Approximate Membership for Words and Trees

Antoine Mbaye Ndione

Université Lille 1 – Sciences et Technologies
Laboratoire d'Informatique Fondamentale de Lille
Institut National de Recherche en Informatique et en Automatique

Thèse

présentée en première version en vue d'obtenir le grade de Docteur, spécialité Informatique

par

Antoine Mbaye Ndione

Approximate Membership for Words and Trees

Thèse soutenue le 16/04/2014 devant le jury composé de :

Tugkan Batu	London School of Economics	Rapporteur
Michel de Rougemont	Université Paris 2	Rapporteur
Joachim Niehren	Inria	Directeur
Aurélien Lemay	Université de Lille 3	Co-Encadrant
Frédéric Magniez	Université Paris 7	Member du jury
Sophie Tison	Université Lille 1	Présidente

Summary

Inspired by property testing, our objective is to obtain sublinear algorithms for deciding properties of XML databases approximatively. More precisely, we investigate the properties of whether an unranked tree is valid for a DTD, or more generally, whether it is recognized by a tree automaton.

We start our studies by the simpler case of words and we considered the approximate membership problem for word non-deterministic automata. For this problem, we provide an efficient tester that runs in polynomial time in the size of the input automata and the error precision. We also improve the previous [Alon, Krivelevich, Newman, and Szegedy, 2000b] approximate membership tester for regular languages modulo the Hamming distance, so that it runs in polynomial time in the size of the input automata.

Secondly, we study approximate membership testing for tree automata modulo the standard edit distance, and obtain a tester with run time exponential in the input tree depth. Next we consider approximate DTD validity modulo the strong edit distance. We then provide a tester that depends polynomially on the height of the tree. Finally, modulo the strong edit distance, we prove a linear lower bound on the depth of the input tree.

Contents

1 Introduction

Contents

Computers are now used in everyday's life to store and compute various kinds of information. With computers we communicate in social networks on the Web, we store huge amouts of private and business data in the cloud, on database, or on the Web, and we extract information from such data for decision making. In the recent decades, the amount of data stored in various kinds of databases has grown massively. This raises many challenges to computer science and database research in particular, on how to store even bigger amounts of data, and how to efficiently compute and extract information from the stored data. In this context, even linear time algorithms may no more be sufficiently efficient. Instead, one is often interested in sublinear processing time [Rubinfeld and Shapira, 2011], possibly after a linear time precomputation; for instance for computing indexes.

1.1 Sublinear algorithms

The most frequent manner to obtain sublinear algorithms used thesedays is to rely on indexes [see e.g. Garcia-Molina, Ullman, and Widom, 2008, chap 14], parallelism, or both. For instance, Google uses indexes and parallelism for keyword querying [Ghemawat, Gobioff, and Leung, 2003; Dean and Ghemawat, 2004; Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, and Gruber, 2006]. IBM [IBM, 2013] and Amazon [Amazon, 2013] also have tools that use parallelism through the programming model MapReduce, in which data exchange between parallel processes is minimized. MapReduce also lacks supports for schemas and thus may not be optimal with common

database systems. Besides database theory, note also that the consideration of indexes to design efficient algorithms is considered in many domains such as bioinformatics [Simpson and Durbin, 2010], programming languages (with hash maps) and Web information extraction [Cafarella, Downey, Soderland, and Etzioni, 2005].

Another way in which sublinear algorithms may be obtained is to consider *randomized algorithms*. Randomized techniques, based on computing "sketch" synopses i.e. statistics, have been proven to provide efficient algorithms for answering aggregate queries in streaming databases [Cormode and Muthukrishnan, 2007]. While some problems can be exactly solved by sublinear randomized (even deterministic [Iwen, 2008]) algorithms, it should be noticed that in most cases the answer provided by sublinear randomized algorithms is in some sense approximate.

The approach we take in this thesis for designing sublinear algorithms is *Property Testing* [Goldreich, Goldwasser, and Ron, 1998]. However, this approach has rarely been investigated for databases, with the expect of the work of de Rougemont and Vieilleribière [2007] on approximate data exchange in XML databases. Property testing's objective is to design sublinear or even constant time algorithms that approximately decide properties of their inputs by simply sampling a small portion of their inputs. We are specially interested to study property testing of data sets such as XML databases. More precisely, we seek to design sublinear algorithms that approximately decide properties of unranked trees, as in the XML data model.

1.2 XML and schema validation

The Extensible Markup Language (XML) has been introduced two decades ago by the W3C [Bray, Paoli, Sperberg-McQueen, Maler, and Yergea, 2008b]. XML has been standard for exchanging data on the Web and in document processing, XML is used to represent documents and transformations (DocBook, SGML). XML have been intensively used these last years, for example, to represent data structures in web services and XML databases have also gained much attention. The XML data model [Berglund, Boag, Chamberlin, Fernández, Kay, Robie, and Siméon, 2010] provides unranked data trees as representation of XML documents and thus usually XML is formally studied by modelling XML documents by trees. XML also comes with schema languages of which DTDs (Document Type Definition) [Bray, Paoli, Sperberg-McQueen, Maler, and Yergea, 2008a] are the simplest ones. Other schemas are XML SCHEMAS [Fallside and Walmsley, 2004] and RELAXNG [van der Vlist, 2003]. Schemas are sets of rules that defines a set of valid XML documents with respect to some concrete application. In formal study of XML schemas, for example in studies of their expressiveness, schemas are often translated into some kind of unranked tree automata [Hosoya and Pierce,

```
<collection>
  <book>
    <title>Principia Mathematica</title>
    <author>Russell</author>
    <author>Whitehead</author>
    <year>1913</year>
  </book>
  <book>
    <title>M.W.M.W.W.S</title>
    <author>Cavell</author>
    <year>1969</year>
  </book>
</collection>
```

Figure 1.1: An XML document representing a collection of books

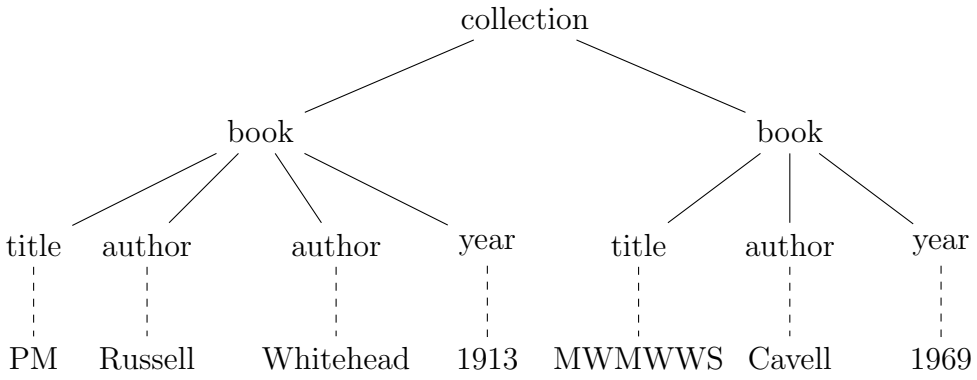


Figure 1.2: The tree representation of the XML document at Figure 1.1

```
<!DOCTYPE collection[
  <!ELEMENT collection     (book*)>
  <!ELEMENT book           (title, author+, year)>
  <!ELEMENT title          (#PCDATA)>
  <!ELEMENT author         (#PCDATA)>
  <!ELEMENT year           (#PCDATA)>
]>
```

Figure 1.3: A DTD satisfied by the XML document at Figure 1.1

2001; Klarlund, Møller, and Schwartzbach, 2000; Lee, Mani, and Murata, 2000; Murata, 1998].

For instance, the XML document of Figure 1.1 represents a collection of books. The tree representation of this document can be found at Figure 1.2, and we also provide a DTD satisfied by the document. This DTD may be used for example in an simplified application with libraries.

A prime task is to check whether some XML document is valid for a given

schema. When modelling XML documents by trees and schemas by tree automata, we can study the *schema validation* task, as the membership problem for trees regular languages. Note that in this abstraction, XML data values are not considered; since for sake of simplicity we are interested only with structural aspects of XML documents.

The problem to test membership of trees in tree automata is the prime interest of this thesis. More precisely, using property testing ideas, we study an approximate version of the membership problem for trees regular languages. We next explain which are those ideas by examples.

1.3 Property testing: approximate schema validation

In property testing, the main idea for obtaining sublinear algorithms for schema validation tasks, is to use randomization as described by the following example. We fix a precision parameter $0 < \epsilon < 1/2$ and an alphabet $\Sigma = \{a, b\}$. Consider the following simple DTD D.

```
<!DOCTYPE b[
   <!ELEMENT b (b*)>
]>
```
(D)

An XML document, on the alphabet Σ, is D-valid if and only if it does not contain any a-tag. So the schema invalidation task for D reduces to checking whether an XML document contains some a-tag. Then, with high probability, we can efficiently invalidate all XML documents with more than an ϵ fraction of a-tags. Indeed, it is sufficient to uniformly choose, $O(1/\epsilon)$ tags in the document and check whether one of the selected tag is labelled a. The idea is as follows: whenever an XML document contains many errors, randomized algorithms may find one of these errors with high probability by inspecting a small part of the XML document. Where a document with many errors means it contains at least an ϵ-fraction of a-tags. Documents with many errors are next referred to as being ϵ-*far* from D. Notice that the above method to invalidate documents works, with high probability, only for documents ϵ-far from D. However documents with small amount of errors can be considered almost valid; and we next refer to such documents as being ϵ-*close* to D. Hence, from this example, we can see that randomized algorithms may efficiently and with high probability solve the approximate schema validation task which consist in invalidating all documents ϵ-far from D and accepting all valid XML documents. For documents which are neither valid nor ϵ-far from D, randomized algorithms may err with high probability.

Let us now show how we can generalize the aforementioned ideas to relational structures. Let $\mathcal{K}$ be a class of relational structures (ex. words, trees,

graphs), and $\mathcal{A}$ be a class of language definitions (ex. automata, DTD, logical formulas); that is a class whose elements $A \in \mathcal{A}$ denote sets $L(A)$ of elements of $\mathcal{K}$. In these settings, for some structure $\Gamma \in \mathcal{K}$ and some language definition $A \in \mathcal{A}$, the above schema validation task corresponds to the membership problem for Γ and A, i.e, the problem of checking whether $\Gamma \in L(A)$. To define the *approximate membership problem*, we first need to specify how the number of errors of some invalid structure is measured. For this purpose, we use a distance measure $d : \mathcal{K} \times \mathcal{K}$ over the class of structures. A structure $\mathcal{K}$ is said ϵ-far from A, if and only if, for all structures $\Gamma' \in L(A)$, $d(\Gamma, \Gamma') > \epsilon|\Gamma|$; where $|\Gamma|$ is the size of Γ. Thus, in property testing, the approximate membership problem for $\mathcal{A}$ is the following: for some input structure $\Gamma \in \mathcal{K}$ and input language definition $A \in \mathcal{A}$, we must separate with high probability the case where $\Gamma \in L(A)$ with the other case where Γ is ϵ-far from A. And this without any probability requirements for structures ϵ-close to A but not in $L(A)$.

The golden goal of property testing is to find randomized algorithms (or *testers*) that may solve the approximate membership problem with complexity depending only on ϵ, the size $|A|$ of A and not the size of the structure Γ. Usually the complexity of algorithms solving the *approximate membership* problem is measured in terms of the number of random inspections of the domain of Γ. Thus it is called a query complexity. So, the query complexity depends on how algorithms (testers) inspect the domain of relational structures Γ and how they find out relations between elements of such domain. We next address this dependence by inputting to testers an oracle which randomly accesses the domain of Γ. In previous studies of property testing, a representation of the studied structures were fixed and thus testers were designed according to such representation. In our studies we aim instead at designing testers that are independent of the representations of structures, and rather depend on the allowed accesses to structures. This way reductions can be easily define between approximate membership problems of different kinds of relational structures.

When there exists a tester whose complexity depends only on ϵ and the size of $|A|$, we say that approximate membership of structures $\Gamma \in \mathcal{K}$ for languages defined by definitions $A \in \mathcal{A}$ is *testable*. However, query complexity independent of the size of the input structure is not always possible [Parnas, Ron, and Rubinfeld, 2001] and algorithms sublinear in the size of Γ are also beneficial compared to algorithms solving the exact membership task.

Blum, Codenotti, Gemmell, and Shahoumian [1995] were the first to consider problems of this kind, and the general notion of property testing was first formulated by Rubinfeld and Sudan [1996]. Goldreich, Goldwasser, and Ron [1998] provided formal definitions and also considered property testing as a framework for studying combinatorial objects such as graphs. And recently, many studies consider property testing approach [Batu, Fortnow, Rubinfeld, Smith, and White, 2013; Goldreich and Ron, 2011; Alon, Fischer,

Newman, and Shapira, 2009; Valiant, 2011].

1.4 Approximate membership of words and trees

Our objective is to study property testing of trees for properties denoted by various kinds of tree automata or XML schemas; so one can provide sublinear algorithms for the membership problem of trees in languages denoted by tree automata or XML schemas. As trees are generalization of word structures and tree automata are generalization of word automata, the first step in our study is to study property testing of words with regard to words automata.

