17].

$$d_q(x, y) = \sum_{v \in \Sigma^q} \bigl|G(x)[v] - G(y)[v]\bigr| \tag{4.3}$$

If $d(x, y)$ is the optimal string edit distance between x and y, the q-gram distance can be $0 \leq d_q(x, y) \leq d(x, y) \cdot q$, given that unit cost is used for all operations on the strings. So the q-gram distance may exceed the optimal string edit distance. To use the q-gram distance as a lower bound for the string edit distance, we can select a disjoint set of q-grams from x. Then one character in the string can only affect one q-gram and thereby only account for one error in the q-gram distance. As a result $0 \leq d_q(x, y) \leq \frac{|x|}{q} \leq e$.

We generalize the q-gram distance to trees by using subpaths of a tree as q-grams. In one tree we select all subpaths of size 1 to q. In the other tree we ensure that the tree q-gram distance is a lower bound for the tree edit distance by selecting a disjoint set of q-grams (when disjoint we call them q-samples) of size at most q.

We select q-grams of different sizes to be able to include any node in at least one q-gram and thereby make the lower bound on the tree edit distance tighter. If $q > 1$ and the tree has more than $2q$ nodes, there will be more than one way to select a disjoint set of q-grams. Therefore, the algorithm which selects q-grams may influence on the q-gram distance for trees.

The q-gram distance can be extended to allow for the cut operation. A q-gram that appears in T_1 and not T_2 may not be accounted as an error, whereas it may if it appears in T_2 and not in T_1. So the tree q-gram distance with cuts is

$$D_q(T_1, T_2) = \sum_{Q \in T_2} \max\big(0, G(T_2)[Q] - G(T_1)[Q]\big) \tag{4.4}$$

In practice, we only have to iterate the q-grams in T_2 to make the above computation. Since the q-grams of T_2 are disjoint, there can be at most $|T_2|$ of them. Thus, the tree q-gram distance with cuts can be computed in $O(|T_2|)$ time (assuming the q-grams have been computed in a pre-processing step).

Having defined the q-gram distance with cuts, we now show that it is a lower bound for the tree edit distance with cuts if the q-grams in T_2 are disjoint.

Lemma 4.1 *Let $D_q(T_1, T_2)$ be the tree q-gram distance with cuts for two trees T_1 and T_2, and let the q-grams in T_2 be disjoint. Then*

$$D_q(T_1, T_2) \leq \delta_c(T_1, T_2) \tag{4.5}$$

Proof. If $\delta_c(T_1, T_2) > 0$ then there is at least one node v in T_2 which is not in T_1. Let v be part of a q-gram in T_2 which is not in T_1. The only free operation that makes structural changes to T_1 is cut. If we remove a leaf we reduce the set of q-grams in T_1 and then the q-gram with v is still not in T_1. Therefore, we need k operations, where $1 \leq k \leq q$, to create the q-gram of T_2 in T_1. If c is the number of q-grams in T_2 which are not in T_1 then we have $c \leq \delta_c(T_1, T_2) \leq c \cdot q$, and since each q-gram can account for only one error we have $D_q(T_1, T_2) = c$. (4.5) clearly follows from this. If $\delta_c(T_1, T_2) = 0$ then T_2 is a subtree in T_1 and any q-gram in T_2 is also in T_1. $\square$

The chosen value of q will affect the outcome of (4.4). A small value for q may produce a tight bound for dissimilar trees. If the trees do not have a lot in common, we want as many q-grams as possible in order to capture the differences. For similar trees, a larger value for q may produce a tight bound. Larger q-grams contain more information about the structure of the tree. If the trees are similar, and the q-grams are small, we risk missing an error.

We may also select overlapping q-grams from T_2 if we bound the number of q-grams a node is allowed to be part of and subsequently divide the distance by the bound. However, there may be q-grams not present in T_1 due to nodes which are only in one q-gram in T_2, and we anticipate that the extra q-grams in T_2 do not make up for this.

Algorithms for computing q-grams and q-samples for a tree are given in appendix B.1 on page 83 and appendix B.2.

4.3 PQ-gram Distance

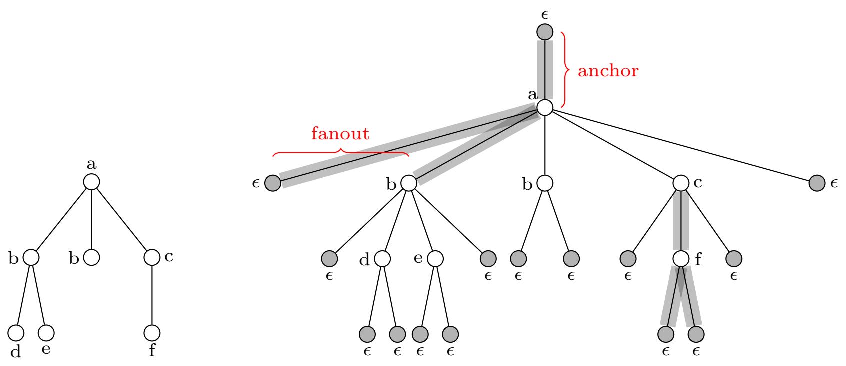
Augsten *et al.* [2] present another generalization of the q-gram distance which is based on pq-grams. A pq-gram is a subtree which consists of a path of length p from its root to an internal node v. This is the anchor. The node v has q children. This is the fanout. When referring to a specific set of pq-grams where the parameters are set to for instance $p = 2$ and $q = 3$ we call them 2,3-grams. We describe their method and show why it can not be adapted to the tree edit distance with cuts.

The generalization of q-grams to subpaths only captures information about the parent/child relationship between nodes. The advantage of the pq-gram is that it is possible to capture more structural information than with q-grams, because each pq-gram holds information of the relation between children of a node. To enable us to select as many pq-grams as possible a tree is extended such that

- the root has $p - 1$ ancestors,
- every internal node has an additional $q - 1$ children before its first child,
- every internal node has an additional $q - 1$ children after its last child,
- every leaf has q children

The resulting tree is called the $T^{p,q}$-extended tree of T. An example of a tree, its extended tree, and a disjoint set of pq-grams is seen in figure 4.1 on the next page.

The pq-gram distance is computed the same way as the q-gram distance for strings, and Augsten *et al.* prove that it is a lower bound for the fanout tree edit distance, which is obtained from using a cost function that depends on the degree of the node to operate on.



(a) T. **(b)** $T^{2,2}$, the 2,2-extension of T.

Figure 4.1: An example of a tree extended for the PQ-gram distance. Gray nodes are new nodes. ϵ is a special character for labels on new nodes. The highlighted subtrees are examples of 2,2-grams selected from $T^{2,2}$.

We now show that the lower bound obtained from using the pq-gram distance always is zero. There are three possible outcomes for each pq-gram in either T_1 or T_2.

1. A pq-gram in T_1 is not in T_2. For all we know, the nodes of the pq-gram in T_1 may be removed using the cut operation, so from this case we can not tighten the lower bound.
2. A pq-gram in T_2 is not in T_1. Consider the case where removing a node from the fanout of pq-gram in T_1 generates the subtree that equals the pq-gram from T_2. Since we do not know if removing a node from the fanout is free due to cuts, this case does not tighten the lower bound either.
3. A pq-gram in T_1 is also in T_2. This is not an error so it does not add to the lower bound.

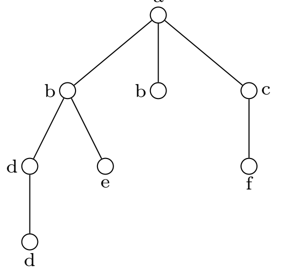
4.4 Binary Branch Distance

Yang *et al.* [16] present a tree edit distance approximation called the binary branch distance. It is based on pq-grams and the following observation. Any edit operation can influence the parent/child relation between arbitrary many nodes whereas it can only influence exactly two sibling relations. By converting the trees to their binary tree representations and selecting all possible 1,2-grams, the authors show that the pq-gram distance (of the transformed problem) is at most 4 times the tree edit distance. The result is thus a lower bound which is found without having to select a disjoint set of pq-grams from one of the trees. We describe their method and show why it can not be extended to include the cut operation.

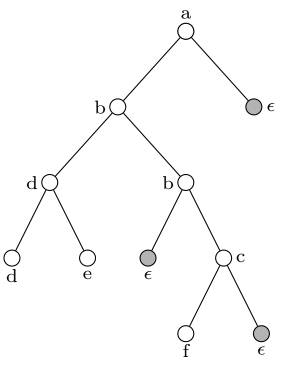
Any tree (or forest) has a binary tree representation, which preserves all structural information, i.e. all parent/child and sibling relations [10]. The conversion is best described using following algorithm.

1. Add an edge from each node to its left and right sibling (if any)
2. Remove edges from each node to all but its first child
3. Add empty nodes such that each node has exactly zero or two children (as it is required in a binary tree)

An example of a transformation of a tree is seen in figure 4.2. The transformation of the trees is neccessary to ensure that a node is in at most two 1,2-grams. Without the transformation we could only use the above observation to select q-grams vertically among siblings, and therefore lose information about the parent/child relations.



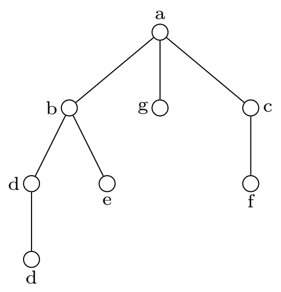
(a) T.



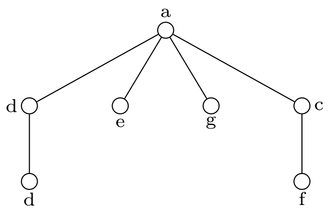
(b) The binary tree representation of T.

Figure 4.2: An example of a binary tree representation.

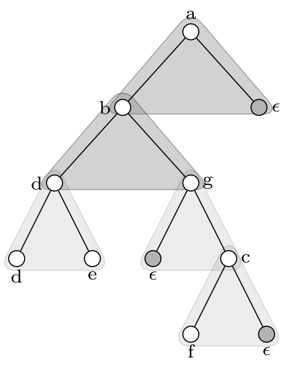
We now show by example that one edit operation can affect the pq-gram distance by at most four*. Consider the example in figure 4.3 where the internal node b is deleted. The two 1,2-grams (a, b, ϵ) and (b, d, g) that contains b will no longer be present in the new tree. Instead the new tree will contain the 1,2-grams (a, d, ϵ) and (e, ϵ, g) which were not present in the original tree. Therefore, the pq-gram distance will be 4. Deletion and insertion is symmetric so the same argument also holds for insertion. Relabeling a node also affects at most four 1,2-grams because a node can be included in at most two 1,2-grams.



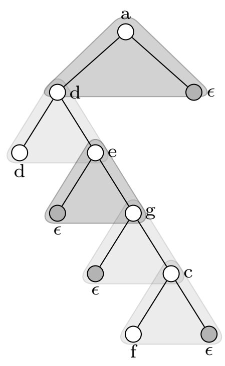
(a) T.



(b) T after deleting b.



(c) Binary tree representation of T.



(d) Binary tree representation of T after deleting b.

Figure 4.3: An example of 1,2-grams affected by a delete operation on a tree. The lightly shaded areas are shared 1,2-grams. The darkly shaded areas are individual 1,2-grams. In this example the pq-gram distance is 4 and the binary branch distance is thus 1.

*In the paper presenting this method it is a factor of five instead of four. This is because of additional ϵ-nodes added as children of each leaf. This encodes information about the leaves in the pq-grams. We omit it here for simplicity.

The binary branch distance is potentially a tighter lower bound of the tree edit distance than the q-gram and pq-gram distance because it allows us to select overlapping pq-grams from both trees and because it captures sibling relations. However, since the distance must be divided by 4 we need to select 4 times as many pq-grams as we would select disjoint q-grams.

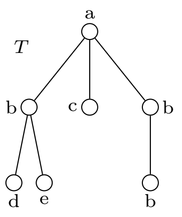
However, the binary branch distance can not be used to produce a lower bound for the tree edit distance with cuts. This is evident from the three possible outcomes for a 1,2-gram in either T_1 or T_2.

1. A pq-gram in T_1 is not in T_2. The nodes of the pq-gram may be part of a subtree that can be cut away, so this can not be used to tighten the lower bound.
2. A pq-gram in T_2 is not in T_1. Consider the tree T of figure 4.3 on the preceding page. Assume that the node a has a child v between g and c. Then the 1,2-grams (g, ϵ, c) and (c, f, ϵ) would not be present in T_2. However, if v is cut away, which is free, then the two pq-grams becomes present in T_1. Therefore, a pq-gram in T_2 and not T_1 can not count as an error.
3. A pq-gram in T_1 is also in T_2. This is not an error so it does not add to the lower bound.

4.5 Euler String Distance

Hierarchical structured data is commonly represented as trees, but may also be represented as a parenthesized string called the Euler string. The parentheses are used to maintain parent/child relationships. Consider for instance the tree in figure 4.4 on the next page whose Euler string representation is `a(b(de)cb(b))`. The Euler string is computed by doing a postorder traversal of the nodes. It can be computed in $O(|T|)$ time and the tree can be restored again in $O(|T|)$ time. However, serialization of tree data has the disadvantage that parent/child relations no longer can be determined in constant time.

Another variant of the Euler string omits parentheses and uses a special inverted character when backtracking from a node. An example is shown in figure 4.4 on the facing page. The special characters must not be a part of the set of all possible labels. This variant is more suited for comparison of the strings because the number of extra characters, i.e. parentheses, is independent of the tree structure. We will use this variant and we denote the Euler string of a tree T for $s(T)$.



$s(T) = \mathtt{bd}\overline{\mathtt{d}}\mathtt{e}\overline{\mathtt{eb}}\mathtt{c}\overline{\mathtt{c}}\mathtt{b}\overline{\mathtt{bb}}$

Figure 4.4: A tree and its Euler string.

The string edit distance of two Euler strings can be used as a lower bound for the tree edit distance based on the following observation by Akutsu [1]. Any operation on a tree T affects at most two characters in $s(T)$, so we have the following theorem.

Theorem 4.2 ([1]) *Let T_1 and T_2 be ordered, rooted trees and let $s(T)$ denote the Euler string of a tree T. Let $d(x, y)$ denote the string edit distance of two strings x and y and $\delta(T_1, T_2)$ the tree edit distance of the trees. Then we have*

$$\frac{1}{2} d\big(s(T_1), s(T_2)\big) \leq \delta(T_1, T_2) \tag{4.6}$$

For two strings of length m and n, computing the string edit distance can be done in $O(mn)$ time [5]. Since $|s(T)| = O(T)$ this method can compute an approximation of the tree edit distance in $O(|T_1||T_2|)$ time.

We will now discuss approaches to convert this method to give lower bounds for the tree edit distance with cuts. We will discuss

- making all *delete* operations free in the string edit distance algorithm,
- postprocessing the result from the string edit distance algorithm,
- and modifying the string edit distance algorithm to handle cuts.

Making delete operations free for the string edit distance algorithm is the only effective way of adapting this method to act as a lower bound for the tree edit distance with cuts. We start by showing that for any mapping obtained using an algorithm for the tree edit distance with cuts, there is a mapping, produced by a regular tree edit distance algorithm, with the same cost or cheaper if using a cost function where deletes are free.

Lemma 4.3 *Let T_1 and T_2 be ordered, rooted trees. Let $\delta(T_1, T_2)$ be the tree edit distance between T_1 and T_2 using a cost function γ where deletes are always free, i.e. $\gamma(v \to \lambda) = 0, \forall v \in V(T_1)$. Let $\delta_c(T_1, T_2)$ be the tree edit distance with cuts. We then have*

$$\delta(T_1, T_2) \leq \delta_c(T_1, T_2) \tag{4.7}$$

Proof. Assume that $\delta(T_1, T_2) > \delta_c(T_1, T_2)$ for some two trees T_1 and T_2. Then there is an edit script E for $\delta_c(T_1, T_2)$ consisting of a inserts, b deletes, c relabels, and d cuts. Their accumulated cost is c_a, c_b, c_c, and c_d, respectively. The cost of E is $c_a + c_b + c_c$ because $c_d = 0$. If cuts are not available we would have to replace cuts by deletes, so the cost of an edit script that produce the same mapping as E would be $c_a + c_b + c_c + c_d$. If deletes are free it is $c_a + c_c$. From our assumption we have that $c_a + c_c > c_a + c_b + c_c$, which is a contradiction because costs are non-negative, cf. the cost function is a distance metric. □

We now establish that the string edit distance, where the delete operation is without a cost, of the Euler strings of two trees is a lower bound for the tree edit distance with cuts.

Lemma 4.4 *Let $d(x, y)$ be the string edit distance and $\delta_c(T_1, T_2)$ the tree edit distance with cuts. If we use a cost function for the string edit distance where delete operations are free then half the string edit distance is a lower bound of the tree edit distance with cuts, i.e.*

$$\frac{1}{2} d\big(s(T_1), s(T_2)\big) \leq \delta_c(T_1, T_2) \tag{4.8}$$

Proof. Since theorem 4.2 on the previous page holds for cost functions that qualify as distance metrics, and the cost function where delete operations are free is a distance metric, we can combine theorem 4.2 and lemma 4.3 to obtain this lemma.

Thus, the tree Euler string edit distance D_e is

$$D_e(T_1, T_2) = \frac{1}{2} d\big(s(T_1), s(T_2)\big) \tag{4.9}$$

The remainder of this section will discuss why the two other approaches will not produce a lower bound as required.

We would like to be able to correct for deletes in a postprocessing phase. Here is an approach. Once a string edit distance has been computed, the sequence of edit operations can be extracted using backtracking in the dynamic programming table. From the sequence of operations it is possible to compute a mapping between characters of the two strings. Assume the data string is split into a set of substrings by the characters in the mapping. For each of these substrings we look for a pair of a character and its inverted character, e.g. $\mathsf{a}\bar{\mathsf{a}}$. This corresponds to a leaf and can be removed without a cost. This is repeated for each substring until no more such pairs exist. The number of characters removed is subtracted from the string edit distance and the result is a lower bound for the tree edit distance with cuts. Unfortunately, we can not guarantee that the resulting mapping is equal to the mapping an algorithm for the tree edit distance with cuts would produce. This means that there may be fewer pairs of characters to remove in the postprocessing phase, and therefore this approach can not be used to compute a lower bound.

Modifying the string edit distance algorithm to handle cuts is not possible either. As mentioned earlier, information about the parent/child relations is lost when tree data is serialized. An algorithm will consider one character of the data string at a time. To determine if we can cut away a character, the algorithm must read at least one other character, which it will have to scan through the string to find.

4.6 Summary

In this chapter we have seen six techniques for computing a lower bound on the tree edit distance with cuts. The binary branch distance and the pq-gram distance always yield 0 as a lower bound, so these can be discarded of for our further studies. Table 4.1 shows a summary of the time and space requiremens of the techniques.

Technique	Preprocessing	Space	Time
Size	$O(\|T_1\| + \|T_2\|)$	$O(\|T_1\| + \|T_2\|)$	$O(1)$
Height	$O(\|T_1\| + \|T_2\|)$	$O(\|T_1\| + \|T_2\|)$	$O(1)$
Q-gram distance	$O(\|T_1\|^2 \cdot q + \|T_2\|^2)^\dagger$	$O(\|T_1\| \cdot q + \|T_2\|)$	$O(\|T_2\|)$
Euler string distance	-	$O(\|T_1\|\|T_2\|)$	$O(\|T_1\|\|T_2\|)$

† See appendix B.1 on page 83.

Table 4.1: Preprocessing, space, and time requirements of the lower bounds methods.

CHAPTER 5

Data Tree Pruning and Heuristics

5.1 Adapting the Fast Unit Cost Algorithm to Cuts

In [14] Shasha and Zhang present a tree edit distance algorithm which is fast for similar trees, i.e. when the tree edit distance is small. It is fast because it assumes that a unit cost function is used, and based on that, some subproblems can be ruled out while the algorithm still computes an optimal solution. In this section we adapt the algorithm to cuts.

Not as many subproblems can be ruled out after the adaption. However, the algorithm serves as inspiration to our pruning approach. Pruning of the data tree means reducing its size prior to running the actual algorithm. The difference between pruning and ruling out subproblems at runtime is subtle but notable. Pruning a branch of the data tree corresponds to ruling out all subproblems containing any nodes in the subtree, so ruling out subproblems at runtime causes fewer subproblems to be computed. Instead, pruning offers two other advantages. We can apply more elaborate methods to decide if a subtree is relevant because we do not have to do it at runtime. And, as we shall see, we can use keyroots when the tree is pruned prior to running the algorithm, which is not possible when ruling out subproblems at runtime.

5.1.1 Algorithm Description

The fast unit cost algorithm by Shasha and Zhang is based on the following two observations.

1. The tree edit distance between two trees is at least the difference in the size of the trees.
2. If the mapping between two subtrees $T_1(v)$ and $T_2(w)$ is part of an optimal solution, then the mapping between the forests to the left of the subtrees is also a part of the optimal solution*.

The observations are combined to rule out subproblems in the permanent dynamic programming table as follows. The algorithm is given a threshold k as input. A subproblem in the permanent table is ruled out if the difference in the size of the subtrees plus the difference in the size of their left subforests exceeds the threshold k. This is formalized in the following lemma.

Lemma 5.1 ([14]) *Let T_1 and T_2 be two trees where the nodes are given postorder indices. When computing the tree edit distance $\delta(T_1, T_2)$ using a unit cost cost function, the minimum cost of the subproblem $\{T_1(v_i), T_2(w_j)\}$ is $|(i-i')-(j-j')| + |i'-j'|$, where $v_{i'} = lml(v_i)$ and $u_{j'} = lml(w_j)$.*

Proof. From the relabel case of lemma 2.3 on page 10 we know that if the mapping between $T_1(v_i)$ and $T_2(w_j)$ is part of an optimal mapping, then the mapping of the forests $v_0, \ldots, v_{i'}$ and $w_0, \ldots, w_{j'}$ must also be part of the optimal mapping. Since we use a unit cost cost function and the nodes have postorder indices, we know that the cost of the mapping between $T_1(v_i)$ and $T_2(w_j)$ is at least $\big||T_1(v_i)| - |T_2(w_j)|\big| = \big|(i-i')-(j-j')\big|$, and the cost of mapping $v_0, \ldots, v_{i'}$ and $w_0, \ldots, w_{j'}$ is at least $|i'-j'|$. □

In practice, the lemma is implemented such that the algorithm only iterates the subproblems where the minimum cost of the subproblem does not exceed k. In the temporary array some subproblems can also be ruled out based on observation 1. Furthermore, since we have to save some operations for the mapping of the left subforests, the threshold used for the temporary array can

*This is evident from the last case of the recursion in lemma 2.3 on page 10.

be tightened to $k - |i' - j'|$ for a subproblem $\{T_1(v_i), T_2(w_j)\}$ where $v_{i'} = lml(v_i)$ and $u_{j'} = lml(w_j)$.

The running time of the algorithm is

$$O\big(k^2 \cdot \min(|T_1|, |T_2|) \cdot \min(leaves(T_1), leaves(T_2)\big)$$

For small values of k, the algorithm is faster than the keyroot algorithm. It is not possible to rule out trees based on lemma 5.1 when employing keyroots because we risk missing the optimal solution. An example is shown in figure 5.1 where the tree edit distance is 1. Now assume that the unit cost algorithm is invoked with $k = 1$. Then the subproblem $\{T_1(v_3), T_2(w_2)\}$ is never computed because $\{T_1(v_3), T_2(w_3)\}$ is ruled out. However, $\{T_1(v_3), T_2(w_2)\}$ is needed for the optimal solution when computing $\{T_1(v_4), T_2(w_3)\}$, so the algorithm returns a sub optimal solution.

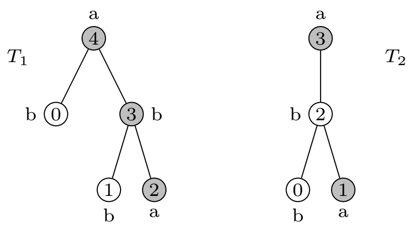


Figure 5.1: A problem which shows that keyroots (the filled nodes) can not be used with the unit cost algorithm.

5.1.2 Adaption to Cuts

Introducing the cut operation means that some nodes can be removed from T_1 without cost. Consequently, the minimum cost from lemma 5.1 on the preceding page is no longer symmetric. In other words, if $|T_1(v_i)| > |T_2(w_j)|$ the tree edit distance for these subtrees may be 0. So the minimum cost of a subproblem $\{T_1(v_i), T_2(w_j)\}$ is

$$\max\big(0, (j - j') - (i - i')\big) + \max\big(0, j' - i'\big) \tag{5.1}$$

where $v_{i'} = lml(v_i)$ and $w_{j'} = lml(w_j)$.