The study of property testing of word automata was initiated by Alon, Krivelevich, Newman, and Szegedy [2000b] with respect to the Hamming distance between words, and they proved that approximate membership of word automata is testable. However, Alon, Krivelevich, Newman, and Szegedy [2000b] considered only Deterministic Finite Automata (DFAs), and thus their algorithm is exponential in the size of the NonDeterministic Finite Automaton (NFAs) denoting a regular language. Indeed, while DFAs and NFAs denote the same class of languages, which are called regular languages, it is known that the DFA denoting some regular language can be exponentially bigger than the smallest NFA denoting the same language [Jirásek, Jirásková, and Szabari, 2007]. Therefore, for efficiency reasons, their algorithm is questionable in practice. Alon, Krivelevich, Newman, and Szegedy [2000b], also proved that word languages denoted by context-free grammars are not testable under the Hamming distance. Other studies of word languages considering the edit distance with moves were done by Fischer, Magniez, and de Rougemont [2006]. The edit distance with moves allow more edit operations, thus it is always smaller than the Hamming distance. So, compared to the edit distance with moves, the Hamming distance accounts for more differences between words. Hence approximate membership with the Hamming distance corresponds to a better approximation of the exact membership problem. However, with respect to the edit distance with moves, Fischer, Magniez, and de Rougemont [2006] showed that approximate membership of words is testable for languages denoted by finite automata, as well as for languages denoted by context-free grammars. The edit distance with moves was also extended to trees by Fischer, Magniez, and de Rougemont [2006] and they showed that approximate membership for regular tree languages can also be tested with constant query complexity.

Challenges

The aforementioned studies raise mainly two challenges, that we address in this thesis. The first concerns studying the query complexity of approximate

```
<!DOCTYPE r [
   <!ELEMENT r  (ab)>
   <!ELEMENT a  (a*)>
   <!ELEMENT b  (b*)>
   <!ELEMENT a  (#PCDATA)>
   <!ELEMENT b  (#PCDATA)>
]>
```

Figure 1.4: Example of DTD on the alphabet $\Sigma = \{r, a, b\}$

membership testers not only in term of the size of the input structure and the error precision, but also in term of the size of the language definition (the size of the property). The language was always fixed in previous studies, so its definition size wasn't considered as an important issue for the query complexity. The problem is to improve the query complexity of testers in terms of the size of the language definition.

The second challenge is more about which kind of the approximation, thus which distance, is appropriate for schema validation. Indeed while the edit distance with moves provides testers with constant query complexity, we illustrate by the following example that it may not appropriately distinguish XML documents, or trees, when the order of nodes is considered.

Let D be the DTD of Figure 1.4. Note that any D-valid tree must contain a root labelled r and below the root are two subtrees such that the first subtree is labelled a and the second one is labelled b. We recall that distances are used in property testing to account for the number of errors in invalid trees, for some membership problem.

Now, for $n \in \mathbb{N}$, we consider the tree

$$\mathrm{t} = r(b(\underbrace{b, \cdots, b}_{n \text{ times}}), a(\underbrace{a, \cdots, a}_{n \text{ times}}))$$

For the membership problem involving t and D, note that t is not D-valid. To see this, it suffices to notice that in the preorder tranversals of all D-valid trees, a-nodes must appear before b-nodes; and in t all b-nodes appear before a-nodes. Next we compare t with the following tree:

$$t' = r(a(\underbrace{a, \cdots, a}_{n \text{ times}}), b(\underbrace{b, \cdots, b}_{n \text{ times}}))$$

It is rather straightforward that the edit distance with moves between t and t' is only 1. This is because, the edit distance with moves accounts for the minimal number of relabelling, insertions and deletions of nodes, and also subtree moves, which are necessary to transform t into t'. So we can transform t into t' only by one move operation. Thus the conclusion is that property testing of trees under the edit distance with moves may be irrelevant in cases where one wants to detect structural errors due to the order of appearance of tags in an XML document.

Contributions

Our contributions are motivated by our aim to answer the previously two mentioned challenges. First we study approximate membership of word regular languages for both Hamming and edit distances. We provide efficient approximate membership testers for these distances. Secondly, in order to remedy the fact that the tree edit distance with moves weakly approximates the exact tree membership problem, we study property testing of trees for more restrictive distances. We consider two distances: the standard edit distance [Bille, 2005] and the strong edit distance [Selkow, 1977]. In the standard edit distance, the operations to modify a tree are only insertion, deletion or relabelling of nodes, whereas in the strong edit distance deletion and insertion are restricted to leaf nodes. Note that the strong edit distance is more restrictive than the standard edit distance between trees, and both distances are more restrictive than the edit distance with moves. Thus approximate membership testing with the strong edit distance corresponds to a better approximation of the exact membership problem.

1.5 Approximate membership for word regular languages

In the case of approximate membership for word regular languages, we consider both the Hamming distance and the edit distance between words, and regular word regular languages are denoted by NFAs. We show that the corresponding approximate membership problems can be solved in polynomial time with respect to the size of the input NFA. The query and time complexities of our testers are independent of the size of the input word; instead they only depend on the precision parameter. We can design randomized algorithms that do not read their whole input words, by representing a word by some array reference whose size is also input to our algorithms. Or else we suppose that our randomized algorithms are provided an oracle that accesses some word as follows. The oracle can uniformly generate some position of the word it accesses, it can also generate the successor of any position; and it also tells whether some position precedes another one.

For the edit distance, we define the notion of blocking fragment as witness of farness. Such a fragment is a set of positions of the word such that on the consecutive subwords defined by those positions, one fails to run the NFA considered in the approximate membership problem. To run an NFA on a fragment defining consecutive subwords with holes, on any subword, we proceed with states that are reachable from the last state of the previous subword. Next, for some precision parameter ϵ, we show that words ϵ-far from an NFA A contain many blocking fragments, whereas words recognized by A have no blocking fragment. Thus our tester randomly selects some

fragment and tests whether it is blocking.

Applying the same ideas to the Hamming distance, we obtained a notion of infeasible fragment and thereby we improved the algorithm of Alon, Krivelevich, Newman, and Szegedy [2000b], so that it runs in polynomial time in the size of the input NFA.

1.6 Approximate DTD validity

We also study the approximate membership for tree regular languages modulo the strong and standard edit distances. As for words, we design randomized algorithms that input an oracle access to some tree. The oracle permits to generate the root of the tree, and to access the firstchild, nextsibling, parent and depth of a given node. It can also uniformly generate some descendent of a given node. The first natural approach for studying this approximate membership problem is to verify whether it does not reduce to some approximate membership problem for regular word languages.

For the standard edit distance, we exploit such approach to obtain a "sublinear" tester for the approximate membership problem for tree regular languages. Indeed we use the relationship from [Akutsu, 2006] which relate the standard edit distance between two trees with: their heights and the edit distance of their linearizations. Then we compile the tree automata denoting some regular tree language into an automata recognizing exactly the linearizations of valid trees. Thereby, we obtain a tester whose query and time complexities are exponential both in the depth of trees and in the size of the regular tree automata.

This first approach has two weaknesses. First it is not applicable to the strong edit distance since there is no interesting relationship between the strong edit distance of two trees and the edit distance of their linearizations. Secondly, the tester obtained by the approach for the standard edit distance has complexities exponential in the height of trees. So it is necessary to consider a more direct approach for the simpler case of approximate DTD validity.

We contribute that approximate DTD validity, modulo the strong edit distance, is possible with query complexity depending polynomially on the depth of the input tree and not on its size. This yields a tester with constant query complexity for the class of trees with bounded depth. We first introduce a notion of weighted words, and next we relate approximate DTD validity to the approximate membership of weighted words for languages denoted by NFAs. We then strengthen our result by a lower bound proving that the complexity of any DTD validity tester is at least linear in the depth of t. Our lower bound also shows that, for the strong edit distance, a sublinear algorithm is not possible, for trees whose size is of almost of the same order as their depth.

Random access to relational structures: our framework As we have previously discussed, for the approximate membership problems of some class of relational structures (ex. words or trees), the query complexity of a tester is the maximal number of accesses to the input structures. Clearly, the query complexity depends on how these accesses are performed. We explained that in previous studies, a fixed representation was supposed for the studied class of structures. We instead want to design testers that are independent of the possible representations, and we consider that testers input oracles that access structures. In order to formalize such accesses we introduce the notion of *random objects* for a class of relational structures of some fixed vocabulary. In our framework, testers will input random objects associated to relational structures. A random object for some structure is just a set of functions (possibily random) or queries which access or sample the structure domain and relations. Random objects also have queries about the sizes of relations in the structure. For instance when words over the alphabet Σ are seen as relational structures over the vocabulary $\sigma = \{start, <, (lab_a)_{a\in\Sigma}\}$; we can for example specify a random object that uniformly generates a successor of any position ($<$) by a random function, generates the starting symbol ($start$) and tells whether some position is labelled a (lab_a). We believe this formalism is a guideline on how to develop property testing's algorithms in various kinds of approximate membership problems involving relational structures.

1.7 Outline

Chapter 2 introduces preliminaries on sets, probability and randomized algorithms, it also introduces the main objects that we study (ex. words, trees) as well as the notion of edit distance between such objects.

Chapter 3 introduces relational structures, the model checking problem for such structures and it presents property testing as approximate model checking. In this chapter we also discuss how the notion of property for relational structures is denoted in computer science using automata, logical languages or schemas. We then present previous studies of model checking and property testing.

Chapter 4 introduces our framework for studying property testing of relational structures. We define random objects of relational structures and explain which queries are provided by random objects of tree, word and graph structures. We also provide hints on how these random objects can be efficiently implemented using usual representations of words, trees and graphs.

Chapter 5 studies property testing of words for NFAs modulo the Hamming distance and the edit distance. We propose a tester with 'constant' query

complexity polynomial in the size of the input NFA.

Chapter 6 studies approximate DTD validity modulo the strong edit distance. We propose a tester whose complexity depends only on the depth of trees. We also provide a lower bound showing that at least linear dependence to trees depths is required.

1.8 Publications

Our results concerning approximate membership of words for NFAs were published in *Theoretical Computer Science* [Ndione, Lemay, and Niehren, 2013]. In this paper we have presented our results using array references to words and in this manuscript we explain how random objects can access words as one does with array references. Our DTD validity tester is in a process of submission.

2 Preliminaries

Contents

We introduce in this chapter our notations, which are quite standard, and basic concepts. In Section 2.1.1, we briefly introduce concepts and notations for sets, relations and functions. For a formal and detailed definition of sets we refer to the book of Kunen [1980]. Later in Sections 2.1.2, 2.1.3, 2.1.4 definitions of words, trees, graphs are respectively provided. Words, trees and graphs are what we refer to when using the name "object". Formally they can be seen just as sets with some relations. In Section 2.2, we discuss the notion of distances and a general method to define *edit distances* between elements of some set is discussed. And we use this method to define edit distances between functions, words, trees and graphs. Probabilities are discussed in Section 2.3, where we introduce probability distribution on finite sets and the notion of random function based on an indexed collection of distributions.

2.1 Basic concepts, notations, basic objects

2.1.1 Sets, relations, functions

The cardinality of a set S will be denoted by $|S|$. The set of all subsets of S is called the power set of S and will be denoted by 2^S. The Cartesian

product of two sets S_1 and S_2 is the set of pairs (x, y) satisfying $x \in S_1$ and $x \in S_2$. We use the notation $S_1 \times S_2$ for the Cartesian product of S_1 and S_2. Thus $S_1 \times S_2 = \{(x, y) \mid x \in S_1 \wedge y \in S_2\}$. The set of integers is denoted by $\mathbb{N}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ is the set of natural numbers and $\mathbb{R}_{\geqslant 0}$ is the set of positive real numbers. The set $\mathbb{B} = \{true, false\}$ is the set of booleans. Both natural orders of $\mathbb{N}_0$ and $\mathbb{R}_{\geqslant 0}$ are denoted by $<$, and $\leqslant$ designates the reflexive closure of $<$. To denote the inverses of these aforementioned orders we use the symbol $>$, and $\geqslant$ denotes the reflexive closure of $>$. For natural numbers $i, j \in \mathbb{N}_0$ (resp. $x, y \in \mathbb{R}_{\geqslant 0}$) the *interval* $[i, j]$ (resp. $[x, y]$) is defined as the set of natural numbers (resp. positive real numbers) which are: less or equal to j and greater or equal to i (resp. less or equal to y and greater or equal to x). We also use notations $[i, j[$, $]i, j]$ or $]i, j[$ to denote *open* intervals where as usual the bounds at the opened brackets are not elements of the denoted set of integers. Note that we abusively use the same notation for both intervals of natural numbers and intervals of real numbers. Which type of interval we denote should always be clear from the context. For all integers $i \in \mathbb{N}$, the set S^i is inductively defined in the following: $S^i = S$ if $i = 1$ and $S^i = S^{i-1} \times S$ otherwise. Elements of S^i are called *tuples* and can be identified to sequences of elements of S and they are denoted by $(x_1, \cdots, x_i)$, where $x_j \in S$ for all $1 \leqslant j \leqslant i$. Hence we define $S^0 = \{\varnothing\}$ (or $\{()\}$) and identifies it to the set containing only the empty sequence as element. The set of all tuples is denoted by $S^* = \cup_{i \in \mathbb{N}_0} S^i$. A *prefix* of the sequence $(x_1, \cdots, x_i)$ is any sequence $(x_1, \cdots, x_j)$, where $j \leqslant i$. Then the empty sequence is a prefix of any other sequence and a sequence of length i has exactly $i+1$ prefixes. Note that the prefix relation between two elements of S^* is an order relation. A subset S' of S^* is called prefix closed if for all elements $w \in S'$, S' contains all prefixes of w.