The lack of symmetry means that a lot fewer subproblems can be ruled out in the permanent dynamic programming table. The immediate consequence is the same for the temporary table, however, since the temporary table is monotonically non-decreasing along its diagonals, we can rule out subproblems based on not only the estimated cost of solving it, but also the estimated cost of getting to it.

Consider figure 5.2 which shows the temporary dynamic programming table for a subproblem $\{T_1(v_i), T_2(w_j)\}$. Let m and n ($0 \leq m \leq |T_1(v_i)|$ and $0 \leq n \leq |T_2(w_j)|$) be the indices of some subproblem in this instance of the temporary table. A path from (m, n) to $(0, 0)$ is a solution to the subproblem at (m, n). The path with the lowest cost is the optimal solution. When estimating the cost we assign the following cost to the path. Going diagonal is free because relabeling a pair of nodes may be free. Going up is also free because removing nodes from $T_1(v_i)$ may be free due the cut operation. Going left corresponds to the insert operation, which has cost 1.

Until now we have ruled out a subproblem in the temporary table based on the minimum cost of solving it, i.e. $\max(0, n - m)$ when cuts are allowed. This cost corresponds to the path with the shortest Euclidian distance from (m, n) to $(0, 0)$ in the table (the gray path in figure 5.2). However, we also know that the subproblem at (m, n) is used to ultimately solve the problem at $(|T_1(v_i)|, |T_2(w_j)|)$, so in order to get from $(|T_1(v_i)|, |T_2(w_j)|)$ to (m, n) we may have used some operations. This corresponds to the black path in figure 5.2, and we estimate the minimum cost of it to be

$$\max\big(0, |T_2(u_j)| - (|T_1(v_i)| - m) - n\big) \tag{5.2}$$

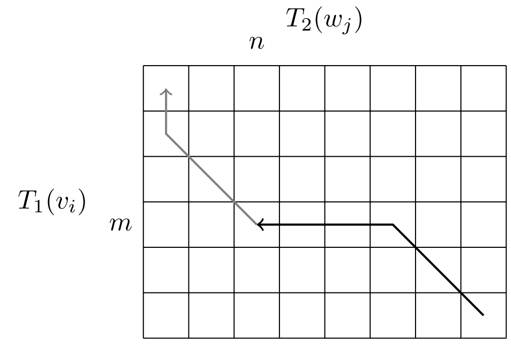


Figure 5.2: A possible solution path to a subproblem in the temporary dynamic programming table in Zhang and Shasha's algorithm.

We formalize the above estimates in the following lemma.

Lemma 5.2 *Given the subproblem $\{T_1(v_i), T_2(w_j)\}$, let m and n, $0 \leq m \leq |T_1(v_i)|$ and $0 \leq n \leq |T_2(w_j)|$, be the indices of a subproblem in the temporary* *dynamic programming table of Zhang and Shasha's algorithm. The minimum cost of the subproblem* $\{T_1(v_i), T_2(w_j)\}$ *then is*

$$\max\bigl(0, |T_2(w_j)| - (|T_1(v_i)| - m) - n\bigr) + \max\bigl(0, n - m\bigr) \tag{5.3}$$

Proof. The best way to get from the n^{th} column to the 0^{th} column is n relabels which may be free. We can at most perform $|n-m|$ relabels, so if $n - m > 0$ we require $n - m$ inserts, which have unit cost, otherwise we require $m - n$ cuts, which are free. Same argument holds for the best way from $(|T_1(v_i)|, |T_2(w_j)|)$ to (m, n). □

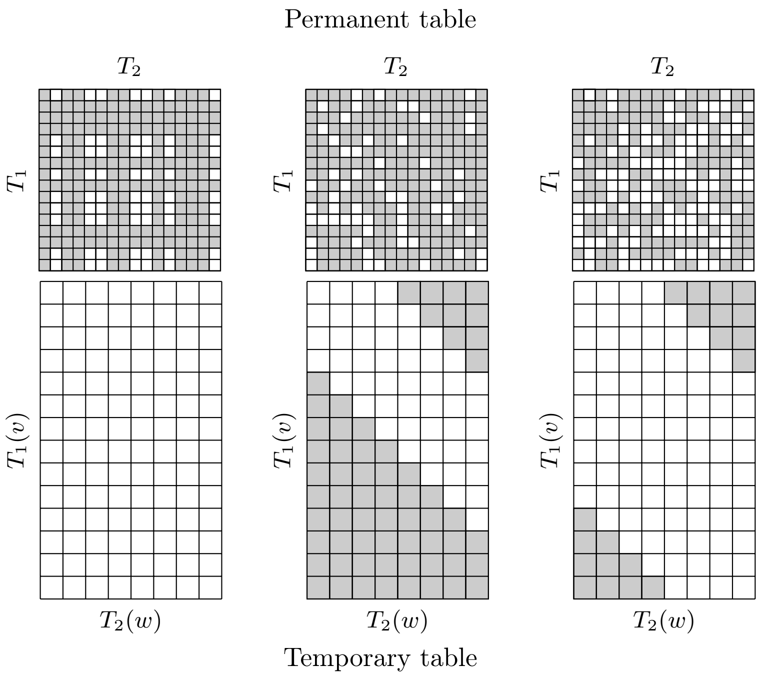
Adapting the algorithm to cuts has some drawbacks. As mentioned earlier, a lot fewer subproblems are ruled out from the permanent table. This is also the case in the temporary table in spite of our attempts to tighten it up with lemma 5.2. The main drawback is in fact in the temporary table where only subproblems of the upper right and lower left corners are ruled out, so the impact diminishes as the difference in the size of the subtrees grows bigger.

In figure 5.2 on the preceding page a graphical comparison of the subproblems computed by the algorithms is shown. We see that a lot fewer subproblems are needed in (b) compared to (c). If we were to show another temporary table where $T_1(v)$ is a lot larger than $T_2(w)$, then it would be the same number of subproblems required for (b) whereas it would be the same number of subproblems ruled out in (c). Clearly, this will have a negative effect on the execution time of our adaption of the algorithm, because the size of a subtree in T_1 generally exceeds that of T_2 in a pattern matching context.

5.2 Using Lower Bounds for Pruning

In the previous section we saw how to rule out subproblems based on a lower bound obtained from the difference in the size of the components of the subproblem. We anticipate that the unit cost algorithm adapted for cuts does not outperform Zhang and Shasha's original algorithm, so in this section we describe a technique for pruning the data tree using the lower bound methods from chapter 4.

We would like to remove subtrees that under no circumstances are a part of an optimal mapping. If there is a pair of subtrees for which the lower bound on their tree edit distance with cuts is less than k, the given subtree in the data tree can not be removed. Using lower bounds ensure that we do not remove subtrees that are part of an optimal mapping unless the cost of the optimal



(a) Algorithm using keyroots.

(b) Fast unit cost algorithm.

(c) Fast unit cost algorithm adapted for cuts.

Figure 5.3: Comparison of how subproblems are ruled out in the permanent and temporary table of Zhang and Shasha's algorithms. White elements are computed subproblems.

mapping is greater than k. We define a relevant subtree and prune the data tree by removing all subtrees that are not relevant.

Definition 5.3 (Relevant subtree (lower bound)) Given two trees T_1 and T_2, a threshold k, and a lower bound function $D : T \times T \rightarrow \mathbb{R}$, a subtree $T_1(v)$ is relevant if there is a subtree $T_2(w)$ for which $D\big(T_1(v), T_2(w)\big) \leq k$.

Definition 5.3 suggests a worst case $O\big(|T_1||T_2| \cdot \sigma(T_1, T_2)\big)$ time algorithm, where σ denotes the running time of the lower bound function D. If the execution time of the pruning approach exceeds that of the tree edit distance algorithm, its purpose is defeated. Therefore, we require that $\sigma(T_1, T_2) \in O(|T_1||T_2|)$, based on the theoretical worst case of Zhang and Shasha's algorithm.

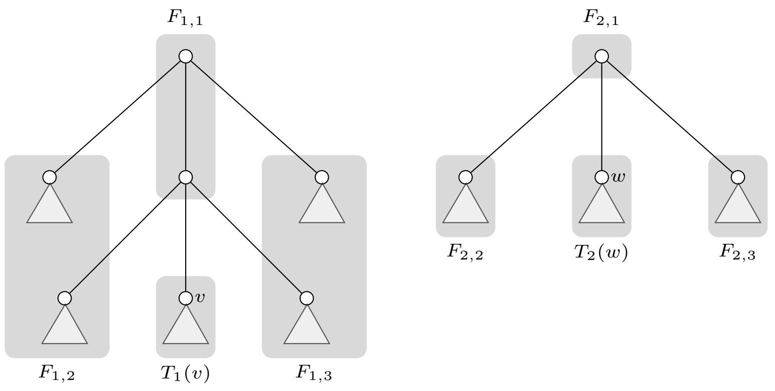
A lower bound $D(T_1(v), T_2(w))$ for two subtrees $T_1(v)$ and $T_2(w)$ may not be very tight, so we draw inspiration from the unit cost algorithm on how to improve it. Recall that the difference in the size of the subforests to the left of the subtrees also could be added to the lower bound. If we preprocess the trees such that the depth and the leftmost leaf of a node can be found in constant time, we can divide the tree into four subforests which all add to the lower bound. The four subforests are shown in figure 5.4 on the following page

The difference in the size of the subforests can be added to the lower bound $D(T_1(v), T_2(w))$ because all subforests are disjoint, so the improved lower bound D_{imp} is

$$\begin{aligned} D_{imp}\big(T_1(v), T_2(w)\big) = & D\big(T_1(v), T_2(w)\big) \\ & + \big||F_{1,1}| - |F_{2,1}|\big| \\ & + \max\big(0, |F_{2,2}| - |F_{1,2}|\big) \\ & + \max\big(0, |F_{2,3}| - |F_{1,3}|\big) \end{aligned} \tag{5.4}$$

Assume that the trees have been preprocessed such that *lml* and *depth* queries take constant time. The first term is computed using any of the methods from chapter 4. The second term is the difference in the length of the path from the roots to v and w. We assume that $T_1(v)$ and $T_2(w)$ are part of the optimal mapping, so the cut operation does not apply to $F_{1,1}$. There are no leaves to cut. Therefore, this term can be computed from $|depth(v) - depth(w)|$ in constant time. The size of the left subforest of v is the index of $lml(v)$. The size of the right forest of v is $root(T_1) - (v - lml(v)) - lml(v) - (depth(v) - 1)$. So term 3 and 4 of (5.4) can also be found in constant time.

This pruning technique excels in being independent of the context. It will have an effect on any data. It can also be extended to use other methods for com-



$T_1(v)$ and $T_2(w)$: the subtrees rooted at v and w,

$F_{1,1}$ and $F_{2,1}$: the path from the root of the trees to v and w, respectively,

$F_{1,2}$ and $F_{1,2}$: the subforests to the left if v and w,

$F_{1,3}$ and $F_{1,3}$: the subforests to the right of v and w.

Figure 5.4: Division into subforests for computing improved lower bound when pruning.

puting the lower bounds for the subforests. We have chosen to use the size of the forests because it can be found in constant time.

We anticipate that the difference in the length of the paths from the roots to v and w is an efficient contributor to the lower bound. In a pattern matching context, the effect will be significant if the pattern is intended to match a shallow subtree. Then deep subtrees will be removed from the data. On the contrary, the effect from comparing the right or left subforests is small in a pattern matching context because the subforests of T_2 are smaller than those of T_1 in most cases.

5.3 Using the HTML Grammar for Pruning

Nesting of HTML elements must comply with the Document Type Definition (DTD). Given the DTD we can use the same procedure as for lower bound pruning to remove subtrees which are likely not to be a part of an optimal mapping. Consider the following definition of a relevant subtree. We remove all subtrees that are not relevant.

Definition 5.4 (Relevant subtree (HTML grammar)) Given two trees T_1 and T_2, a subtree $T_1(v)$ is relevant if there is a node w in T_2 such that the HTML element represented by w can be derived from the element represented by v from the DTD by applying one production rule or there is a relevant subtree $T_1(u)$ where u is a descendant of v.

The definition is recursive which is utilized by the algorithm. It traverses the nodes of the data tree in postorder. For each node it checks if there is a child that is the root of a relevant subtree. If so, the current subtree is also relevant. If not it tests if there is a tag in T_2 that is a possible descendant of the current node's tag. To do this it is allowed to apply one rule from the DTD.

The traversal of the children of each node entails that each node is visited at most two times. For each node in T_1 each node in T_2 is visited once. Finally, at most $|T_1|$ nodes are deleted. Thus, the running time of the algorithm is $2|T_1| + |T_1||T_2| \in O\big(|T_1||T_2|\big)$.