A *relation* R is a set whose elements are ordered pairs, i.e, there are two sets S_1 and S_2 such that $R \subseteq S_1 \times S_2$. The set $\{x \mid \exists y, (x, y) \in R\} \stackrel{def}{=} dom(R)$ is called the domain of the relation R and $\{y \mid \exists y, (x, y) \in R\} \stackrel{def}{=} ran(R)$ is the range of R. We also use the terminology R is a relation between (on) S_1 and S_2 when $R \subseteq S_1 \times S_2$. For an integer $k > 1$, we define a *k-ary relation* over a set S to be any relation between S^{k-1} and S. Then the domain of any k-ary relation over S is a subset of S^{k-1} and its range is also a subset of S. The inverse of a relation R is defined as the relation $R^{-1} = \{(x, y) \mid (y, x) \in R\}$. The image of the relation R on the subset S of $dom(R)$ is defined as the set $\{y \mid \exists x \in S \cap dom(R), (x, y) \in R\} \stackrel{def}{=} R(S)$. We will in the following also use the notation $R(x, y)$ for $(x, y) \in R$. The composition of two relations $R \subseteq S_1 \times S_2$ and $R' \subseteq S_2 \times S_3$ is the set $\{(x, z) \in S_1 \times S_3 \mid \exists y \in S_2, R(x, y)\ and\ R'(y, z)\} \stackrel{def}{=} R' \circ R$, where S_1, S_2, S_3 are sets. For integers $i \geqslant 0$, the i^{th} iteration of the relation R is denoted $R \circ_i R$ and is inductively defined such that $R \circ_{i+1} R = R \circ (R \circ_i R)$; where $R \circ_0 R = R$. And the transitive closure of any relation R is the relation

$R^+ = \cup_{i\in\mathbb{N}_0}(R \circ_i R)$.

A *function* f is a relation such that for all elements x of the domain of f, there is only one element y such that $(x, y) \in f$. Such element y is denoted by $f(x)$. The set $S_1 \to S_2$ of *total functions* from a set S_1 to another set S_2 is the set of functions f such that: $dom(f) = S_1$ and $ran(f) \subseteq S_2$. We also use the notation $f : S_1 \to S_2$ for $f \in S_1 \to S_2$. A *partial function* $f : S_1 \hookrightarrow S_2$ from S_1 to S_2 is a function such that $dom(f) \subseteq S_1$ and $ran(f) \subseteq S_2$. If not explicitly stated the opposite we will consider only total functions from S_1 to S_2. This is not important since any partial function $f : S_1 \hookrightarrow S_2$ can be identified to a total function of $dom(f) \to S_2$. A *bijection* or one to one correspondence is any total function f such that for every element $y \in ran(f)$, there is an unique element $x \in dom(f)$ satisfying: $f(x) = y$. If f is a one to one correspondence, so is its inverse f^{-1}. The identity function $id_S : S \to S$ is the bijection of S satisfying for all elements e of S: $f(e) = e$.

2.1.2 Words

An alphabet Σ is a finite set of symbols. A word $w = (a_1, \ldots, a_n) \in \Sigma^n$ over alphabet Σ is a finite sequence of labels, $n \in \mathbb{N}_0$ and for $i \in [1, n]$: $a_i \in \Sigma$. The size (length) of w is $n \stackrel{def}{=} |w|$, and we denote w by $a_1 \ldots a_n$. A word of length $n \in \mathbb{N}_0$ is then an element of Σ^n. The set of positions of w is $pos(w) = [1, n]$, and the domain of w is defined by $dom(w) = pos(w) \cup \{0\} = [0, n]$. For $i \in pos(w)$, $w[i]$ denotes the label at position i. We denote the empty word () by ε and the concatenation of w and w' is denoted by $w \cdot w'$. The set of all words over the alphabet Σ is $\Sigma^* = \cup_{n\in\mathbb{N}}\Sigma^n$.

Example 2.1. $\Sigma = \{a, b\}$, $w = abbbaab$, $|w| = 7$, $w[4] = b, w[1] = a$

For $n \in \mathbb{N}$, notice that there is a unique one to one correspondence F from Σ^n to $([1, n] \to \Sigma)$ such that, for every word $w \in \Sigma^n$ and integer $i \in [1, n]$: $F(w)(i) = w[i]$. Thus for words $w \in \Sigma^n$ and functions $f : [1, n] \to \Sigma$, where $n \in \mathbb{N}$, we denote $F(w)$ by f_w and $F^{-1}(f)$ by w_f. Hence more generally we say that w is a word over any set S (possibly infinite) if w is an element of S^*. We will mostly be concerned only about words over finite sets (finite sets for which the name alphabet is reserved).

Fragments, Intervals, Subword, Factors A *fragment* of a word w is a subset of $pos(w)$. Fragments without holes are intervals, hence we often prefer to use the terminology *interval of* w for such fragments. Note that any fragment F of w is a union of consecutive non-overlapping intervals, i.e there exists $k \in \mathbb{N}_0$ and intervals $I_1, \cdots, I_k$ such that for all integers $1 \leqslant i < j \leqslant k$: $I_i \cap I_j = \varnothing$, $F = \cup_{1\leqslant i\leqslant k} I_i$ and for all pairs $(x, y) \in I_i \times I_j$, $x < y$. Furthermore there is a unique such decomposition into intervals that minimizes k. The *factor* wI of w at the interval $I = [i, j]$ is the word of w at

the positions in I: $wI = w[i]\cdots w[j]$. For fragment a $F = \cup_{1\leqslant i\leqslant k} I_i$, where $I_1, \cdots, I_k$ is the minimal decomposition of F into intervals, the subword wF is the sequence of factors $(wI_1, \cdots, wI_k)$. The role of intervals, fragments, factors and subwords is essential in property testing of words, because this is the only information 'queried/sampled' by algorithms for deciding properties of words. This will be fully detailed in Chapter 5 where property testing of words is studied.

2.1.3 Trees

The notion of tree arises in many areas of computer science. For example, in logic, a proof may be seen as a tree. In Linux, files and directories are hierarchically organized as trees and programming languages are parsed as terms that can be abstractly identified to labelled trees. With the introduction of XML as the lingua franca format for exchanging data on the web, trees have lately regained much interest. Indeed, XML documents can be modelled as unranked trees. For a complete and formal introduction to trees, the reader is referred to: "Tree Automata Techniques and Applications", a book by Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Löding, Tison, and Tommasi [2007a]. We herein define ranked and unranked trees, and discuss some of their properties.

A *ranked alphabet* is a pair $\Sigma^r = (\Sigma, arity)$ where Σ is a finite set of symbols and $arity : \Sigma \to \mathbb{N}_0$ is a function which assigns an arity to each symbol of Σ. A ranked alphabet over $\Sigma = \{a_1, a_2, \cdots, a_k\}$ will be denoted by $\Sigma^r = \{a_1^{r_1}, a_2^{r_2}, \cdots, a_k^{r_k}\}$, where $k \in \mathbb{N}$, and for all integers $i \leqslant k$, r_i is the arity of a_i: $arity(a_i) = r_i \in \mathbb{N}_0$. We sometimes abuse notations and denote the ranked alphabet over some alphabet Σ by the same letter Σ. And the arity function of all ranked alphabet will always be denoted *arity*. Thus we often say that a_i is a ranked label of Σ with arity r_i, for $a_i^{r_i} \in \Sigma^r$.

A *tree domain* $\mathrm{D} \subset \mathbb{N}^*$ is a finite, non-empty and prefix-closed subset of $\mathbb{N}^*$ which satisfies: for all words (or tuples) $w \in \mathbb{N}^*$ and integers $i \in \mathbb{N}$, if $w\cdot(i+1) \in \mathrm{D}$ then $w\cdot i \in \mathrm{D}$. Then clearly the condition above can be replaced by: if $w \cdot i \in \mathrm{D}$ then for all integers $j \leqslant i$, $w \cdot j \in \mathrm{D}$. The elements of a tree domain D are called *nodes*. Note that the empty word ε is an element of any tree domain and is called the *root* of the tree domain. Commonly the *parent* (*parent*), *child* (*child*), *first-child* (*fc*), *next-sibling* (*ns*), *previous-sibling* (*ps*) and preorder ($<$) relations are defined over a tree domain as follows: for all nodes $w, w' \in D$

- $child(w, w')$ iff. there exists $i \in \mathbb{N}$, $w' = w \cdot i$
- $parent(w, w')$ iff. $child(w', w)$
- $ns(w, w')$ iff. there exist $w'' \in D$ and $i \in \mathbb{N}$, $w = w''\cdot i$ and $w = w''\cdot(i+1)$

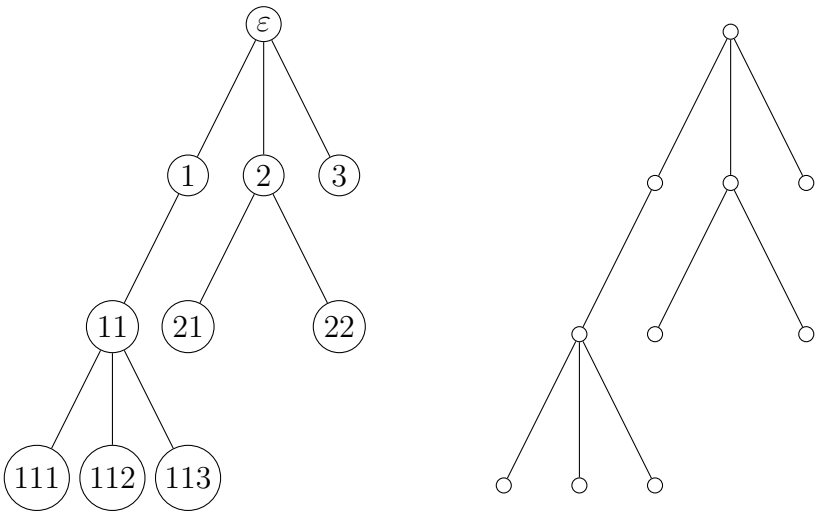


Figure 2.1: The tree domain $\mathrm{D} = \{\varepsilon, 1, 11, 111, 112, 113, 2, 21, 22, 3\}$ is represented in the left side with nodes indicated in circles. Links are the parent relation so in the figure at the right side we drop references to nodes.

- $ps(w, w')$ iff. $ns(w', w)$
- $fc(w, w')$ iff. $w' = w \cdot 1$
- $<(w, w')$ iff. $ns^+(w, w')$ or there exists $w_1, w_2 \in D$, $parent^+(w, w_1)$, $parent^+(w', w_2)$ and $ns^+(w_1, w_2)$

Notice that the *parent*, *ns*, *ps*, *fc* relations are partial functions over the set of nodes. The transitive closure of the child relation is called the *descendant* relation and is denoted by $desc = child^+$. And the transitive closure of the parent relation is called the ancestor relation and denoted by $anc = parent^+$. Usually, the depth of each node w of a tree domain D, is defined by $d(w) = |w|$. Extending the depth function to tree domains, the depth of a tree domain D is the maximal depth of its nodes: $d(\mathrm{D}) = \max_{w \in \mathrm{D}} d(w)$. A *leaf* of some tree domain D is any node $w \in \mathrm{D}$ without children: that is for all $i \in \mathbb{N}$, $w \cdot i \notin \mathrm{D}$. Next we give an example of a tree domain with its classic graphic representation at Figure 2.1.

Example 2.2. $\mathrm{D} = \{\varepsilon, 1, 11, 111, 112, 113, 2, 21, 22, 3\}$ *is an example of a tree domain. Note that we use the same notation than words to denote sequences of elements of* $\mathbb{N}$*. And the empty sequence is also denoted by* ε*. The node* 11 *is a child of* 1*,* 112 *is the next sibling of* 111 *and* 21 *is the fist-child of* 2*.*

A *ranked tree* $\mathrm{t} = (\mathrm{D}, lab)$ over the ranked alphabet $\Sigma^r = (\Sigma, arity)$ is a pair of a tree domain D with a function $lab : \mathrm{D} \to \Sigma$ such that: for all nodes

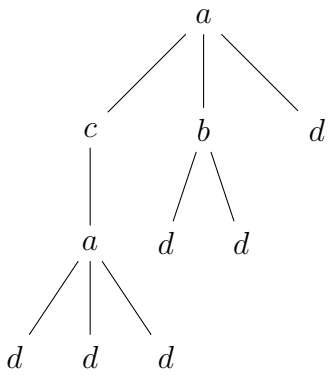


Figure 2.2: A ranked tree

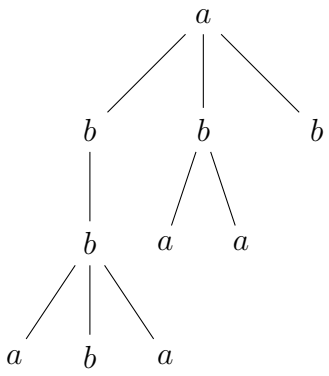


Figure 2.3: An unranked tree

$w \in \mathrm{D}$, $|\{w' \in \mathrm{D} \mid child(w, w')\}| = arity(lab(w))$. Putting it into words the arity of the label of every node is equal to its number of children. The set of ranked trees over the ranked alphabet Σ^r will be denoted by $\mathcal{T}^r_\Sigma$.