We now discuss the expected efficiency of HTML pruning. If the provided pattern contains tags in the `body` branch of the HTML document, this approach will remove at least the `head` branch, and vice versa. Also, inline elements can not contain block elements, and most interactive elements can not contain elements from any other group. For instance, all inline elements will be pruned if the pattern is sheer block elements. Another example: the `select` element can only contain `optgroup` and `option` elements. So if the pattern is composed of `div` and `span` tags, all `select`, and potentially many `option` tags, are removed.

Unfortunately, the restrictions posed by the DTD on basic layout elements are not very strict. For instance, given a pattern of several `div` containers, it is unlikely that this is within a table cell, and it would have been nice to be able to remove all tables from the data. Alas, table cells can contain `div` containers so this is not possible. If a transitional DTD is used, the restrictions become even more vague, and this may also influence the efficiency of the approach.

In special cases, where there is no obvious subtree to match the pattern, this pruning approach may remove subtrees that are part of the optimal solution. In such cases the optimal solution may be to relabel many nodes to fit the pattern. The relabeled nodes may semantically be very different but be arranged similar to the pattern. Because of the semantic difference they are prone to being removed by HTML pruning.

5.4 Pre-selection of Subtrees

We have proposed two techniques for pruning the data tree, but in some cases the effect from either of these may not be substantial. In this section we propose some heuristics for selecting subtrees of the data tree based on the root of the pattern. When using XPaths for web scraping it is common to use the `id` or `class` attribute (or any suitable attribute or CSS property) to select subtrees where the target data may possibly be. This is based on the assumption that it is less likely that for instance the `id` of a tag changes than the path from the root to the tag changes. We adopt this approach to approximate tree pattern matching.

We have identified three cases where the root of the pattern will map to a node in the data tree with zero cost, because a mapping to any other node will conflict with the semantic meaning of the HTML tags.

Case 1. If the root of the pattern is a `body` or `head` tag it is unlikely that it will map to a node with another tag because the three mentioned tags are unique and only appear once each in the HTML document.

Case 2. If the root of the pattern has the `id`-attribute, it is unlikely that it will map to a node representing tag with another `id`. Since the `id` uniquely identifies a tag it is unlikely that it will change due to a modification of a website layout. This of course depends on the extent of the modification. For example, it is not unlikely that a tag is removed when the layout is modified, so this only applies to tags that can be identified as playing a key role in the layout of the website.

Case 3. If the root of the pattern is a tag with a restricted set of possible descendants, it is unlikely that it will map to a node representing another tag. Tags such as `form` and `table` is expected to contain certain tags, so renaming either of these requires most of its descending tags to be renamed as well.

The three cases can be used to select subtrees by doing a linear search for nodes that match the tag (case 1 and 3) or the `id` (case 2) of the root of the pattern. Because they are based on assumptions, using them for pre-selection of subtrees can result in sub optimal solutions.

Case 1 is the strongest assumption, but in practice it will at most eliminate the `head` or `body` branch of the data tree. Case 2 requires the user to identify an element with a key role in the layout. A web site often consists of a number of `div` containers given `ids` and styled using a Cascading Style Sheet (CSS). For the average user it can be difficult to determine how likely it is that for instance a `div` container is discarded in a layout modification. Therefore, this case should only be used as a last resort. Case 3 only applies to a small set of patterns but when used it is subject to large reductions of the data tree.

5.5 Linear Matching

As we have seen in section 3.2 on pattern design, patterns naturally end up having an anchor of some length. Likewise, we have identified that many websites use a number of nested, styled `div` containers in order to create their visual appearance. The resulting HTML tree therefore contains paths among its internal nodes where each node has exactly one child. If we apply pre-selection and the result is a subtree with an anchor, there is a chance that the anchors of the selected subtree and the pattern will be part of a mapping with zero cost. We can exploit this by mapping nodes in a top-down manner for as long as the cost of renaming nodes is zero.

Lemma 5.5 formalizes that a mapping obtained from linear matching is part of an optimal mapping.

Lemma 5.5 (Linear matching) *Let T_1 and T_2 be trees where nodes are given preorder indices. The lenght of the anchor of T_1 is a_1 and the length of the anchor of T_2 is a_2. Let γ be a unit cost function on the nodes, and let M be an optimal mapping from T_1 to T_2. Let $v_i \in V(T_1)$ and $w_i \in V(T_2)$, then we have*

$$\big\{(v_i, w_i),\, 0 \leq i \leq \min(a_1, a_2) \wedge \gamma(v_i, w_i) = 0 \wedge \big(i = 0 \vee (v_{i-1}, w_{i-1}) \in M\big)\big\} \subseteq M$$

Proof. Let $v = root(T_1)$ and $w = root(T_2)$. It is sufficient to show that (v, w) is part of an optimal mapping if $\gamma(v, w) = 0$, because we know from lemma 2.3 on page 10 that the algorithm will recurse on $T_1 - v$ and $T_2 - w$, and the proof then holds recursively. Assume that the cost of relabeling v to w is 0, but there is a mapping M' with cost c such that w maps to some other node v' that descends from v. Let d be the depth of v', then $0 < d \leq c$. We can transform M' to a mapping M'' that contains (v, w) by removing (v', w) and inserting (v, w). To change the cost of the mapping accordingly, we need to subtract the depth of v' and the cost of deleting it. Thus, the cost of M'' is $c - (d - 1)$. Since we know that $d > 0$ we see that M'', which contains the pair (v, w), is just as good or better than M'. $\square$

The advantage of this heuristic is that it is very fast. If no nodes match, only the roots of the data and the pattern is compared, so the overhead is constant. If one or more nodes match, the overhead is $O(height(T_2))$, but in return we get a reduction in both the data and the pattern before invoking the approximate tree pattern matching algorithm.

Chapter 6

Experiments

This chapter presents the results from conducting a range of experiments using our implementation (described in appendix A).

6.1 Setup

The objective of the experiments is to

- compare execution time of Zhang and Shasha's algorithm to our adaption of their fast unit cost algorithm for cuts when used for web scraping,
- compare the lower bound methods,
- compare HTML grammar pruning and lower bound pruning using all lower bound methods,
- substantiate the effect of the heuristic methods, and
- show how tolerant Zhang and Shasha's algorithm is to changes in a web page and compare it to XPath.

This results in five independent experiments.

Since the main purpose of the thesis is to apply approximate tree matching to web scraping, we want to show how the algorithms perform in scenarios where the data is HTML and the pattern is much smaller than the data tree. For the comparison of the pruning methods and the heuristic methods it is evident that we use HTML data because some methods are designed only to work on HTML. For all experiments except the data extraction experiment we use the test cases described in table 6.1.

Case	URL	Purpose	$\|T_D\|$	$\|T_P\|$	δ_c
1	`http://berlingske.dk/`	Extract standings in the danish football league.	3371	58	2
2			3371	48	11
3	`http://version2.dk/`	Extract headline of the most recent article.	728	24	2
4			728	16	10
5	`http://reddit.com/`	Extract headline of most recent entry.	1949	17	4
6			1949	15	7

Table 6.1: Test cases used for experiments. The patterns used for the test cases are enclosed as appendix C on page 85. 2011-08-31.

We refer to Zhang and Shasha's approximate tree pattern matching algorithm as ZhangShashaATM, and to our adaption of the fast unit cost algorithm for cuts as FastUnitCostATM.

All execution times are measured using the built-in Python function `clock()` which gives a good estimate of how much CPU time a process has used [26]. A new process is spawned for each run of a test case to even out variation due to cache misses and garbage collection.

All tests are executed on a Dell E4300 laptop with an Intel Core 2 Duo (2.26 GHz) and 4 GB of RAM running Ubuntu Linux 9.10 and Python 2.6.4.

6.2 Algorithm Execution Time

The results from running ZhangShashaATM is shown in table 6.2 on the next page. We see that for case 1 and 2, i.e. for large data and pattern trees, ZhangShashaATM is slow. For the other cases, it performs reasonable.

FastUnitCostATM has been run with increasing values of k starting from the

Case	ZhangShashaATM
1	67.99 s
2	34.37 s
3	1.54 s
4	1.21 s
5	4.39 s
6	4.57 s

Table 6.2: Execution times of Zhang and Shasha's approximate tree pattern matching algorithm.

optimal tree edit distance. The results are shown in table 6.3 on the following page. As anticipated, the algorithm does not outperform ZhangShashaATM.

6.3 Lower Bound Methods

The lower bound methods have been tested on the 6 test cases and the results are shown in table 6.4 on the next page. We see that the only test where the approximation is close to the optimal edit distance with cuts δ_c is when we use the Euler string distance in case 3. This indicates that the methods are unfit for use as an approximation of the cost of matching a HTML pattern to a HTML tree.

The lower bound methods were developed with the application to pruning in mind. In our pruning algorithm the methods are applied to all pairs of subtrees of two trees T_1 and T_2. Therefore, we have run the methods on all pairs of subtrees in the 6 cases and accumulated the distances. The results are compared to the accumulated tree edit distance $\sum \delta_c = \sum_{\forall(v,w)\in V(T_1)\times V(T_2)} \delta_c(v,w)$ between all pairs of subtrees. Based on the results shown in table 6.5 on page 53, the q-gram tree distance, where $q = 1$, and the Euler string distance are the tightest lower bound approximations.

6.4 Pruning Methods

We want to determine how effective the pruning methods are on HTML data. Our approach is to compare the size of the data tree before and after a pruning method has been applied and measure the CPU time of executing the pruning

k	Execution time
2	304.92 s
3	314.85 s
4	319.72 s
5	322.58 s
6	327.38 s
7	330.02 s

(a) Case 1.

k	Execution time
11	107.31 s
12	106.95 s
13	108.02 s
14	119.34 s
15	118.45 s
16	110.28 s

(b) Case 2.

k	Execution time
2	18.41 s
3	19.17 s
4	19.13 s
5	19.73 s
6	20.13 s
7	20.63 s

(c) Case 3.

k	Execution time
10	9.52 s
11	9.47 s
12	9.91 s
13	10.02 s
14	10.28 s
15	10.30 s

(d) Case 4.

k	Execution time
4	14.47 s
5	14.74 s
6	15.22 s
7	15.86 s
8	15.91 s
9	16.44 s

(e) Case 5.

k	Execution time
7	10.35 s
8	10.56 s
9	10.70 s
10	10.88 s
11	11.21 s
12	11.41 s

(f) Case 6.

Table 6.3: Shasha and Zhang's fast unit cost algorithm adapted for cuts on all test cases for increasing values of k.

Case	δ_c	Size	Height	Q-gram				Euler string
				$q = 1$	$q = 2$	$q = 4$	$q = 8$	
1	2	0	0	0	0	0	0	0
2	11	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	1
4	10	0	0	0	0	0	0	0
5	4	0	0	0	0	0	1	1
6	7	0	0	0	0	0	0	2

Table 6.4: Results for lower bounds test.

Case	$\sum \delta_c$	Size	Height	Q-gram $q = 1$	Q-gram $q = 2$	Q-gram $q = 4$	Q-gram $q = 8$	Euler string
1	2288	1904	402	2207	1195	679	627	2235
2	657	409	147	603	367	229	224	611
3	197	151	88	179	100	63	52	192
4	74	49	22	64	38	26	23	70
5	204	158	67	183	118	86	80	201
6	120	83	31	105	73	57	57	116

Table 6.5: Results for alternative lower bounds test. The distances are the sum of the distance between each pair of subtrees. All numbers are in thousands.

method. We also execute ZhangShashaATM together with the pruning methods to determine if the speed-up from pruning the data tree compensates for the execution time of the pruning method. When using the lower bound methods k is set to the optimal tree edit distance with cuts plus 2.

Based on the results from running pruning methods exclusively, we combine the best pruning methods to see how big the overlap of removed nodes is. Finally, we choose the three best combinations of pruning methods and execute them with increasing values of k.

In table 6.6 on page 55 we see the results from using the pruning methods on all cases. The following summarizes the findings.

- In case 1 the execution time is almost reduced by half when using q-gram and $q = 1$. The lower bound method reduces the data tree by 45.24 %. In all other cases there is little or no reduction in the execution time. In case 3, large amounts of the data tree is removed, but the pruning methods are too slow.

- The Euler string lower bound method is the most effective. In case 3 it reduces the data tree by 94.2 %. It is also the slowest.

- As anticipated, the HTML grammar pruning is the only algorithm that always reduces the data tree. This is because it removes the `head` branch of the HTML tree in all cases.