An *unranked tree* over the alphabet Σ is a tuple $\mathrm{t} = (\mathrm{D}, lab)$ where D is a tree domain and $lab : \mathrm{D} \to \Sigma$ is a function which assigns labels to nodes of D; and in contrast with ranked trees, this assignment is done without any restriction on the number of children of a node based on its label. An unranked tree is therefore any labelled tree domain and thus all elements of $\mathcal{T}^r_\Sigma$ can also be considered as unranked trees. The set of unranked trees over the alphabet Σ is denoted by $\mathcal{T}_\Sigma$ and it contains all ranked trees on any possible arity function of Σ.

For a (ranked or unranked) tree t, the parent, child, first-child, next-sibling, previous-sibling, preorder, descendent and ancestor relations over the domain of t will be denoted by $parent_\mathrm{t}$, $child_\mathrm{t}$, fc_t, ns_t, ps_t, $<_\mathrm{t}$, $desc_t$, anc_t respectively.

The label function of a (ranked or unranked) tree t will be denoted lab_t. When it is clear from the context, we again abuse notations and denote the label function of any tree simply by lab. The root of any tree t is denoted by $root_t$ or simply by ε. We will also denote the tree domain of any tree t by nod_t, and the size of a tree $|t| = |nod_t|$ is the size of its domain. Note that we already defined the depth of any node of nod_t. The depth $d(t)$ of any tree is the depth of its tree domain: $d(t) = d(nod_t)$. Example of trees are provided at Figure 2.3.

Example 2.3. *Let* $\mathrm{D} = \{\varepsilon, 1, 11, 111, 112, 113, 2, 21, 22, 3\}$ *be a tree domain,* $\Sigma^r = \{a^3, b^2, c^1, d^0\}$ *a ranked alphabet and* $\Sigma' = \{a, b\}$ *an alphabet. Figure 2.2 and Figure 2.3 represent a ranked tree over* Σ^r *and an unranked tree over* Σ' *respectively. The value of the label function on any node is placed at the node position in the graphic representation (see Figure 2.1) of the tree domain* D.

We further will denote trees by terms. A term is first inductively defined in what follows for all nodes of some tree and the term associated to a tree

will simply be the term of its root. The term $term^{\mathrm{t}}(w)$ representing any node $w \in \mathrm{D}$ of a tree $\mathrm{t} = (\mathrm{D}, lab)$ is defined in the following way:

- $term^{\mathrm{t}}(w) = lab(w)$ if w is a leaf, and else
- $term^{\mathrm{t}}(w) = lab(w)(term^{\mathrm{t}}(w \cdot 1), \cdots, term^{\mathrm{t}}(w \cdot j))$, where $j \in \mathbb{N}$ is the maximum integer such that $w \cdot j \in \mathrm{D}$

The term that represents the tree t is $term(\mathrm{t}) = term^{\mathrm{t}}(\varepsilon)$. It is straightforward that the term function so defined is injective, that is for any two trees t, t′, $term(\mathrm{t}) = term(\mathrm{t}')$ iff. $\mathrm{t} = \mathrm{t}'$. The term function is intentionally defined without specifying whether trees considered here are ranked or unraked. The reasons are that as previously noticed, the set of unraked trees contains all possible ranked trees, when arities are ignored; and that our definitions applies to any kind of tree. The term representing the tree at Figure 2.3, for example, is $a(b(a, b, a), b(a, a), b)$. It is also straightforward that for every tree $\mathrm{t} = (\mathrm{D}, lab)$ and node $w \in \mathrm{D}$, there exists a tree t′ such that $term(\mathrm{t}') = term^{\mathrm{t}}(w)$. The tree t′ is called the subtree rooted by w in t, and is denoted by $\mathrm{t}_{|w}$. Notice that $\mathrm{t}_{|w} = (\mathrm{D}', lab)$, where $\mathrm{D}' = \{w' \mid w \cdot w' \in \mathrm{D}\}$ and for all nodes $w' \in \mathrm{D}'$, $lab(w') = lab(w \cdot w')$. The reader may wonder about the usefulness of terms hereinafter. Identifying trees with their terms ease up many definitions as we will soon see with tree encodings. It also facilitates the study of set of trees as formal languages, using rewriting systems, grammar rules and some known results of words languages (even though this has its limits). We furthermore identify the term a with $a()$ for all labels $a \in \Sigma$.

An *hedge* over the alphabet Σ is any sequence $(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_k)$, where k is an integer and for all $i \leqslant k$, t_i is a tree over the alphabet Σ. Hedge are usually denoted by $\mathrm{t}_1 \cdot \mathrm{t}_2 \cdots \mathrm{t}_{k-1} \cdot \mathrm{t}_k$. Note that any tree t over the alphabet Σ can be described by $\mathrm{t} = a(term(\mathrm{t}_1), term(\mathrm{t}_2), \cdots, term(\mathrm{t}_k))$ or equivalently $\mathrm{t} = a(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_k)$, where $a \in \Sigma$ and $(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_k)$ is an hedge over Σ. In such case $\mathrm{t}_1 \cdot \mathrm{t}_2 \cdots \mathrm{t}_{k-1} \cdot \mathrm{t}_k$ is the sequence of subtrees rooted by the children of the root of t. For alphabet Σ, we will use the terms Σ-trees and Σ-hedges for trees and hedges over Σ respectively. The set of hedges is denoted by $\mathcal{H}_\Sigma$.

Let Σ be an alphabet, $n \in \mathbb{N}$, $\mathrm{t} = a(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_n) \in \mathcal{T}_\Sigma$ a tree and $\Box \notin \Sigma$ a place holder symbol. The *context* C^t_w of any node of $w \in nod_t$ is inductively defined as follows:

- $\mathrm{C}^t_\varepsilon = \Box$, and
- $\mathrm{C}^t_w = a(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, \mathrm{C}^{\mathrm{t}_i}_{w'}, \mathrm{t}_{i+1}, \cdots, \mathrm{t}_n)$, where $w = i \cdot w'$ and $i \leqslant n$.

A context is therefore a tree over the alphabet $\Sigma \uplus \{\Box\}$, where the only node with label $\Box$ is a leaf. We could also use different place holders: $\Box_1, \cdots, \Box_k$ where $k \in \mathbb{N}$, and define contexts as trees with leafs labelled by these place holders. However, most of the time we will need only one place holder. We now define for trees $\mathrm{t} \in \mathcal{T}_\Sigma$ and nodes $w \in nod_t$, the substitution of the

subtree $\mathrm{t}_{|w}$ by any tree t'. We denote such substitution by $\mathrm{t}[w \leftarrow \mathrm{t}']$. For $\mathrm{t} = a(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_n)$ and $n \in \mathbb{N}$, the definition of the substitution is inductive as we see below.

- $\mathrm{t}[\varepsilon \leftarrow t'] = t'$, and
- $\mathrm{t}[w \leftarrow \mathrm{t}'] = a(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, \mathrm{t}_i[w' \leftarrow t'], \mathrm{t}_{i+1}, \cdots, \mathrm{t}_n)$, where $w = i \cdot w'$ and $i \leqslant n$.

The definition of substitutions is very related to the one of context. The reason is that $t[w \leftarrow t']$ consists in replacing the place holder of C_w^t by t'. Therefore we will also use the notation $\mathrm{C}_w^t[t']$ for such substitution. To ease up definitions for tree distances in Section 2.2.3, we also introduce, for $\mathrm{t} = a(\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_n)$, the substitution $\mathrm{C}_w^{\mathrm{t}}[H]$ of a node $w = i \cdot w'$ by an Σ-hedge $H = \mathrm{t}'_1 \cdots \mathrm{t}'_k$, where $i \in \mathbb{N}$:

- if $w' = \varepsilon$ then $\mathrm{C}_w^t[H] = a(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, t'_1, \cdots, t'_k, \mathrm{t}_{i+1}, \cdots, \mathrm{t}_n)$, else
- $\mathrm{C}_w^{\mathrm{t}}[H] = a(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, \mathrm{C}_{w'}^{\mathrm{t}_i}[H], \mathrm{t}_{i+1}, \cdots, \mathrm{t}_n)$

Note that the result of the substitution of any node $w \in nod_t$ by the empty hedge is the tree obtained from t after deleting the node w with all descendants of w.

While the notion of unranked trees looks more general than the one of ranked trees, many theoretical studies on trees focus only on ranked trees. One of the main reasons might be that many methods used in the study of formal languages hardly translate to unranked trees whereas they can be generalized to ranked trees without many difficulties. As an example many nice properties of words automata, such as determinism, are hard to define on possible generalisations of automata for unranked trees. And also, the algebraic approach to study formal languages hardly generalizes to unranked trees. Another reason is that unranked trees can be encoded into binary trees, so we can use results of formal languages theory (at the cost of the used encoding). Two types of encodings are commonly used: the *first-child-next-sibling* (fcns) and the *curried* (@) encodings. We detail these encodings below.

First-child-next-sibling encoding: When we are concerned with representing general (thus unranked) trees with data structures such as arrays or lists, it is possible to store each node as a label and a set of pointers to its children. This would require, for each node, to store a number of pointers that can be almost equal to the total number of nodes. However this is not the best way to store trees with less memory, and the first-child-next-sibling encoding allows us to do better. In fact we could store each node as a label, with only two pointers. One pointer for the node first-child and the other one

for its next-sibling. Starting from the root, the tree obtained by following at each node, first the first-child pointer and after the next-sibling pointer, is called the first-child-next-sibling encoding of the stored general tree. We next show this encoding on an example at Figure 2.4 and then give a formal definition below.

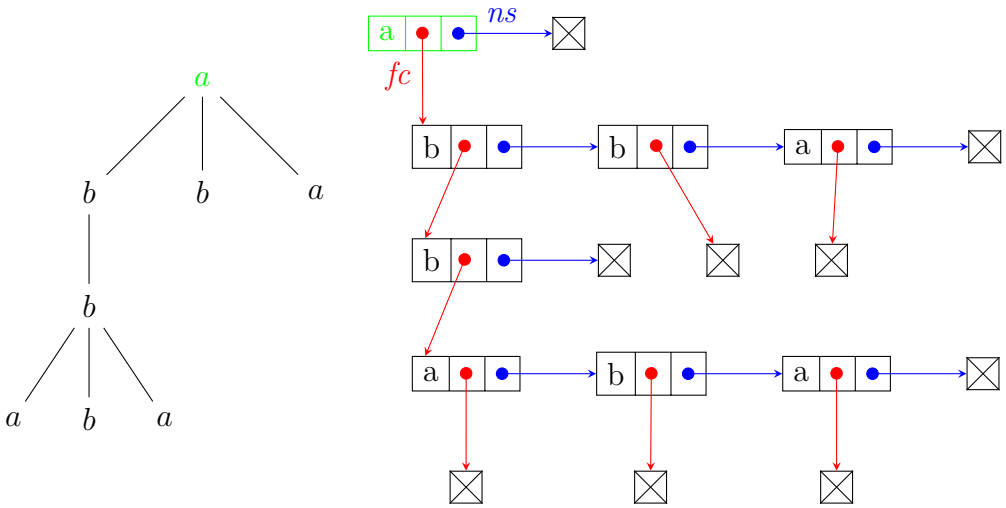


Figure 2.4: A tree with its representation as linked lists. Each node is represented as a list of three elements: the label of the node, a pointer to the node first-child and a pointer to the node next-sibling. The data structure so obtained can be seen as a binary tree. Such binary tree is the first-child-next-sibling encoding of the tree. Note that a crossed rectangle is for a null pointer. The root of the trees are coloured in green.

Let $\perp \notin \Sigma$ be a symbol that is not contained in the alphabet Σ. Using tree terms, we define the first-child-next-sibling encoding generally on Σ-hedges as follows: for all hedges $\mathrm{H} = (\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_n)$

- $\mathrm{fcns}(\mathrm{H}) = \perp$, if $\mathrm{H} = ()$ is empty and
- $\mathrm{fcns}(\mathrm{H}) = a\,(\mathrm{fcns}(\mathrm{t}'_1, \mathrm{t}'_2, \cdots, \mathrm{t}'_k), \mathrm{fcns}(\mathrm{t}_2, \cdots, \mathrm{t}_n))$, if $a \in \Sigma$, $k \in \mathbb{N}$ and $\mathrm{t}_1 = a(\mathrm{t}'_1, \mathrm{t}'_2, \cdots, \mathrm{t}'_k)$.

The first-child-next-sibling encoding of a Σ-tree t is defined as the encoding of the hedge of length 1 containing only t, that is:

$$\mathrm{fcns}(\mathrm{t}) = \mathrm{fcns}((\mathrm{t}))$$

The fcns encoding of any Σ-tree is a binary tree over the ranked alphabet $\Sigma^2 = \{a^2 \mid a \in \Sigma\} \uplus \{\perp^0\}$. All symbols of Σ^2 are of rank 2 except $\perp$ which is

of rank 0. The fcns encoding is extended to any subset S of $\mathcal{T}_\Sigma$ in the usual way:

$$\text{fcns}(S) = \{\text{fcns}(\text{t}) \mid \text{t} \in S\}$$

It is rather simple to prove that the first-child-next-sibling fcns : $\mathcal{H}_\Sigma \to \mathcal{T}^r_{\Sigma^2}$ so defined is a bijection between the set $\mathcal{H}_\Sigma$ of Σ-hedges and the set $\mathcal{T}^r_{\Sigma^2}$ of binary ranked trees over the ranked alphabet Σ^2.

Curried encoding: The perceptive in the curried encoding is rather functional. Indeed, for an alphabet Σ, and over the set of unranked Σ-trees, we define an operator $@ : \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ which allows to construct a Σ-tree from two trees t, t′ such that:

$$@(\text{t}, \text{t}') = \text{t} \ @ \ \text{t}' = a(\text{t}_1, \cdots, \text{t}_n, \text{t}'), \text{where } \text{t} = a(\text{t}_1, \cdots, \text{t}_n) \text{ and } n \in \mathbb{N}$$

Hence any unranked tree can be built, in a unique way, from the alphabet Σ by applying the extension operator @. Then the extension encoding of any tree t is the tree corresponding to the term of the sequence of application of @ which yields t as result. Where we start by applying @ on the symbols of Σ and then applie it again on the results obtained so far. This is detailed in the example at Figure 2.5.

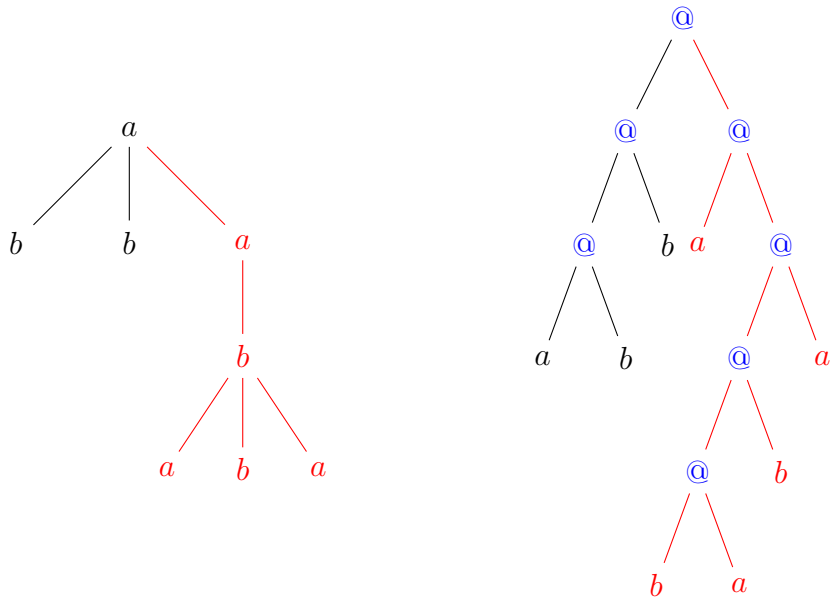


Figure 2.5: The extension encoding of the left side tree $t = a(b, b, a(b(a, b, a)))$ is represented at the right side. The tree t is obtained with $@(@(@(a, b), b), @(@(a, @(@(@(b, a), b), a))))$. The subtrees in red correspond to the curried encoding of the same colour

The symbol @ which denotes the extension operator is also used for the curried encoding. There is no possible confusion since the encoding of trees

is a function of two elements while the extension operator has instead two arguments. Formally, for an alphabet Σ, the curried encoding is defined inductively in the following way: for all Σ-trees t,

- $@(t) = a$, if $t = a$ with $a \in \Sigma$, and
- $@(a(\mathrm{t}_1, \cdots, \mathrm{t}_n)) = @(@(a(\mathrm{t}_1, \cdots, \mathrm{t}_{n-1})), @(\mathrm{t}_n))$, if $t = a(\mathrm{t}_1, \cdots, \mathrm{t}_n)$, $n \in \mathbb{N}$, $a \in \Sigma$ and $\mathrm{t}_1, \cdots, \mathrm{t}_n$ are Σ-trees.

For an alphabet Σ, note that the term obtained by the curried encoding of any unranked Σ-tree is a binary ranked tree over the ranked alphabet $\Sigma^2 = \{a^0 \mid a \in \Sigma\} \uplus \{@^2\}$; where all symbols of Σ have rank 0 and the only symbol of rank 2 is @. Hence, it is rather straightforward that the curried encoding is an bijective function $@ : \mathcal{T}_\Sigma \to \mathcal{T}_{\Sigma^2}$ from $\mathcal{T}_\Sigma$ to $\mathcal{T}_{\Sigma^2}$. This encoding is named 'curried encoding' to emphasise its natural connection to 'curried functions' in functional programming languages or in λ-calculus. In fact the curry encoding sees Σ-trees as functions obtained with the extension operator @ on the set Σ. And writing such functions in their curried form yield curried encoding of trees.

It is clear that the encodings of a general tree can be constructed in linear time on the size of the input tree. Therefore when properties of trees reduce to properties of their encodings (as with regular tree languages), we may only study binary trees. Because then algorithms designed for those properties of binary trees could be used to study general trees only at an additional complexity cost linear in the tree size. However this method is not always possible. As an example, computing these encodings in a streaming manner is hard [see e.g. Konrad and Magniez, 2012]. So in the usual streaming model, where constant memory space is required, this method does not work.

2.1.4 Graphs

Graphs or generally hypergraphs are used in many scientific areas to model objects (*nodes*) that are linked or involved in some relations (*edges*). This high level of abstraction offered by graphs allows us to reduce many scientific problems into graph theory problems. In particular in computer science lots of algorithms are obtained using graph theory [see e.g. Shirinivas and Elango, 2010]. Other domains where graph theory are applied are sociology, biology, operations research. In this section we provide formal definition of *labelled directed multi-graphs*. When the labels and directions are ignored we obtain *graphs*. A complete introduction to graph theory and to related notions can be found in the book of Diestel [2012]: "Graph theory".

Let Σ be an alphabet. A *labelled directed multi-graph* over the alphabet Σ, is a pair $G = (V, E)$ where V is a finite set, $E \subseteq V \times V \times \Sigma$ is a subset of pairs of elements of V with labels in Σ. The elements of V are called *nodes*

and an *edge* of G is any element $e = (v_1, v_2)$ of $V \times V$ such that there exists a label $a \in \Sigma$ satisfying $(v_1, v_2, a) \in E$. Note that an edge may have many labels. When the alphabet is ignored or contains only one single element we identify $V \times V \times \Sigma$ with $V \times V$, and thus G is called simply a directed (*simple*) *graph*. Generally any Σ-labelled directed multi-graph $G = (V, E)$ can also be seen as a directed simple graph $G' = (V, E' \subseteq V \times V)$ along with a label function $lab_{G'} : V \times V \to 2^{\Sigma}$ such that for all edges $e = (v_1, v_2) \in V \times V$ and labels $a \in \Sigma$: $a \in lab_{G'}(e)$ if and only if $(v_1, v_2, a) \in E$. Therefore without lost of generality directed multi-graphs can be identified with directed simple graphs with label functions. Hence, in what follows we only refer to Σ-labelled directed multi-graph, as simple directed graphs, leaving the mention of the alphabet and labels. However this identification cost an exponential growth of the number of labels of the simple directed graph. This alphabet is then mentioned explicitly only when needed and without confusion, the label function of all directed graphs is simply denoted by *lab*.

An *undirected* simple graph $G = (V, E \subseteq V \times V)$ is any simple directed graph such that the *edge relation* E is reflexive, that is for all nodes $v_1 \in V$, $v_2 \in V$: $(v_1, v_2) \in E$ if and only if $(v_2, v_1) \in E$. Furthermore if G has a label function then $lab_G((v_1, v_2)) = lab_G((v_2, v_1))$. In the case of undirected graphs we ease up notations by identifying every two edges $(v_1, v_2), (v_2, v_1)$ by the set $\{v_1, v_2\}$. Below we give a graphical representation of a simple graph without labels at its edges (Figure 2.6).

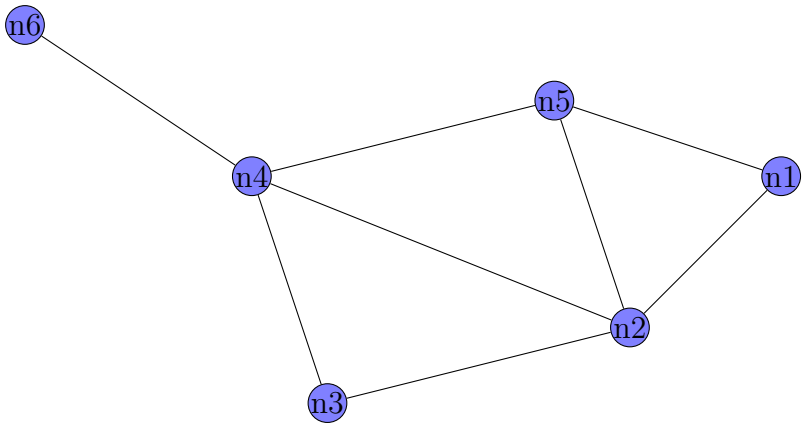


Figure 2.6: Picture representing the simple graph $G = (V, E)$, where the set of nodes $V = \{n1, n2, n3, n4, n5, n6\}$ and the edges are $\{n6, n4\}$, $\{n5, n4\}$, $\{n5, n1\}$, $\{n1, n2\}$, $\{n2, n5\}$, $\{n2, n3\}$, $\{n3, n4\}$, $\{n2, n4\}$.

When a simple graph have label function, we might either put the set of labels of any edge on the arrow representing it, or simply multiply this edge by the size of its set of labels, and put labels separately on these new arrows (Figure 2.7).

For two nodes $v \in V$, $v' \in V$ of some (simple) graph $G = (V, E \subseteq V \times V)$,

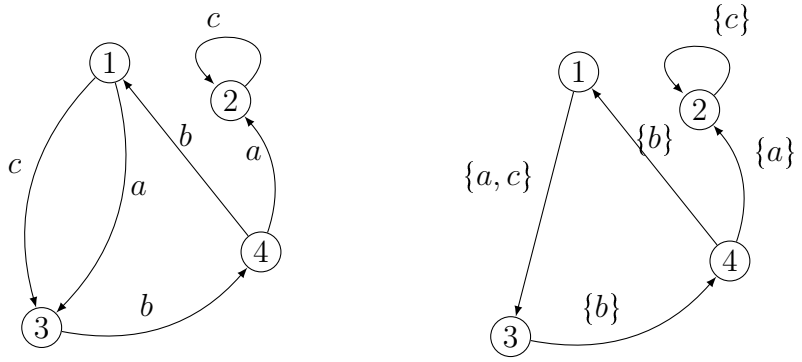


Figure 2.7: Two ways of representing a multi-graph over the alphabet $\{a, b, c\}$. We will next drop the accolades on the edge labels and replace for instance the label $\{a, c\}$ by a, c

we use the notation $v \to v'$ for: (v, v') is an edge of G. A path of G is any sequence of nodes $v_1, \cdots, v_n$ such that for all integers $i < n$: $v_i \to v_{i+1}$. We then say that $p = v_1 \cdot v_2 \cdots v_{n-1} \cdot v_n$ is a path from v_1 to v_n with length $|p| = n$; where n is an integer. We use the notation $v \to^* v'$ for: there exists a path form the nodes v and v'. When G has a label function, we say that the path $p = v_1 \cdot v_2 \cdots v_{n-1} \cdot v_n$ is labelled by a word w, if $|w| = n-1$ and for all natural numbers $i < n-1$, $w[i] = lab_G((v_i, v_{i+1}))$. One remark should be made at this point. Previously, we identified every Σ-labelled directed multi-graph G to a 2^Σ-labelled simple graph G' on the same set of nodes. Thus the label of every path in G' is simply a description of all possible labels of the same path in G.

The size of a simple graph $G = (V, E)$ is defined as the size of its set of nodes plus the size of its edges: $|G| = |E| + |V|$. A graph $G = (V, E)$ is κ-dense, for a real number $\kappa \in [0, 1]$, if its size is at least $\kappa \cdot |V|^2$. We may next talk only about dense graphs, without any κ parameter, and by this we will mean that the size of the graph is almost (or greater than) $|V|^2$.