- In case 4, 5, and 6, lower bound pruning has no effect. This is possibly because the data tree mostly consists of branches that resembles the pattern. The pruning methods do have effect if the pattern contains nodes that are not used very frequently in the data as in case 1, or the pattern contains many different nodes as in case 3.

Based on these findings we have combined the size, height, q-gram ($q = 1$), and HTML pruning methods for a similar round of tests. However, these tests have only been run on case 1, 2, and 3 where the pruning methods proved to have an effect when run exclusively. The results are shown in table 6.7 on page 56, and are summarized below.

- There is an overlap in the data removed by the pruning methods. However, case 3 shows that the overlap between the size and q-gram methods is so small that the combination of the two is fast.
- Combining the pruning methods generally result in an increase in the amount of removed data, but it is too small to compensate for the overhead of using two pruning methods.

The effectiveness of the lower bound pruning methods depends on the threshold k. We have selected 1-gram, size+height, and size+1-gram as the three best pruning methods, and have executed them on case 1 (figure 6.1 on page 57) and case 3 (figure 6.2 on page 57) for increasing values of k. In both cases the optimal tree edit distance with cuts is 2.

Figure 6.1 shows that for small values of k, the size+1-gram combination is marginally better than using just 1-grams. Most notable is the increase in execution time when k is changed from 6 to 7. For values greater than 7 the pruning methods are useless. The same tendency is seen in figure 6.2, but the break even is between $k = 4$ and $k = 5$. The latter figure also shows that if the k-value provided is not sufficiently tight, then the pruning methods may add significant overhead. In practice we should only use the pruning methods if we can provide a tight k-value.

6.5 Heuristics

The results from using pre-selection and linear matching on the 6 test cases are shown in table 6.8 on page 58. We see that pre-selection selects a subtree to work on in case 2, 3, and 6. The patterns used in these cases have a `table`, `body`, and a `div` tag with an `id` attribute, respectively, as roots. Linear matching has no influence on the test cases. The execution time of the heuristic methods is omitted from the table because it is smaller than 0.01 seconds for all cases.

Case	None	Lower bound						HTML
		Size	Height	q-gram ($q = 1$)	q-gram ($q = 2$)	q-gram ($q = 4$)	Euler string	
	-	5.68 %	1.37 %	45.24 %	45.24 %	45.24 %	45.24 %	0.9 %
1	-	0.05 s	0.79 s	5.09 s	8.65 s	14.4 s	33.66 s	0.04 s
	62.06 s	58.25 s	61.95 s	**35.46 s**	39.68 s	46.54 s	63.92 s	62.03 s
	-	0.31 %	0.31 %	0.31 %	0.31 %	0.31 %	0.31 %	0.91 %
2	-	0.04 s	0.33 s	2.46 s	4.09 s	6.83 s	7.45 s	0.04 s
	31.76 s	31.84 s	31.69 s	34.37 s	36.76 s	40.47 s	39.59 s	**31.23 s**
	-	10.64 %	7.87 %	51.24 %	75.0 %	12.98 %	94.2 %	2.62 %
3	-	0.01 s	0.11 s	0.74 s	1.24 s	2.22 s	5.37 s	< 0.01 s
	1.45 s	1.37 s	1.50 s	**1.36 s**	1.53 s	3.45 s	5.44 s	1.45 s
	-	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	2.62 %
4	-	< 0.01 s	0.03 s	0.13 s	0.23 s	0.39 s	0.32 s	< 0.01 s
	1.22 s	1.22 s	1.16 s	1.26 s	1.37 s	1.54 s	1.43 s	**1.14 s**
	-	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	3.42 %
5	-	0.02 s	0.08 s	0.38 s	0.61 s	0.92 s	1.07 s	0.01 s
	4.42 s	4.41 s	4.51 s	5.15 s	5.16 s	5.42 s	5.56 s	**4.36 s**
	-	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	3.42 %
6	-	0.02 s	0.08 s	0.31 s	0.51 s	0.73 s	0.79 s	0.01 s
	4.53 s	4.55 s	4.66 s	4.91 s	5.20 s	5.49 s	5.42 s	**4.50 s**

Table 6.6: Results from pruning method tests. First row of each case is percentage of data tree removed. Second row is the execution time of the pruning method. Third row is the execution time of the pruning method followed by ZhangShashaATM.

Case	Best	Size+Height	Size+1-gram	Size+HTML	Height+1-gram	Height+HTML	1-gram+HTML
	45.24 %	5.71 %	45.24 %	6.06 %	45.24 %	1.72 %	45.58 %
1	5.09 s	0.81 s	4.89 s	0.08 s	5.98 s	0.83 s	5.12 s
	35.46 s	59.64 s	35.98 s	58.89 s	37.95 s	62.31 s	36.31 s
	0.91 %	0.31 %	0.31 %	1.22 %	0.31 %	1.22 %	1.22 %
2	0.04 s	0.37 s	2.5 s	0.07 s	2.82 s	0.37 s	2.5 s
	31.23 s	32.01 s	34.76 s	31.51 s	35.02 s	31.24 s	34.28 s
	57.24 %	13.89 %	52.13 %	13.2 %	52.13 %	7.84 %	51.31 %
3	0.74 s	0.10 s	0.62 s	0.02 s	0.76 s	0.11 s	0.71 s
	1.36 s	1.41 s	**1.30 s**	1.31 s	1.42 s	1.50 s	1.39 s

Table 6.7: Results from combining two pruning methods. First row of each case is percentage of data tree removed. Second row is the execution time of the pruning method. Third row is the execution time of the pruning method followed by ZhangShashaATM.

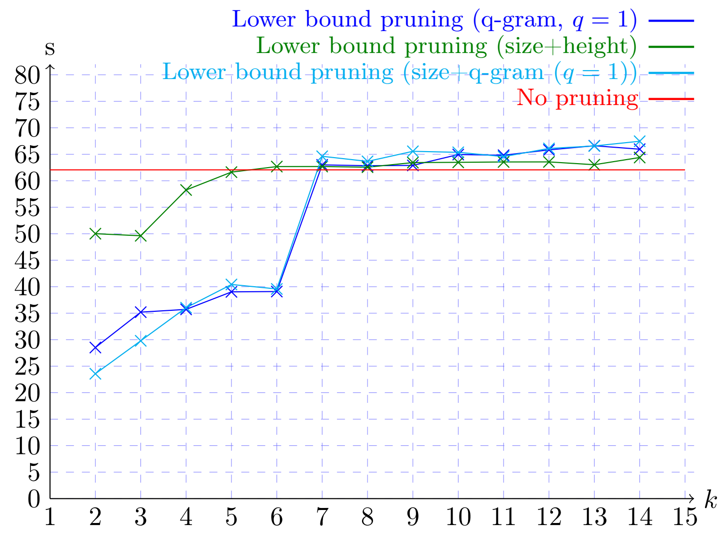


Figure 6.1: Execution time of lower bound pruning methods followed by ZhangShashaATM on case 1 for increasing values of k.

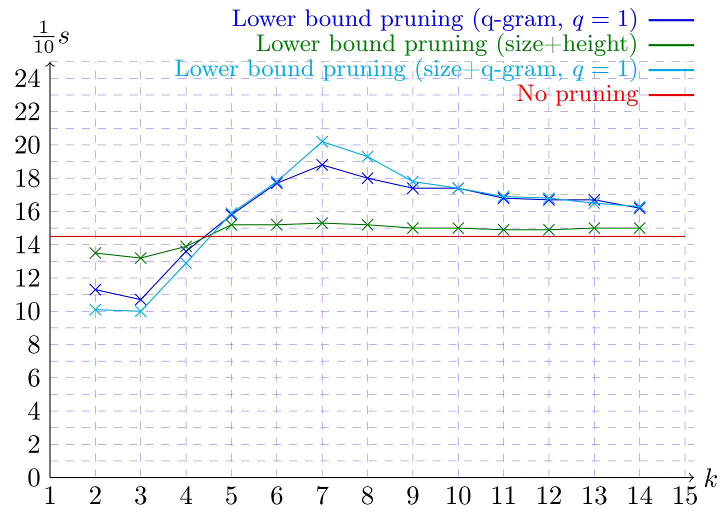


Figure 6.2: Execution time of lower bound pruning methods followed by ZhangShashaATM on case 3 for increasing values of k.

Case	None	Pre-selection	Pre-selection and Linear Matching
1	-	0.0 %	0.0 %
	62.97 s	63.19 s	63.45 s
2	-	99.75 %	99.75 %
	32.09 s	0.25 s	0.25 s
3	-	3.24 %	3.24 %
	1.3 s	1.05 s	1.04 s
4	-	0.0 %	0.0 %
	0.99 s	0.98 s	1.0 s
5	-	0.0 %	0.0 %
	4.36 s	4.33 s	4.38 s
6	-	56.07 %	56.07 %
	4.51 s	0.99 s	0.98 s

Table 6.8: Results from using the heuristics with ZhangShashaATM. The first row for each case is the percentage of the data tree removed and the second row is the execution time of ZhangShashaATM plus the overhead from using the heuristic.

6.6 Data Extraction

Zhang and Shasha's algorithm extracts the correct data from the 6 test cases in table 6.1 on page 50, so to compare the behaviour of the algorithms we have constructed the simple web page shown in figure 6.3 on the next page. The web page consists of 5 `div` containers which are given different `class` attributes. The two innermost `div`s contain a bold text in a paragraph and three bullet points. We make 8 changes to the web page. The changes cover some possible changes in markup that a website can be subject to.

The target of the experiment is to extract the text in the green `div`. The orange `div` is present to mislead the algorithms. We compare our algorithms to the result from applying the following two different XPaths to the web page.

XPath pattern 1	XPath pattern 2
`//div/div[1]/div/p/strong |`	`//div[@class='green']/p/strong |`
`//div/div[1]/div/ul/li`	`//div[@class='green']/ul/li`

We also include Zhang's algorithm for the isolated-subtree mapping in the comparison. Although it was rejected for not being suited for web scraping in chapter 3 we want to take this opportunity to investigate if there are more cases where it fails to locate the correct data. The pattern used with the approximate

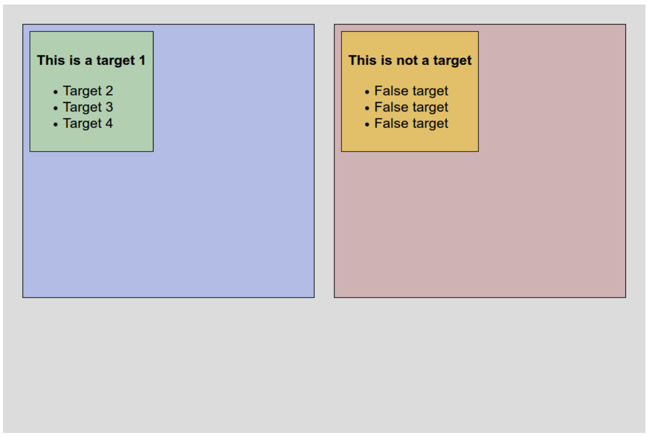


Figure 6.3: Initial layout of web page used for data extraction experiment.

tree matching algorithms is shown below.

```html
<div id="outer">
  <div class="blue">
    <div class="green">
      <p><strong><!-- target 1 --></strong></p>
      <ul>
        <li><!-- target 2 --></li>
        <li><!-- target 3 --></li>
        <li><!-- target 4 --></li>
      </ul>
    </div>
  </div>
</div>
```

We define the result from running an algorithm to be a four-tuple where each element is either T, if the target was found, F, if another text was found, or $-$, if no data was found. All algorithms are able to extract the target data from the web page shown in figure 6.3.

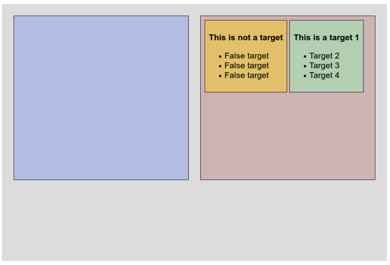
The results are shown in figure 6.4 on page 61 and figure 6.5 on page 62. Below we analyze the cases where ZhangShashaATM fails.

- In case (c) ZhangShashaATM extracts the incorrect data. Had the algorithm relabeled the red `div` to the blue `div` instead of relabeling the green `div` to the orange `div`, it would have extracted the target data. The latter case has the same cost as the first, so it is a matter of breaking ties for the algorithm. We could also argue that since the orange `div` has taken the place of the green `div`, the data we are trying to extract is in fact the contents of the orange `div`. The interpretation depends on the context. The second XPath pattern extracts the data because it targets the green `div` on its class name.