2.2 Edit distances

An *extended metric* (or *distance measure* or simply *distance*) over elements of some set S is a function $d : S^2 \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ such that: for all elements $x, y, z \in S$

- $d(x, y) = 0$ iff $x = y$, separability
- $d(x, z) = d(x, y) + d(y, z)$, triangular inequality
- $d(x, y) = d(y, x)$, symmetry

Note that in contrast with standard metrics, an extended one might take infinite values. But as far as topology is concerned (convergence, continuity) extended metrics has the same properties than standard metrics which take only finite values. Since the notion of extended distance is more general than the one of standard metrics, hereinafter we simply use the terminology *distance* without making any precision on whether infinite values are permitted. This will ease up definitions as it able us to define the Hamming distance between words as a total function over pairs of words.

Any distance measure over elements of some set S, can be extended to a distance between elements of S and subsets of S. Where the distance $d(e, S')$ between an element $e \in S$ and a subset $S' \subseteq S$ is the infimum of the distances between e and elements $e' \in S'$. More formally $d(e, S') = \inf_{e' \in S'} d(e, e')$. As fas as we are concerned we will deal only with sets and distances such that the infimum in the previous definition is a minimum. That is there will always exist an element $e' \in S'$ such that $d(e, S') = d(e, e')$.

Distance measures are also called *similarity measures*, because a distance over elements of some set S is usually designed to account for how much two elements of S are similar. This approach has raised important studies of distances in scientific fields such as: semantics, molecular biology, physics, computer science, mathematics, chemistry and ecology; to only name some. A general method of studying similarity (or defining distances) between elements of some set S, is to specify some collection of basic operations (or transformations) over those elements. Such operations transform one element to another one (possibly the same). Depending on what is studied about the elements of S, we associate a cost to all these basic operations. These costs reflect how unlikely it is to consider that some element of S is similar to another. Then the distance between some element to another thus accounts for the minimal cost of the best way of transforming one element to another using only the basic operations. This kind of so defined distances are called *edit distances* and are formally defined in what follows.

A collection O of basic operations on elements of some set S is a set of functions from $S \to S$ such that:

- $id_S \in O$, And
- for all functions $f \in O$ and element $e \in S$: there is $f' \in O$ such that $f'(f(e)) = e$.

The last condition of the previous definition ensures that any application of some operation on some element can be reversed. A cost function for a set of basic operations O, on elements of S, is a function $cost : O \to \mathbb{R}_{\geqslant 0}$ such that:

- $cost(id_S) = 0$, And

- for all functions $f \neq id_S$, $f' \neq id_S$: if for some $e \in S$, $f'(f(e)) = e$ and $e \neq f(e)$ then $cost(f) = cost(f')$.

A cost function assigns equal cost to two operations that are different to the identity function but whose composition might leave some element unchanged. And the identity function is always assigned a zero cost. Now a transformation based on the set of operations O is any finite sequence of compositions of operations of O. The set of transformations O^* can then be inductively defined in the following way:

- $O \subseteq O^*$, And
- for all functions $f \in O$ and transformation $f' \in O^*$: $f \circ f' \in O^*$

Notice that a transformation can be identified with a sequence of operations or simply with a word over the alphabet O. Indeed the sequence $f_1, \cdots, f_n \in O$, where $n \in \mathbb{N}$, corresponds to the transformation $f_1 \circ f_2 \cdots \circ f_n$ which is also denoted by $f_1 \cdot f_2, \cdots f_n$. The empty sequence of transformations corresponding to the identity function. Thus if the set of operations O is assigned a cost function $cost : O \to \mathbb{R}_{\geqslant 0}$, the extension of the cost function to O^* is defined by: for all sequence $f_1, \cdots, f_n \in O$,

$$cost(f_1 \circ f_2 \cdots \circ f_n) = cost(f_1 \cdot f_2, \cdots f_n) = \sum_{1 \leqslant i \leqslant n} cost(f_i)$$

Finally, for some set S, the *edit distance* $d_O : S^2 \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ based on the basic operations $O : S^2$ is defined such that for elements $e_1 \in S$ and $e_2 \in S$, the distance $d_O(e_1, e_2)$ is the minimal cost over all transformations $\tau \in O^*$ satisfying $\tau(e_1) = e_2$, or the $d_O(e_1, e_2)$ is infinite when such transformation does not exist. More formally:

- if $\{\tau \in O^* \mid \tau(e_1) = e_2\} = \varnothing$ then $d_O(e_1, e_2) = \infty$
- otherwise $d_O(e_1, e_2) = \min_{\tau \in \{\tau \in O^* | t(e_1) = e_2\}} cost(\tau)$

We next show that edit distances defined by this method are sound. The separability property of d_O follows from the fact that $id_S \in O$ and the cost of the identity is $cost(id_S) = 0$. The symmetry is a direct consequence of the following property: for all elements $e_1, e_2 \in S$ and interger $n \in \mathbb{N}$, if there is a transformation $f = f_1 \cdots f_n$ such that $\tau(e_1) = e_2$, then there is also a transformation τ' which has the same cost than τ and $\tau'(e_2) = e_1$. Such a transformation τ' can inductively be defined by choosing for all $i \in [1, n]$, a function $f'_i \in O$ such that $f'_i(f_i \circ f_{i+1} \cdots \circ f_n(e)) = f_{i+1} \circ f_{i+2} \cdots \circ f_n(e)$; and then setting $\tau' = f'_n \circ f'_2 \cdots \circ f'_1$. This construction is allowed by the last conditions of the definitions of basic operations and cost functions. Finally the triangular inequality follows from the fact that d_O is defined as

a minimum cost over a set of transformations and the composition of two transformations is also a transformation.

Next we define edit distances for functions, words, trees and graphs. To do so, we will simply define a set of basic operations along with a cost function (we sometimes leave the identity function implicit). As it is classically done when studying edit distances, the cost function will always be equal to one for all basic operations. These distances will be considered later on in the context of property testing.

2.2.1 Distance between functions

We define here a distance over the set $S_1 \hookrightarrow S_2$ of functions from S_1 to S_2. This distance is called the *Hamming distance* over the set of functions as it corresponds to the *Hamming distance* of words; in the case when words are considered as functions (see. 2.2.2).

We consider for every element $e_1 \in S_1$ and element $e_2 \in S_2$, the operation $change_{e_1,e_2}$ consisting at changing the value of any function defined on e_1 and setting it to e_2. Formally, for all function $f : S_1 \hookrightarrow S_2$, the function $change_{e_1,e_2}(f) : S_1 \hookrightarrow S_2$ have the same domain than f and for all element $e \in dom(f)$:

- if $e = e_1$ then $change_{e_1,e_2}(f)(e) = e_2$
- otherwise $change_{e_1,e_2}(f)(e) = f(e)$

The Hamming distance between functions is based on the following set of basic operations which operates on functions of $S_1 \hookrightarrow S_2$:

$$O = \{change_{e_1,e_2} \mid e_1 \in S_1, e_2 \in S_2\} \cup \{id_{S_1 \hookrightarrow S_2}\}$$

And the cost function for the basic set of operations O associates 1 to all operations $change_{e_1,e_2}$. Note that two functions with different domains are at infinite Hamming distance from one to another. It is also straightforward that the Hamming distance between two functions of the same domain accounts for the (possibly infinite) number of elements on which the two functions differ. We use the notation d_h for the Hamming distance between functions.

Example 2.4. *let $n \in \mathbb{N}$, $f : \mathbb{N} \hookrightarrow \mathbb{N}$ and $f' : \mathbb{N} \hookrightarrow \mathbb{N}$ be the functions such that $dom(f) = dom(f') = [1, n]$ and for all $i \in [1, n]$, $f(i) = 0$ and $f'(i) = 1$. And $d_h(f, f') = n$.*

2.2.2 Distances between words

There are different notions of edit distances that can be defined over the set of words with alphabet Σ. Those distances are usually defined with four kinds of operations: *relabelling*, *insertion*, *deletion*, and *move*.

Hamming distance

The Hamming distance d_h between two words corresponds to the case when O contains only relabellings of positions. The relabelling $rel_{i,a} : \Sigma^* \to \Sigma^*$ of position $i \in \mathbb{N}$ to label $a \in \Sigma$ leaves unchanged words of size smaller than i, and for words $b_1 \cdots b_{i-1} \cdot b_i \cdot b_{i+1} \cdots b_n \in \Sigma^*$ of size $n \geqslant i$,

$$rel_{i,a}(b_1 \cdots b_{i-1} \cdot \mathbf{b_i} \cdot b_{i+1} \cdots b_n) = b_1 \cdots b_{i-1} \cdot \mathbf{a} \cdot b_{i+1} \cdots b_n$$

Notice that relabellings never change the size of a word, therefore there is no transformation which can transform some word to another one of different size. The cost function is constant and equal to 1 for all relabellings. Also, for labels $b \in \Sigma$ and words $w \in \Sigma^*$ satisfying $|w| < i$ or $w[i] = b$, we have $rel_{i,b}(rel_{i,a}(w)) = w$. Hence the Hamming distance is well defined. The Hamming distance was named after Richard Hamming [1950] who introduced it for *error detecting and error correcting codes*. Hamming distance is used in many disciplines including information theory, coding theory and cryptography.

Remark 2.1. *We recall that any word $w \in \Sigma^*$ can be seen as the function $f_w : [1, |w|] \to \Sigma$. We recall that the Hamming distance between words is related to the one of functions. Since for every words w and w': $d_h(w, w') = d_h(f_w, f_{w'})$.*

However Hamming distance compares only strings of the same size and therefore in many situations where also deletions, or substitutions is expected (for example in noisy channels), this distance isn't appropriate and other metrics as the *Levenshtein distance* are considered.

Levenshtein distance

Adding insertions and deletions to the edit operations permits to compare words of different sizes and corresponds to the Levenshtein distance. The insertion $ins_{i,a}$ of label $a \in \Sigma$ after position $i \in \mathbb{N}$ has no effect on words of size less than i, and for $w = b_1 \cdots \mathbf{b_i} \cdot \mathbf{b_{i+1}} \cdots b_n \in \Sigma^*$,

$$ins_{i,a}(w) = b_1 \cdots \cdot b_i \cdot \mathbf{a} \cdot b_{i+1} \cdots b_n$$

The deletion of position i transforms w into $del_i(w) = b_1 \cdots \mathbf{b_{i-1}} \cdot \mathbf{b_{i+1}} \cdots b_n$, and it leaves unchanged words of length less than i. As for the Hamming distance the cost associated to all operations is 1. It is also straightforward that for words w satisfying $w[i+1] = a$, or of length less than i, we have $del_{i+1}(ins_{i,a}(w)) = ins_{i,a}(del_{i+1}(w))$. Therefore the distance is well defined.

While in some cases (communication channels, typing errors) insertions and deletions may be seen as basic operations, thus providing interests in studying the Levenshtein distance; in other applications (for example whenever strings are considered as linked lists), moving a substring from one

place to another become with lower cost and can therefore be considered as basic. Therefore the edit distance with moves, considered by Cormode and Muthukrishnan [2007]; Shapira and Storer [2007], become appropriate for example in applications involving evolutionary changes in biological sequences.

Edit distance with moves

The distance obtained by adding move operations to the Levenshtein distance is simply called *The edit distance with moves*. The move operation $mov_{i,j,l}$ of a substring of length l from position i to position j, has no effect on words of size less than $i+l$. Furthermore for all words w of size $n \geqslant i+l$, $mov_{i,j,l}(w)$ is defined as:

- w, if $i \leqslant j \leqslant i+l$
- $w[1, j-1] \cdot \mathbf{w[i,i+l]} \cdot w[j, i-1] \cdot w[i+l+1, n]$, for $j < i$ and
- $w[1, i-1] \cdot w[i+l+1, j-1] \cdot \mathbf{w[i,i+l]} \cdot w[j, n]$ for $j > i+l$

As the other operations, each move operation can be reversed i.e. there is i', j' satisfying $mov_{i',j',l}(mov_{i,j,l}(w)) = w$, so assigning cost 1 to all edit operations, we clearly obtain a valid edit distance.

Through the rest of this thesis d_h, d_l and d_m denote the Hamming distance, the Levenshtein distance and the edit distance with move respectively. We next give example of pair of strings with the distance between the two strings in those pairs. It is straightforward from their definitions that each distance in the previous order is weaker than its precedents. That is for all pair of words w, w', $d_h(w, w') \geqslant d_l(w, w') \geqslant d_m(w, w')$.

Example 2.5.

$$w_1 = kitten, w_2 = sitting,\ d_l(w_1, w_2) = 3 \text{ and } d_H(w_1, w_2) = \infty$$
$$w_3 = 0110\downarrow 001111001101, w_4 = 0110111100001100,\ d_m(w_3, w_4) = 2$$

2.2.3 Distances between trees

Tree distances are used in many area of science as a notion of similarity. As Bille [2005] already noticed in his survey on tree edit distances, "the problem of comparing trees occurs in several diverse areas such as computational biology, structured text databases, image analysis, automatic theorem proving, and compiler optimization". We hereinafter define trees edit distances as generalisations of words edit distances; with insertions, deletions and relabellings as basic operations.

Let Σ be an alphabet and Σ^r be a ranked alphabet over Σ i.e. a pair consisting in Σ along with an arity function. For ranked trees, generalizing the edit operations previously defined for words as a collection of relabellings, insertions, deletions and moves seems unnatural. The main reason is that

defining edit operations as functions over the set $\mathcal{T}_\Sigma^r$ of ranked trees over Σ^r must be done respecting its arirty function. In particular when deletions or insertions of nodes are considered, the natural results of such operations would be an unranked tree over Σ and not an element of $\mathcal{T}_\Sigma^r$. This is illustrated in the following example.

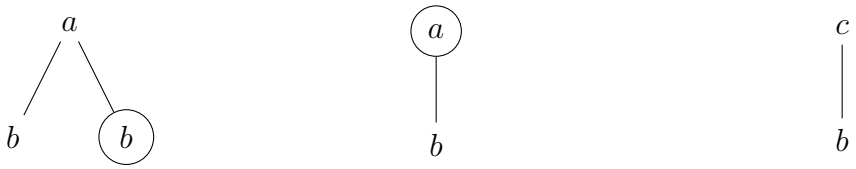


Figure 2.8: The leftmost and rightmost trees are ranked trees over the alphabet $\Sigma^r = \{a^2, c^1, b^0\}$. In the middle is represented an unranked tree over Σ

In this example, we would naturally like to be able to transform the leftmost tree into the rightmost one using insertions, deletions and relabellings of nodes. However if the basic operations could only yield a valid ranked tree over $\{a^2, c^1, b^0\}$, then it is not hard to see that this would be impossible. So in order to have finite distance between all ranked trees, deletions and insertions must be defined as operations in the set of unranked trees over the alphabet $\{a, b, c\}$. Therefore in what follows, trees edit distances will be defined on the set $\mathcal{T}_\Sigma$ of unranked trees. However this is not a limiting issue as we already mentioned that all ranked trees can be considered unranked when ignoring labels arities. Hence in this example, we can delete the circled b-node at the leftmost tree, and then relabel the root of the tree in the middle to obtain the rightmost ranked tree. Now the remaining issue is how to define deletion of nodes that have children. There are two ways of tackling this issue. In the classical way, the children of the deleted node become children of its parent. The other way consists at forbidding such deletion and allowing only deletion of leafs (*strong edit distance* [Selkow, 1977]). Below we detail the set of basic operations.

Standard edit distance In the classical edit distance the basic operations are relabellings, insertions and deletions. Hereinafter we define these operations. Let Σ be some alphabet. For all elements $w \in \mathbb{N}^*$ and labels $a \in \Sigma$, the operations $rel_{w,a} : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ applied to a tree t changes the label of the node w, if $w \in nod_t$; otherwise it leaves t unchanged. Hence, For all trees t, $rel_{w,a}(\mathrm{t})$ can be inductively defined in the following way:

- if $w \notin nod_t$ then $rel_{w,a}(\mathrm{t}) = \mathrm{t}$, else
- $rel_{w,a}(\mathrm{t}) = \mathrm{C}_w^{\mathrm{t}}[\mathbf{a}(\mathrm{t}_1, \cdots, \mathrm{t}_n)]$, where $t_{|w} = b(\mathrm{t}_1, \cdots, \mathrm{t}_n)$

We recall that $\mathrm{C}_w^{\mathrm{t}}$ is the context of w in t. Insertion operators, denoted by $ins_{w,i,j,a} : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$, are defined for all words $w \in \mathbb{N}^*$, integers $i \leqslant j$ and

labels $a \in \Sigma$. When applied to a tree t, the operation $ins_{w,i,j,a} : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ changes t only if w, $w \cdot i$ and $w \cdot j$ are nodes of t. And in such case it inserts a node labelled a as a children of w and it makes the sequence of the previous children $w \cdot i, \cdots, w \cdot j$, children of the new node. Hence formally for all trees $\mathrm{t} \in \mathcal{T}_\Sigma$, with $t_{|w} = b(\mathrm{t}_1, \cdots, \mathrm{t}_n)$:

- if $w \cdot j \notin nod_t$ then $ins_{w,a}(\mathrm{t}) = \mathrm{t}$, else
- $ins_{w,a}(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[b(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, a(t_i, \cdots, t_j), \mathrm{t}_{j+1}, \cdots, t_n)]$

Now the last basic operations of the edit distance are deletions. Deleting a non-root node w of some tree w, will replace this node by the edge of its children. Hence for all non-empty words $w \in \mathbb{N}^*$, the operation del_w is defined such that:

- if $w \notin nod_t$ then $del_w(\mathrm{t}) = \mathrm{t}$, else
- $del_w(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[\mathrm{t}_1 \cdots \mathrm{t}_n]$, where $t_{|w} = b(\mathrm{t}_1, \cdots, \mathrm{t}_n)$

It is easy to verify that the effect of any of these basic tree operations can be reversed using another operation. Insertions for examples are reversed with deletions and reballings by other relabellings. So when setting the cost function to be 1 for all operations we obtain the classically called edit distance between trees.

Edit distance with moves When trees are represented as lists, it is natural to consider that moving the subtree rooted by some node to another place is costless and then in this case such operations can be considered as basic operations along with relabellings, deletions, and insertions. These move operations $mov_{w_1,w_2} : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ are defined for pairs $w_1, w_2 \in \mathbb{N}^*$ of nonempty words satisfying that none of w_1 or w_2 is a prefix of the other. Applied to a tree t, the operation mov_{w_1,w_2} moves the subtree rooted by w_1 and places it after the node w_2. We below define the tree resulting from the application of the move operation mov_{w_1,w_2} to a tree t.

- if $w_1 = w \cdot i$, $w_2 = w \cdot j$, $i < j$ and $t_{|w} = a(\mathrm{t}_1, \cdots, \mathrm{t}_n)$ then:
$$mov_{w_1,w_2}(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[a(\mathrm{t}_1, \cdots, \mathrm{t}_{i-1}, \mathrm{t}_{i+1}, \cdots, \mathrm{t}_j, \mathrm{t}_i, \mathrm{t}_{j+1}, \cdots, \mathrm{t}_n)]$$
- else if $w_1 = w \cdot i$, $w_2 = w \cdot j$, $i > j$ and $t_{|w} = a(\mathrm{t}_1, \cdots, \mathrm{t}_n)$ then:
$$mov_{w_1,w_2}(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[a(\mathrm{t}_1, \cdots, \mathrm{t}_j, \mathrm{t}_i, \mathrm{t}_{j+1}, \cdots, \mathrm{t}_{i-1}, \mathrm{t}_{i+1}, \cdots, \mathrm{t}_n)]$$
- else, $mov_{w_1,w_2}(\mathrm{t}) = \mathrm{C}^{\mathrm{t}_{w_1}}_{w_2}[\mathrm{t}_{|w_2} \cdot \mathrm{t}_{|w_1}]$, where $\mathrm{t}_{w_1} = \mathrm{C}^{\mathrm{t}}_{w_1}[()]$

The formal definition is a little harsh to read, but we have to separate the case when w_1 and w_2 have the same parent and the case they do not. The reason is that in the case they have the same parent, deleting the tree rooted by w_1 in t changes its domain in a way that the node w_2 in the new tree is no longer the one after which we want to insert the subtree $t_{|w}$. It is simple to see that this is not the case when w_1 and w_2 does not share the same parent.

Strong edit distance Notice that the edit distance may not correctly differentiate structural differences of trees. The main reason is that with insertions or deletions (which operation cost only 1), we are able to change the parent of an important sequence of children. In situations where a better notion of similarity is required, this could be a limiting issue. Hence to tackle this issue, we may restrict deletions or insertions only to leafs. Hence in the strong edit distance, the relabelling operations are the same compared to the standard edit distance. In contrast, for the strong edit distance, deletions and insertions operations differs with the standard case and are defined for all non-empty words $w \in \mathbb{N}^*$. Insertions $ins_{w,a}$, where $a \in \Sigma$ is label, modify every tree t which contains a non-root node w by inserting a leaf labelled a after w.

- if $w \notin nod_t$, $w \neq \varepsilon$ then $ins_{w,a}(\mathrm{t}) = \mathrm{t}$, else
- $ins_{w,a}(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[\mathrm{t}_{|w} \cdot \mathbf{a}()]$

Deletions del_w modifies a tree t only if the non-root node w is a leaf of t, and in such case using substitutions of nodes with hedges, we have:

- if w is not a leaf of t then $del_w(\mathrm{t}) = \mathrm{t}$, else
- $del_w(\mathrm{t}) = \mathrm{C}^{\mathrm{t}}_w[()]$

We will use notations d_{stand}, d_{move}, and d_{strong} to denote the standard edit distance, the edit distance with move and the strong edit distance respectively. It is straightforward that for every trees t, t$'$, we have the inequalities:

$$d_{move}(t, t') \leqslant d_{stand}(t, t') \leqslant d_{strong}(t, t')$$

These distances are later discussed in the context of property testing of trees. What distance is most appropriate for an application is really a matter related to the specific application we are considering and how we want to account similarities or dissimilarities of the involved trees. So it is really a design problem. But what these inequalities tells us is: the distances that most separate trees are the strong edit distance, the standard edit distance and the edit distance with moves in this order.

2.2.4 Distances between graphs

The natural operations in labelled simple graphs would be to add, delete or relabel edges. Thus the edit distance we define on graphs consists only on these operations.

Let V be a set and Σ be an alphabet. In the following, we defined the basic operations on the set of Σ-labelled graphs with V as the set of nodes. For all graphs $G = (V, E)$, pairs of nodes $v_1 \in V$, $v_2 \in V$ and labels $a \in \Sigma$,

when the relabelling $rel_{v_1,v_2,a}$ of the edge (v_1, v_2) is applied to G, it sets the label of (v_1, v_2) to a. That is $rel_{v_1,v_2,a}(G)$ is the graph $G_1 = (V, E)$ satisfying: for all $e \in E$

- if $e = (v_1, v_2)$ then $lab_{G_1}(e) = a$, else
- $lab_{G_1}(e) = lab_G(e)$

The insertion, $ins_{v_1,v_2,a}(G)$, of the edge (v_1, v_2) with label a is the graph $G_2 = (V, E \cup \{(v_1, v_2)\})$ satisfying: for all $e \in E \cup \{(v_1, v_2)\}$

- if $(v_1, v_2) \in E$ then $lab_{G_1}(e) = lab_G(e)$
- else if $e = (v_1, v_2)$ then $lab_{G_1}(e) = a$, else
- $lab_{G_2}(e) = lab_G(e)$

And finally the deletion, $ins_{v_1,v_2}(G)$, of the edge (v_1, v_2) from the edges of G is the graph $G_3 = (V, E \backslash \{(v_1, v_2)\})$ which satisfies: for all $e \in E \backslash \{(v_1, v_2)\}$

- $lab_{G_3}(e) = lab_G(e)$

The cost of all these operations are 1 and it is clear that every operation can be reversed. In particular relabellings are reversed using other relabellings and insertions reverses deletions. So the edit distance defined by these operations is sound. Through the rest of these thesis we will denote this distance by d_e.

2.3 Probabilities

Probability theory was developed during the last century and have many applications in modern science, specially in computer science. For example probability theory is used in biometry, quantum physics, economics. Dekking, Kraaikamp, Lopuhaä, and Meester [2010] provide "a modern introduction to probability and statistics". The objective of probability theory can be understood as the analysis of random or unpredictable phenomena, thus it is a good tool to model uncertainty in decision making [Tran, Peng, Li, Diao, and Liu, 2010; Sarma, Benjelloun, Halevy, Nabar, and Widom, 2009]. Whether probability theory is the appropriate tool for modelling human decision making or the world phenomena remains an open philosophical question but notice that using probability theory engineers, scientists, business persons, manufactures and others have developed methods to efficiently solve many problems. Thus if we are specially concerned with practical issues, probability theory has proven to be a good modelling tool. Usually the theory of probability is presented with a slight distinction depending on whether the *sample espace* is finite, countable or it is of cardinality greater than the

continuum. However, during this thesis we will mostly be concerned with probability over finite sets and therefore we give in this section all definitions on finite sample sets.

A *sample space* Ω is simply a set. We will deal only with finite sample spaces. A *probability distribution or probability function* over a finite sample space Ω is a real valued function $p : \Omega \to [0, 1]$ satisfying $\sum_{e \in \Omega} p(e) = 1$. The distribution p is said *uniform* if $p(e) = 1/|\Omega|$, for all elements $e \in \Omega$. Probability distributions over finite sets are also called *categorical distributions* and they are generalisations of Bernoulli distributions.