- In case (f) the paragraph and the unordered list have switched place. We know that ZhangShashaATM can not match both because it would violate the definition of a mapping. Therefore, it chooses to remove the smallest subtree of the two. Again, we could also argue that the original target 1 has been removed and a new text has been inserted, in which case the result from the algorithm is correct. Both XPath patterns extract the data because they are a concatenation of two paths.

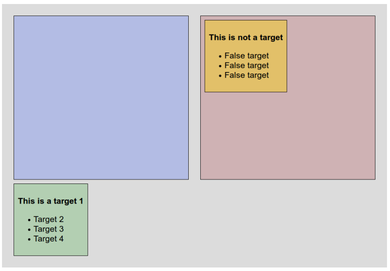
- In case (g) the incorrect data is extracted. The change is radical and one might argue that in such cases the algorithm is not supposed to find a matching. The XPaths also fails.

The results show that IsolatedSubtreeATM locates the data in the same cases as ZhangShashaATM.



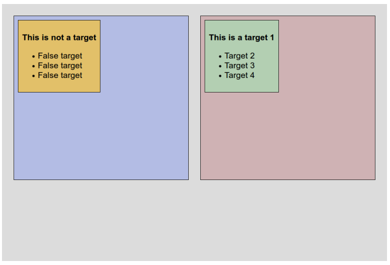
Algorithm	Result
XPath 1	$(-,-,-,-)$
XPath 2	(T,T,T,T)
ZhangShashaATM	(T,T,T,T)
IsolatedSubtreeATM	(T,T,T,T)

(a)



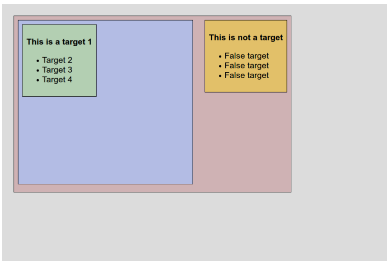
Algorithm	Result
XPath 1	$(-,-,-,-)$
XPath 2	(T,T,T,T)
ZhangShashaATM	(T,T,T,T)
IsolatedSubtreeATM	(T,T,T,T)

(b)



Algorithm	Result
XPath 1	(F,F,F,F)
XPath 2	(T,T,T,T)
ZhangShashaATM	(F,F,F,F)
IsolatedSubtreeATM	(F,F,F,F)

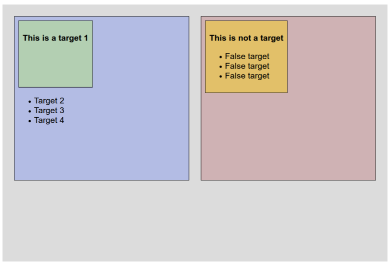
(c)



Algorithm	Result
XPath 1	(F,F,F,F)
XPath 2	(T,T,T,T)
ZhangShashaATM	(T,T,T,T)
IsolatedSubtreeATM	(T,T,T,T)

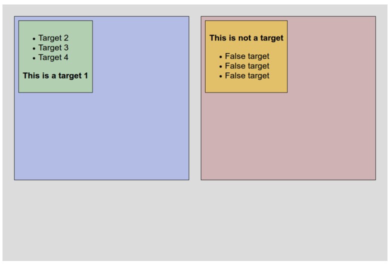
(d)

Figure 6.4: Results from data extraction experiment (part 1).



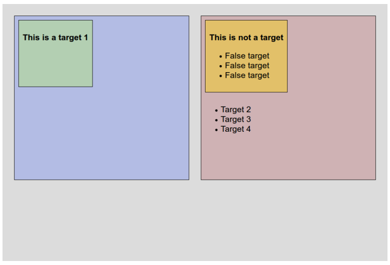
Algorithm	Result
XPath 1	$(T, -, -, -)$
XPath 2	$(T, -, -, -)$
ZhangShashaATM	(T, T, T, T)
IsolatedSubtreeATM	(T, T, T, T)

(e)



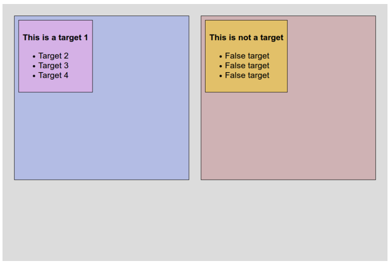
Algorithm	Result
XPath 1	(T, T, T, T)
XPath 2	(T, T, T, T)
ZhangShashaATM	(F, F, F, F)
IsolatedSubtreeATM	(F, F, F, F)

(f)



Algorithm	Result
XPath 1	$(T, -, -, -)$
XPath 2	$(T, -, -, -)$
ZhangShashaATM	(F, F, F, F)
IsolatedSubtreeATM	(F, F, F, F)

(g)



Algorithm	Result
XPath 1	(T, T, T, T)
XPath 2	$(-, -, -, -)$
ZhangShashaATM	(T, T, T, T)
IsolatedSubtreeATM	(T, T, T, T)

(h)

Figure 6.5: Results from data extraction experiment (part 2).

Chapter 7

Discussion

In this chapter we will consider our findings as a whole and discuss how they are best combined to a solution for web scraping. The discussion will focus on choosing an algorithm and suitable pruning methods, and the consequences the choices will have.

7.1 Algorithms

In the litterature, web scraping using approximate tree pattern matching is achieved using algorithms that create a top-down mapping, because the focus is on generating the pattern from a set of web pages. However, there are cases where the data to be extracted resides on just one web page, so the need for a more versatile solution is present.

We have considered Zhang and Shasha's algorithm and Zhang's isolated-subtree algorithm for web scraping, because unlike the top-down mapping, they impose no requirements on the pattern. Although fictitious, the Reddit example shows that there is a simple case of change in the markup where the isolated-subtree algorithm fails to locate the targeted data. We regard this example as a change that is likely to happen due to the fact that it is a simple way of adding or removing visual attributes to a collection of tags. Furthermore, the data extraction experiment shows that there are no cases where the isolated-subtree algorithm locates data that Zhang and Shasha's algorithm was unable to locate. To meet our aim that the solution should be as error tolerant as possible, the isolated-subtree algorithm is not an option for web scraping.

That being said, the data extraction experiment shows a couple of cases where Zhang and Shasha's algorithm extract incorrect data. It goes to show that the optimal solution is not always what we intuitively would call the correct solution. To deal with this, we could guide the algorithm using the cost function. However, this is not covered in this thesis because the pruning methods assume that a unit cost function is used. On the other hand, the experiment also shows that the correct data is found in 7 out of the 8 cases using either the evaluation of the second XPath or approximate tree pattern matching. This suggests a hybrid between the methods where e.g. approximate tree pattern matching is used as fallback if at least one of the XPaths yields an empty result set.

We have shown that Shasha and Zhang's fast unit cost algorithm is not suited for pattern matching when the pattern is small compared to the data tree. There may still be a speed-up compared to Zhang and Shasha's algorithm in cases where the pattern is close to the size of the data tree. To be faster, the fast unit cost algorithm must rule out about as many subproblems as ruled out when Zhang and Shasha's algorithm selects keyroots.

7.2 Lower Bound Methods

The cut operation has made it difficult to obtain tight lower bounds. Two novel approximation methods from the litterature, the PQ-gram distance and the binary branch distance, were not tranformable to produce lower bounds at all, and the size and height methods only apply if the pattern tree is bigger or higher than the data tree.

The tightest lower bounds are obtained from the Euler string distance and q-gram distance where $q = 1$. The two methods are in fact quite similar. When $q = 1$ we check if the nodes in the pattern tree are also present in the data tree. This is basically the same that happens when comparing the Euler strings because the delete operation in the string edit distance algorithm is free. However, the order of the nodes in the Euler string embeds some information about the tree structure, so the Euler string distance gives slightly better approximations. In return it is slow.

Our experiments show that if the data is HTML and the pattern is found one or more times in the data, the methods are poor approximations. However, further experiments show that if the cost of an optimal mapping of a pattern to the data is high, the approximations become better. This is useful when the methods are used in our pruning algorithm.

7.3 Pruning Method and Heuristics

We lower bound pruning algorithm is independant of the type of data the tree models. Our experiments show that when the data is HTML the combination between the size and 1-gram distance lower bound methods are the most effective as long as the threshold k is reasonably tight.

An important property of the algorithm is that branches used in an optimal solution is not removed. The drawback of the algorithm is its running time, which is $O\big(|T_1||T_2| \cdot \sigma(T_1, T_2)\big)$, where $\sigma(T_1, T_2)$ is the running time of the lower bound method used. If using the Euler string distance with the algorithm, the running time becomes $O\big(|T_1|^2|T_2|^2\big)$, which is worse than the actual running time of Zhang and Shasha's algorithm. However, from our experiments we see that the constant factor of overhead of the algorithm is little, so there are cases where pruning is beneficial and cases where the data tree is not reduced but the overhead is negligible.

The lower bound pruning algorithm becomes more effective when the pattern is large and has many nodes with distinct labels. However, if it is too large and pruning fails to reduce the data tree significantly, the whole process is slowed down by both the pruning algorithm and subsequently by the approximate tree pattern matching algorithm.

Our experiments show that the HTML grammar pruning algorithm is not very effective. However, the patterns used in the tests consist mostly of generic block and inline, so the algorithm may become more effective if more diverse patterns are used. HTML inherits a lot of its properties from XML, so the algorithm is applicable to any tree model of semi-structured data with a DTD. In other domains, the algorithm may also be more effective. It is not guaranteed not to remove a branch that is part of an optimal solution, but for sufficiently large patterns, we regard it as unlikely.

Finally, we have defined some heuristics for reducing the data tree. Pre-selection yields large reductions in the size of the data tree. To take advantage of the heuristic we must strive to select a pattern where the root is either the `body`, the `head`, a `table` or a `form` tag or has the `id` attribute. However, the size of the pattern should not be too small. If pre-selection fails, the node that is expected to map to the root of the pattern should not be too deep because this may result in incorrect data being extracted. Therefore, the pattern should have an anchor of suitable length.

Linear matching is only applicable in a limited set of cases. There may be some subtle differences in the mapping of the nodes in the anchors compared to the mapping obtained from using Zhang and Shasha's algorithm, but since we often want to extract data from leaf nodes (or deep nodes) in the pattern, this should not influence the outcome when used for web scraping.

Chapter 8

Conclusion

The aim of this thesis was to develop a fast and error tolerant solution for web scraping based on the tree edit distance without imposing any restrictions on the pattern. The litterature treats web scraping as the second phase in a two-phase procedure, where the first phase deals with learning a pattern. If the set of web pages to learn from is not sufficiently large, the web scraping phase will fail because algorithms for the top-down mapping are used. Since we often want to extract data from just one web page, the methods from the litterature are inadequate.

First, we proposed using an algorithm for the optimal mapping or an algorithm for the isolated-subtree mapping. We gave an example of a simple and likely change to the markup of a web page that made the algorithm for the isolated-subtree mapping fail. In our experiments, the result from the algorithm for the optimal mapping and the algorithm for the isolated-subtree mapping was the same. Based on the first example, the isolated-subtree mapping is not apt for web scraping. Among the algorithms for the optimal mapping, Zhang and Shasha's algorithm is the most suited because its running time depends on the height of the input trees, which generally is low for HTML trees.

We adapted Shasha and Zhang's fast unit cost tree edit distance algorithm to pattern matching, but our experiments confirmed that the fourth edit operation, cut, which is required for pattern matching, makes it difficult to rule out a significant number of subproblems. However, the work on the algorithm served as inspiration for the two pruning algorithms that were developed.

The lower bound pruning algorithm extends the technique from the fast unit cost algorithm to prune the data tree prior to running an approximate tree pattern matching algorithm. In particular, it uses one of the lower bound methods that were developed from novel tree edit distance approximation methods found in the litterature. Transforming these methods to produce tight lower bound approximations for the tree edit distance with cuts proved to be difficult because of the issue of distinguishing between the free cut operation and the non-free delete operation when comparing parts of the trees. The second algorithm uses the HTML grammar from the DTD to exclude branches of the data tree. Of the two algorithms, the lower bound pruning algorithm is the most effective, given that the 1-gram distance is used as lower bound method. It is also the slowest because the lower bound methods are slow.

Our experiments showed that in one case, the data tree could be reduced by 95 % using the lower bound pruning algorithm with the Euler string distance lower bound method. In another case, the execution time was reduced from 62 seconds to 35 seconds due to pruning using the lower bound pruning algorithm and the 1-gram distance lower bound method. In both cases the algorithm tolerated two more errors than the cost of the optimal mapping. Furthermore, our experiments showed that the algorithm needs to be provided with a relatively small value of k in order to be beneficial.

Algorithms, pruning algorithms, and lower bound methods were implemented in Python such that they could be combined arbitrarily for the experiments.

8.1 Further Work

The data extraction experiment showed that the there are cases where the approximate tree pattern matching algorithm extracts incorrect data. To avoid this and thus cover more cases, a final solution requires more work.

We only briefly discussed how to utilize this technique for web scraping when we want to extract several matches of a pattern. Since this is a common use case in web scraping, it will require some further work to determine how to achieve this effectively.

We also believe that there are faster and better methods for finding lower bounds for the tree edit distance with cuts that are waiting to be uncovered.

Bibliography

[1] Tatsuya Akutsu. A relation between edit distance for ordered trees and edit distance for Euler strings. *Information Processing Letters*, Vol. 100, Issue 3, 105–109, 2006.

[2] Nikolaus Augsten, Michael Böhlen, Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, Vol. 35, ACM, New York, 24:1–4:36, 2010.

[3] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci*, Vol. 337, 217–239, 2005.

[4] Sudarshan S. Chawathe. Comparing Hierarchical Data in External Memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc, 90–101, 1999.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill 2001. ISBN 0-262-53196-8.

[6] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An Optimal Decomposition Algorithm for Tree Edit Distance. *ACM Transactions on Algorithms*, Vol. 6, No. 1, Article 2, 2009.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns – Element of Reusable Object-Oriented Software. Addison-Wesley 1995. ISBN 0-201-63361-2.

[8] Yeonjung Kim, Jeahyun Park, Teahwan Kim, Joongmin Choi. Web Information Extraction by HTML Tree Edit Distance Matching. In *Proceedings*

of the 2007 International Conference on Convergence Information Technology, pages 2455–2460, IEEE Computer Society, Washington, DC, USA, 2007.

[9] P.N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA)* 1998., pages 91–102. Springer-Verlag, 1998.

[10] Donald Erwin Knuth. The Art of Computer Programming: Volume 1, Fundamental Algorithms (section 2.3.2, Binary Tree Representation of Trees). Addison-Wesley, 1997. ISBN 0-201-89683-4.

[11] Davi De Castro Reis, Reis Paulo, Alberto H. F. Laender, Paulo B. Goglher, Altrigran S. da Silva. Automatic Web News Extraction Using Tree Edit Distance. In *Proceedings of World Wide Web Conference (WWW04)*, New York, USA, 2004.

[12] Thorsten Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in Lecture Notes of Computer Science (LNCS)*, Vol. 1264, Springer, 150–166, 1997.

[13] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

[14] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11:581–621, 1990.

[15] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26:422–433, 1979.

[16] Rui Yang, Panos Kalnis, Anthony K. H. Tung. Similarity evaluation on tree-structured data. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM, Baltimore, Maryland, 754–765, 2005.

[17] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science* 1, 191–211, 1994.

[18] Gabriel Valiente. An efficient bottom-up distance between trees. In *Proceedings of the 8th International Symposium of String Processing and Information Retrieval*, Press, 212–219, 2001.

[19] Kaizhong Zhang, Dennis Shasha, Jason T. L. Wang. Approximate Tree Matching in the Presence of Variable Length Don't Cares. *Journal of Algorithms*, 16:33–66, 1993.

[20] Lin Xu, Curtis Dyreson. Approximate retrieval of XML data with ApproX-Path. In *Proceedings of the nineteenth conference on Australasian database - Volume 75*, pages 85–96, ADC '08, Gold Coast, Australia, 2007.

[21] Wuu Yang. Identifying Syntactic Differences Between Two Programs. *Software – Practice and Experience*, 21:739–755, 1991.

[22] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[23] Kaizhong Zhang. Algorithms for the constrained editing problem between ordered labeled trees and related problems. *Pattern Recognition*, 28:463–474, 1995.

Web sites

[24] Algorithm Implementation/Strings/Levenshtein distance – Wikibooks, open books for an open world.
`http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance#Python`.
2011-07-24.

[25] HTML 4.01 Transitional Document Type Definition.
`http://www.w3.org/TR/html401/sgml/loosedtd.html`.
2011-07-26.

[26] Time access and conversions – Python documentation.
`http://docs.python.org/library/time.html#time.clock`.
2011-08-31.

[27] Wikipedia. Metric (mathematics).
`http://en.wikipedia.org/wiki/Metric_(mathematics)`.
2011-08-09.

[28] XPath Syntax.
`http://www.w3schools.com/Xpath/xpath_syntax.asp`.
2011-08-12.

APPENDIX A

Implementation

In this chapter we give a brief overview of the implementation with emphasis on the overall design and the implementation of the approximate tree pattern matching algorithms.

A.1 Design

We have implemented a number of algorithms, the lower bound and pruning methods, and the heuristics described in this report. The implementation is in Python (version 2.6.4) which enables easy integration with for instance web services written in Python. The result is a Python package with a fairly simple API.

The functional requirements to the package is that it should be possible to

- combine pruning methods, linear matching, and pre-selection arbitrarily, and
- easily use it with other applications.

To achieve the first goal, the pruning methods, linear matching, and pre-selection has been implemented as decorators for the algorithms*. Any number of decorators can be wrapped around the algorithm in arbitrary order.

Using decorators introduce an overhead besides the extra functionality. The input trees must be preprocessed for each decorator because the trees may have been manipulated by a previous decorator. Particularly when adding several LowerBoundPruning decorators the superflous tree preprocessing is done.

To make the algorithms usable for many purposes they operate on a generic datastructure. To use the algorithm for a specific purpose such as web scraping, the calling application either has to be designed to use the generic datastructure, or define a context for the algorithm†, which must create a mapping between the generic datastructure and the one used by the calling application. The latter option introduces some extra computation from creating the mapping between the datastructures. However, we opt for the context approach because we want to use the lxml‡ library for parsing HTML.

Algorithms, decorators, lower bound methods, and cost functions are subclasses of base classes that define the minimum requirement for functions in order to be used with the rest of the package§.

A class diagram is shown in figure A.1 on the facing page. For the sake of clarity, private attributes and empty constructors for non-abstract classes are omitted. If no constructor is specified, it is inherited from its superclass.

A.2 Modules

The implementation consists of five Python modules which will be described in this section.

*The decorator pattern [7, pp. 175] allows functionality to be added transparently to a core function which in this case is the algorithm.

†The purpose of the context pattern [7, pp. 317] is to be able reuse an implementation of an algorithm in other applications without having to redeclare types, etc.

‡lxml is Python library for parsing and processing XML and HTML. `http://lxml.de/`.

§This is sometimes referred to as the strategy pattern, but because Python is typeless, abstract classes are considered superflous. We opt to use it anyway for the sake of clarity for future contributors.

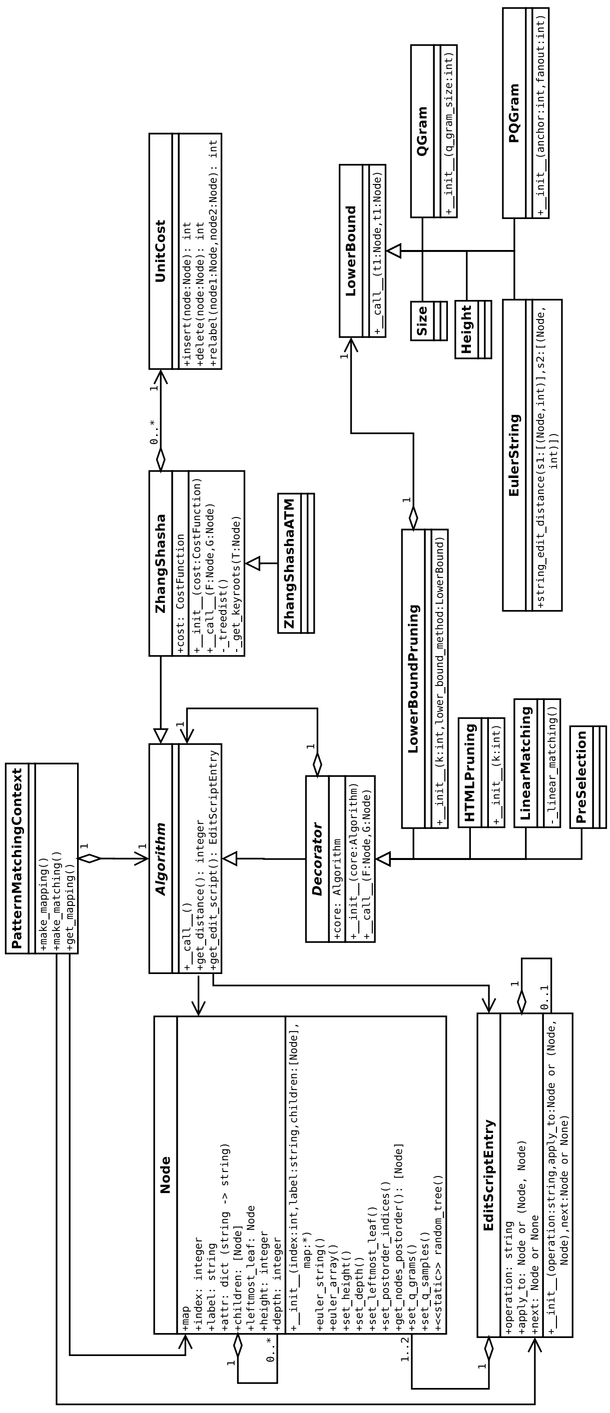


Figure A.1: Class diagram.

A.2.1 `algorithms`

This module contains implementations of the algorithms and the unit cost function. The Algorithm class defines the interface used by all algorithms. An algorithm is executed by invoking its object. Subsequently, the results can be retrieved from the functions `get_distance()` and `get_edit_script()`. An overview of the implemented algorithms is given in table A.1.

Class	Algorithm
ZhangShasha	Zhang and Shasha's algorithm [22]
ZhangShashaATM	Zhang and Shasha's algorithm modified for cuts [22]
ZhangShashaUC	Shasha and Zhang's fast unit cost algorithm [14]
ZhangShashaUCATM	Shasha and Zhang's fast unit cost algorithm modified for cuts (section 5.1.2 on page 39)
IsolatedSubtreeATM	Zhang's algorithm for the isolated-subtree mapping modified for cuts [23]

Table A.1: List of algorithm implementations.

The implementation of the first four algorithms is naturally very similar. They are implemented as bottom-up dynamic programs as described in the papers. They have a shared `_distmap` two-dimensional list which represents the permanent table. The function `_treedist()` computes the distance between a pair of subtrees, and local to this function is a two-dimensional list, `_forestdist`, which is the temporary table.

The first four algorithms have been implemented such that edit scripts are computed with constant overhead. Edit scripts are linked lists and a reference to the head of the list is stored together with the cost of each subproblem. When relabeling nodes we need to concatenate a list from the permanent table and from the temporary table. To do this in constant time we maintain a reference to the tail of the lists as well. The edit script of a subproblem in the permanent table may be part of a solution to a bigger subproblem which means it has other edit scripts concatenated to it. A subproblem may later be needed in an even bigger subproblem, so to be able to detach the subproblem from other edit scripts, the tail is a reference to the end of the edit script of the subproblem and not the actual tail of the linked list.

Zhang's algorithm for the isolated-subtree mapping also appears in a version for pattern matching in the paper. However, Zhang has chosen to give the pattern as the first input to the algorithm and the cut operation on a node only removes the children of said node. Consequently, insert and delete operations are swapped in the edit script and the cut operation has the same cost as deleting the root of the subtree to be cut. For conformance to the other algorithms we have corrected for the misbehaviour in our implementation. Although not impossible, this implementation does not compute edit scripts in constant time.

UnitCost implements the simple unit cost cost function. Its `relabel()` function considers both label and attributes of the nodes when comparing them. For two nodes to be equal, the latter node must have the same label and the same attributes with the same values as the first node. Cost function classes must implement the functions `insert()`, `delete()`, and `relabel()`.

A.2.2 `context`

This module contains the class PatternMatchingContext. The purpose of this class is to create a mapping between the datastructure used by the lxml library and the generic datastructure used by the algorithms, and to create a matching from the edit script produced by the algorithm. This is handled by the following functions.

`make_mapping()` Requires the root of a lxml tree as input parameter. Then it copies the provided lxml tree to the generic datastructure Node and sets the `map` field to reference to the corresponding lxml nodes.

`make_matching()` Creates a matching from the edit script generated by the algorithm. The matching is a dictionary with lxml nodes as keys and lxml nodes as data.

`get_matching()` Executes the above functions and the approximate tree matching algorithm.

The module also serves as an example of how to write a context.

A.2.3 `datastructures`

This module contains the class Node which is the generic datastructure used by the algorithms. The important fields of Node are:

`index` The nodes index in the tree. The node indices must be set manually or by calling the `set_postorder_indices()` function on the root node when the tree has been built.

`label` A string label of the node. In the web scraping context this is the tag name.

`attr` A dictionary of attributes. In the web scraping context this is the tag attributes `id`, `class`, and `name` if available for the given tag.

`children` An array of Node instances. It is empty if the node is a leaf.

`map` This field can be used as a reference to a node in another datastructure.

Additionally it contains some fields used by the algorithms. The Node class also implements a number of functions which operate on the tree that is rooted at the node. Most of these function are required by the algorithms and the decorators. Refer to the comments in the source code for further descriptions of the functions.

The module also contains the class EditScriptEntry which is an element in a linked list and has the following three fields.

`operation` A string defining the operation, i.e. `insert`, `delete`, `relabel`, and `cut`.

`apply_to` If the operation is insert, this is a tuple of Node instances. The first instance is the node to insert and the second, optional, instance is the node which becomes a child of the new node. If the operation is delete or cut, this is the Node to remove. It is a tuple of Node instances if the operation is relabel.

`next` A reference to the next entry in the edit script.

A.2.4 `decorators`

The module contains the classes LinearMatching, PreSelection, LowerBoundPruning, and HTMLPruning which implement the pruning methods described in chapter 5 as decorators for the algorithms. This means they have the same interface as the algorithms, but their constructors require an algorithm (or another decorator) to be given as input. This is the *core* algorithm seen from the point of view of the decorator and will be invoked at some point in the algorithm implemented by the decorator.

When using the PreSelection decorator the edit script will be missing some delete and cut operations. The algorithm selects one or more subtrees to match and applies the core algorithm to these. The result is an edit script applying only to one of the selected subtrees. To correct this, the edit script would have to be extended by the deletion of all nodes on the path from the actual root of the data tree to the root of the subtree, and the cut of all other branches. Since it introduces further overhead to add these operation to the edit script, and because it has no influence on the resulting matching, we choose not to implement the correctional behaviour in the decorator.

The HTMLPruning class requires a path to a local DTD file when instanciated. The `_parse_dtd()` function will be invoked and the information about allowed tag relations will be stored in the local dictionary `_elements`.

A.2.5 `lowerbounds`

The module contains the classes Size, Height, QGram, and EulerString which implement the approximation methods described in chapter 4. To use the classes with the LowerBoundPruning decorator they must accept two Node instances (the roots of the trees) as input and return an integer.

To represent q-grams we use a dictionary that maps tuples of nodes (paths) to a list of indices to the nodes where a given path appears.

The EulerString class uses a string edit distance algorithm from Wikibooks [24]. It has been modified such that deletes are free, cf. the requirements for the method to produce a lower bound, and it operates on the datastructure produced by the `euler_array()` function on Node in the `datastructures` module.

A.3 Examples

In this section we give some examples of how to use the application. The PatternMatchingContext class serves as an example on how to write a context.

A.3.1 Tree Edit Distance and Printing Edit Script

The first example shows how to get the tree edit distance and print the edit script for two trees, `t1` and `t2`.

```python
from algorithms import ZhangShashaATM, UnitCost
```

```

atm = ZhangShasha(UnitCost())
atm(t1, t2)
print 'Edit distance: ' + str(atm.get_distance())
eds = alg2.get_edit_script()
while eds:
  print str(eds)
  eds = eds.next
```

A.3.2 Using Pruning

In this example we show how to use a decorator with the approximate tree matching algorithm. First we instanciate the algorithm which is then passed as input to the LowerBoundPruning decorator when the latter is instanciated.

```
from algorithms import ZhangShashaATM, UnitCost
from lowerbounds import QGram
from decorators import LowerBoundPruning

k = 10
q_gram_size = 3
atm = ZhangShashaATM(UnitCost())
lwb = QGram(q_gram_size)
dec = LowerBoundPruning(atm, k, lwb)
dec(t1, t2)
print 'Edit distance: ' + str(dec.get_distance())
```

A.3.3 Combining Decorators

Finally we show how to combine four decorators. In this example we apply pre-selection followed by linear matching. Neither of these are guaranteed to reduce the data tree, so regardless of the outcome we apply Euler string distance pruning and q-gram distance pruning.

The order in which the decorators are applied is significant. For instance, it makes no sense to apply lower bound pruning before pre-selection, because the lower bound algorithm will spend computation time removing branches from parts of the data tree that are not considered after pre-selection.

```
from algorithms import ZhangShashaATM, UnitCost
from lowerbounds import QGram, EulerString
from decorators import PreSelection, LinearMatching, LowerBoundPruning

k = 10
q_gram_size = 3
atm = ZhangShashaATM(UnitCost())
lwb1 = QGram(q_gram_size)
lwb2 = EulerString()
```

```python
dec1 = LowerBoundPruning(atm, k, lwb1)
dec2 = LowerBoundPruning(dec1, k, lwb2)
dec3 = LinearMatching(dec2)
dec4 = PreSelection(dec3)
dec4(t1, t2)
print 'Edit distance: ' + str(dec.get_distance())
```

Appendix B

Algorithms

B.1 Algorithm for finding Q-grams

Below the algorithm for finding Q-grams is shown. A q-gram profile is a tuple $(v_x, v_y, \ldots, v_z)$ of nodes.

The Q-gram algorithm takes a tree T and a value q as input and computes all paths of length $1, 2, \ldots, q$ for each subtree $T(v)$. The index of the highest node is stored in a dictionary *qgram* (Q-gram profile $\rightarrow$ [index]) at v. It also uses a dictionary *path* (length of path $\rightarrow$ [Q-gram profile]) for each node v to store all q-grams starting at v. This is to avoid computing a q-gram more than once. The running time of the algorithm is $O\big(q|T|^2\big)$.

B.2 Algorithm for finding Q-samples

The q-sample algorithm takes a tree T and a q value as input and computes as many disjoint q-grams of size at most q as possible. It is a greedy algorithm in the sense that it will attempt to create the biggest possible q-gram for the current node. Since this may result in one large and several small q-grams it

Algorithm 3 Q-GRAM(T, q)

```
1: for each node $v_x$ in $T$ in postorder do
2:    $v_x.qgram[(v_x)] \leftarrow [x]$
3:    for each child $w_y$ of $v_x$ do
4:       for each q-gram profile $p$ in $w_y.qgram$ do
5:          $v_x.qgram[p] \leftarrow v_x.qgram[p] \cup w_y.qgram[p]$
6:       if $q \geq 2$ then
7:          $v_x.qgram[(v_x, w_y)] \leftarrow v_x.qgram[(v_x, w_y)] \cup [x]$
8:          $v_x.path[2] \leftarrow v_x.path[2] \cup (v_x, w_y)$
9:          for $i \leftarrow 2 \ldots q$ do
10:            for each path $p$ in $w_y.path[i]$ do
11:               $v_x.qgram[(v_x) \oplus p] \leftarrow v_x.qgram[(v_x) \oplus p] \cup [x]$
12:               $v_x.path[i+1] \leftarrow v_x.path[i+1] \cup (v_x) \oplus p$
```

may not be an optimal strategy. It uses the same dictionary *qgram* as Q-GRAM. The running time of the algorithm is $O(|T|^2)$.

Algorithm 4 Q-SAMPLE(T, q)

```
1: Let $nodes$ be a FIFO queue of the nodes of $T$ in post order
2: while $nodes \neq \emptyset$ do
3:    $v_x \leftarrow nodes.pop()$
4:    $k \leftarrow q$
5:    $s \leftarrow (v_x)$
6:    $n \leftarrow v_x$
7:    while $n$ has at least one child and k $> 1$ do
8:       Let $n$ be the some child $c$ of $n$
9:       Remove $c$ from $nodes$
10:      $s \leftarrow s \oplus (n)$
11:      $k \leftarrow k - 1$
12:   $v_x.qgram[s] \leftarrow x$
13: for each node $v$ in $T$ in postorder do
14:   for each child $w$ of $v$ do
15:      for each q-gram profile $p$ in $w.qgram$ do
16:         $v.qgram[p] \leftarrow v.qgram[p] \cup w.qgram[p]$
```

Appendix C

Test Case Patterns

C.1 Case 1

```
<div class="st_content container_24">
  <div class="grid-wrapper clearfix">
    <div class="grid_24 panel-region region-bottom">
      <div class="panel-pane">
        <div class="content">
          <div class="section">
            <section>
              <div class="grid_5 panel-region d">
                <div class="module block-league-table">
                  <h2 class="section-sub-header section-sub-header-
                       style3">
                  </h2>
                  <table>
                    <tbody>
                      <tr>
                        <td class="text-b last">
                          <a class="black-nl">
                            <img src="" />
                          </a>
                        </td>
                      </tr>
                      <tr>
                        <td class="text-b last">
                          <a class="black-nl"></a>
                        </td>
                      </tr>
                      <tr>
                        <td class="text-b last">
                          <a class="black-nl">
```

```
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                          </td>
                        </tr>
                        <tr>
                          <td class="text-b last">
                            <a class="black-nl">
                              <img src="" />
                            </a>
                        </td>
                      </tr>
                    </tbody>
                  </table>
                </div>
              </div>
            </section>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

C.2 Case 2

```
<table>
  <tbody>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
    <tr>
      <td class="text-b last">
        <a class="black-nl">
          <img src="" />
        </a>
      </td>
    </tr>
  </tbody>
</table>
```

C.3 Case 3

```
<body class="html front not-logged-in one-sidebar sidebar-second">
  <div id="page-wrapper">
    <div id="page" class="clearfix">
      <section id="zones-content" class="clearfix">
        <div id="content-outer-wrapper" class="clearfix">
          <div id="content-container" class="clearfix container-12 zone
              -dynamic zone-content zone content-zone">
            <div id="region-content" class="region region-content
                content-region grid-8 even">
              <div id="block-system-main" class="block block-system
                  block-without-title odd first">
                <div class="block-inner clearfix">
```

```
                  <div class="content">
                    <div class="view-Forside view-id-Forside view-display-id-toparticles view-dom-id-1">
                      <div class="view-content">
                        <div class="frontpage-standalone">
                          <aside class="illustration">
                            <div class="frontpage-standalone-image">
                              <img src="" alt="" />
                            </div>
                          </aside>
                          <section>
                            <div class="frontpage-standalone-node-title">
                              <h1 class="node-title">
                                <a>
                                </a>
                              </h1>
                            </div>
                            <div class="frontpage-standalone-teaser">
                              <p>
                                <a>
                                </a>
                              </p>
                            </div>
                            <div class="article-equipment">
                              <div class="post-date">
                              </div>
                            </div>
                          </section>
                        </div>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </section>
    </div>
  </div>
</body>
```

C.4 Case 4

```
<div class="block-inner clearfix">
  <div class="content">
    <div class="view-Forside view-id-Forside view-display-id-toparticles view-dom-id-1">
      <div class="view-content">
        <div class="frontpage-standalone">
          <aside class="illustration">
            <div class="frontpage-standalone-image">
              <img src="" alt="" />
            </div>
          </aside>
          <section>
            <div class="frontpage-standalone-node-title">
              <h1 class="node-title">
```

```
                <a>
                </a>
              </h1>
            </div>
            <div class="frontpage-standalone-teaser">
              <p>
                <a>
                </a>
              </p>
            </div>
            <div class="article-equipment">
              <div class="post-date">
              </div>
            </div>
          </section>
        </div>
      </div>
    </div>
  </div>
</div>
```

C.5 Case 5

```
<html>
  <body>
    <div class="content">
      <div id="siteTable" class="sitetable linklisting">
        <div>
          <a class="thumbnail ">
            <img />
          </a>
          <div>
            <p class="title">
              <a class="title "></a>
              <span class="domain">
                <a></a>
            </p>
            <p class="tagline">
              <time></time>
              <a></a>
            </p>
            <ul class="flat-list buttons">
              <li class="first">
                <a class="comments"></a>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

C.6 Case 6

```html
<div id="siteTable" class="sitetable linklisting">
  <div>
    <a class="thumbnail ">
      <img />
    </a>
    <div>
      <p class="title">
        <a class="title ">
        </a>
        <span class="domain">
          <a>
          </a>
      </p>
      <p class="tagline">
        <time>
        </time>
        <a>
        </a>
      </p>
      <ul class="flat-list buttons">
        <li class="first">
          <a class="comments">
          </a>
        </li>
      </ul>
    </div>
  </div>
</div>
```