An example of a random experiment is a coin toss where the set of possible outcomes are 'heads' or 'tails'. If the coin is not biased, then we say that the experiment follows the uniform distribution on the set $\{head, tail\}$. To describe the outcomes of random experiments, *random variables* are used. Note that the results of a random experiment can have any kind of outcome. To be able to make computation (ex. additions) with random variables, usually a number is associated its outcomes. Thus a *random variable* is usually defined as a real valued function that associates a number to every element of the sample space on which the experiment is performed. Therefore a *random variable* $\mathrm{X} : \Omega \to \mathbb{R}$ is any real valued function from Ω to $\mathbb{R}$. Hence, if p is a probability distribution over the finite sample space Ω, we say that the probability for the random variable X to take some value $x \in \mathbb{R}$ is: $p(X = x) = \sum_{e \in X^{-1}(x)} p(e)$, where $X^{-1}(x) = \{e \in \Omega \mid X(e) = x\}$. However, since we consider only finite sample sets, and aren't usually concerned with computation of random variables, then without lost of generality, we consider that random variables are injective functions and that their values are elements of Ω. Indeed in such case, for all values x taken by the random variable, there is only one element e of the sample space such that $X(e) = x$, and then $p(x) = p(e)$. Therefore we will write $p(X = e)$ for $p(X = x)$, where $X(e) = x$. Note that in cases when a random variable is not injective, it transforms the sample space into a new sample space that is most appropriate to the experiment for which it models the randomness of the outcomes. An example is when we uniformly select an integer in the interval $[1, 10]$ and are only interested on whether the outcome is even or not. Thus in our case, random variables collapse with probability distributions underling random experiments for which they represent the outcomes.

Probability distributions are used to characterise the randomness of phenomena. And experiments following some distribution can be *simulated* or mimicked by computers *(pseudo) random generator*. A process that simulates p is called a *simulation* of p. The result of a simulation of p is called a random choice according to p and is usually denoted by $X \sim p$. By this we mean X is assigned a random value according to the distribution p. This way we obtain a random variable for p.

Let S' be a collection of distributions over a sample set Ω indexed by elements of some set S. That is $S' = \{p_e | e \in S\}$. A random function

$rdm : S \to \Omega$ for S' is any function (process) which inputs elements $e \in S$ and returns a random choice $X \sim p_e$. Hence the value of a random function is not known a priori, but we only know the distribution of follows the random choice. In this thesis we use random functions to design random accesses to objects.

2.4 Randomized Algorithms

Algorithms are designed to finitely describe tasks or computations. Those tasks are performed either by humans or computers. There are many ways to specify algorithms. On the high level, algorithms are described with human languages, pseudo-code; and more formally they are described by computer languages such as C, Java, Caml etc... On the very low level or more abstractly, as descriptions of algorithms we can use Turing state machines or equivalently lambda terms of the λ-calculus. Turing machines and Random Access Machines (RAM) are formal models for computers; and we see a deterministic algorithm just as a complete specification of the rules (or basic computations/machines) to apply from any configuration with specifications of following configuration, termination and answers. Hence a deterministic algorithm is determined by a state machine such as Turing machines or deterministic automata, where for each input the execution path is unique. Each state corresponds to a configuration in the machine and also to a basic instruction to perform. And depending on the result of this instruction, the next state on which to move on is exactly determined. Therefore algorithms might be designed just as labelled graphs or decision trees (this is the case for all algorithms discussed in this thesis). When from some state (or node), we can move to many different states, and this for the same result of the performed instruction, we say that the algorithm is non-deterministic. In the execution of non-deterministic algorithms on some input, we are sometimes led to a choice of the path to follow with the only guaranty that some path of these choices yield a correct result. Furthermore, when algorithms are allowed to use a random source or the result of a coin toss to help deciding which path to follow, they are called *randomized algorithms*.

The efficiency of an algorithm over all its inputs is usually measured in terms of the maximum time a computer (or machine) would take, or else the maximum space it would use, before termination. Notice that this is dependent on the machine. In particular when we use descriptions of algorithms by states machines this is clearly dependent on the unit time to perform the basic instructions/computations at its state (or node). To get rid of this dependence, the big O notation was introduced in complexity theory. Hence, the time complexity of an algorithm over all its input is a big O of the longest path form the initial state to a terminating state. Where, for $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$, we have $f = O(g)$ iff. there exist $M \in \mathbb{R}$ and $n_0 \in \mathbb{N}$ such

that: for all $n \leqslant n_0$, $|f(n)| < M \cdot |g(n)|$. We write $f = \Theta(g)$ for $f = O(g)$ and $g = O(f)$. Thus complexity theory is concerned about classifying problems depending on the complexity of the most efficient algorithms solving computation tasks. This study leads to complexity classes such as P (for deterministic algorithms), NP (for non-deterministic algorithms), RP and BPP (for randomized algorithms). For an introduction to complexity theory, the reader is referred to the book "Computational complexity" by Papadimitriou [1994].

As an example of algorithm, consider the task of computing the Hamming distance between two words. Such algorithms input two words w, w' and answer the distance between w, w'. Computing $d_h(w, w')$ can be done in time $O(\min(|w|, |w'|))$. It suffices only to read positions of the two words starting from the beginning and stopping after the end of the shortest word. The same task can be considered for the Levenshtein distance and the edit distance with moves. Using dynamic programming Landau and Vishkin [1986] and Gusfield [1997] obtained an upper bound of $O(|w||w'|)$ for computing $d_l(w, w')$. Substantial improvements of this bound remains an open problem but progress were obtain by Masek and Paterson [1980], using the Four Russians method to get the best known bound of $O(|w||w'|/\log(|w|))$. Further improvement to this question is obtained by Andoni and Krauthgamer [2007], who proved hardness of the computation of $d_l(w, w')$ with regard to $d_h(w, w')$. The main reason of the hardness of computing d_l comes from the fact that insertions and deletions makes the problem of finding good alignments harder. Thus we would expect the computation of $d_m(w, w')$ to be even harder and indeed it was proven by Shapira and Storer [2007] that this problem is NP-complete. The proof of the NP-completeness relies on a reduction to the bin-packing problem, the later being known to be NP-complete in the strong sense.

Further we give some details about the behaviours of randomized algorithms. Since a randomized algorithm may base its decisions on a coin toss; then it might give an incorrect answer with some probability. Randomized algorithms that never err with their answers are called *Las Vegas* algorithms and those that may err with non-null probability are called *Monte Carlo* algorithms. A class of Monte Carlo algorithms with much interest is the one with only algorithms whose error probability can be bounded by a constant smaller than 1/2 (BPP). Because the error of such algorithms can be made as small as needed just by iterating them a polynomial time. Hereinafter we will only use such algorithms. For a detailed introduction to randomized algorithms the reader is referred to: "Randomized Algorithms", a book by Motwani and Raghavan [1997]. Notice that the outcome of any random algorithm may be seen as a random variable depending on the probability space of the algorithm random source.

Besides the fact that it could be argued that any computation made by a physical device is indeed probabilistic, random algorithms had led to more

efficient algorithms; when compared to deterministic ones (ex: primary test, quick sort). This is sufficient to say that randomized algorithms are useful, and it also leads to the question of knowing for some specific problem whether randomized algorithms help gain efficiency when compared to deterministic ones. Using game theoretic techniques and in particular Neumann's Minimax Theorem, Yao [1977] provided an answer to this question by proving a lower bound on the performance of randomized algorithms.

2.4.1 Yao's Mininimax Principle

Let Π be a problem. With the following requirement, Yao's principle applies to any kind of complexity measure C of algorithms solving Π. The requirement is that the number of distinct inputs of a fixed size is finite, and so is the number of deterministic algorithms solving Π. let $\mathcal{I}$ be the set of inputs with a fixed size, and $\mathcal{A}$ be the set of deterministic algorithms solving Π. Notice that in this case all randomized algorithms can be reduced to a probability distribution over $\mathcal{A}$. Let p be a probability distribution over $\mathcal{I}$ and q a probability distribution over $\mathcal{A}$. For random inputs I_p selected according to p, we denote by $C_\nu(I_p, A)$ the complexity of any deterministic algorithm $A \in \mathcal{A}$ which errs on I_p with probability at most ν. For inputs I and random choices (with distribution q) of some randomized algorithm A_q which errs on I with probability at most ν; $C_\nu(I, A_q)$ denotes the complexity of A_q on I. Yao's minimax principal can be stated as follows:

Theorem 2.1 ([Yao, 1977]). *For all distributions p over $\mathcal{I}$ and q over $\mathcal{A}$ and $\nu \in [0, 1/2]$,*

$$\frac{1}{2}(\min_{A \in \mathcal{A}} \mathbf{E}[C_{2\nu}(I_p, A)]) \leqslant \min_{I \in \mathcal{I}} \mathbf{E}[C_\nu(I, A_q)]$$

Where $\mathbf{E}$ is the expectation.

Hence Yao's minimax theorem says that the expected complexity of any randomized algorithm on its worst input can not be better than the expected complexity of the best deterministic algorithm on the worst-case distribution over the inputs. Thus to prove a lower bound on randomized algorithms, we might just find a distribution on the inputs and prove a lower bound on the expected complexity of deterministic algorithms. In this thesis, this principle will be our main tool for proving lower bounds for randomized algorithms. An example where we use Yao's principle can be found in Section 3.3.2.

3 Exact and approximate model checking: related work

Contents

In the previous chapter we have introduced the main objects that concern us in this thesis. Although that chapter could be skipped for those who are familiar with the notions of words, trees or graphs; we believe it clarifies the subject of our studies which we detail in this chapter. Because as W.V.O. Quine once said "confusion of signs and objects is original sin coeval with the word", but "language is conceived in sin and science is its redemption". Now on the road to redemption, one wants to be able to talk about those objects and study some of their properties. Therefore in this chapter we begin with showing how words, trees and graphs can be seen as objects of the same kind: **relational structures**. Relational structures can be used to model any kind of databases and our interest is to query databases and

study their properties. We then provide an brief introduction to formal languages used in computer science to talk about relational structures and study their properties: **logics**. Next we will formally define the notion of property for objects and then give known results in the study of properties: **model checking**. We will present exact study of properties and an approximate version based on distance measures: called **property testing** in the literature. We finish with the notion of **automata** as another way of representing properties and discuss the standard data format XML. Studying XML (databases), is the main subject of this thesis and it will be done in the context of property testing in the last chapter.

3.1 Logic

As previously noticed in the introduction of this chapter, we are interested to query information about the objects we study (i.e words, trees, graphs) and decide whether or not they satisfy some property; as it is usually done with databases. Thus in this section we show how these objects (i.e. trees, words, graphs), can be identified with relational structures. Before getting to that let us clarify what a relational structure is.

3.1.1 Relational σ-structures

Science in general and computer science in particular is full of structures. Example of structures are rings and fields in mathematics and crystal structures in crystallography. In logic this term means a set along with finitary operations and relations. This view also applies to data structures in computer science; and hence permits formal analysis of the complexities of algorithms, using tools from mathematical logic. In order to use logic for studying properties, relational structures are defined for some vocabulary σ. Vocabulary σ which is the set of names of all relations in the structures along with their arities.

Vocabularies/Signatures A *relational vocabulary or signature* is a finite set $\sigma = \{r_1, \cdots, r_m\}$ of relation symbols with specified arities. For $k \in \mathbb{N}$, σ_k denotes the subset of σ of relations with arity k. Note that a relation symbol with its arity can be seen as a pair of a relation name and an integer. Thus vocabularies will be defined as sets of pairs of relation name with an integer for the relation arity. For example $\{(label, 1), (<, 2)\}$ is a vocabulary with two relations: one relation named 'label' with arity 1 and another relation of name $<$ and of arity 2. However for a more succinct notation we will denote such set by $\{label^1, <^2\}$.

Structures Let $\sigma = \{r_1, \cdots, r_m\}$ be a relational vocabulary.

A (relational) σ-structure Γ is a pair of some none empty set $dom(\Gamma)$ along with a set $rel(\Gamma) = \{r_1^\Gamma, \cdots, r_m^\Gamma\}$ of relations over $dom(\Gamma)$ such that: $arity(r_i) = arity(r_i^\Gamma)$, $1 \leqslant i \leqslant m$. The set $dom(\Gamma)$ is called the *domain* or *universe* of Γ. A morphism $\theta : \Gamma_1 \to \Gamma_2$ is a structure-preserving mapping from σ-structure Γ_1 to σ-structure Γ_2, i.e θ is a function from $dom(\Gamma_1)$ to $dom(\Gamma_2)$ and for all integer $k \in \mathbb{N}$, k-ary relation $r \in \sigma$ and tuple of elements $(e_1, \cdots, e_k) \in dom(\Gamma_1)$, it holds that: $r^{\Gamma_1}(e_1, \cdots, e_k)$ if and only if $r^{\Gamma_2}(\theta(e_1), \cdots, \theta(e_k))$. Γ_1 is isomorphic to Γ_2 if θ is bijective and its inverse θ^{-1} is also a morphism from Γ_2 to Γ_1. The degree of an element $e \in dom(\Gamma)$, in the relation $r \in \sigma$ of arity k and at a position $i < k$, is the number of tuples in r^Γ which contains a in the i^{th} position. The degree of the structure Γ is the maximal degree of all elements of its domain, in all the relations of σ and all positions. The degree of a structure have been very useful in the study of structures as for example, better algorithms are obtained for structures with bounded degree (see Section 3.3.1). When there is no confusing we use the same notation for the relation in a vocabulary σ and its corresponding relation in all σ-structures. This ease up notations by removing exponents on structures relations (i.e. we use r for r^Γ).

The notion of isomorphism also plays an important role in the study of structures. In fact the 'isomorphic to' (meta) relation is an equivalent relation in the class of structures; and since many properties of structures are closed under isomorphism, then algorithms designed to decide such properties should behave the same on all input isomorphic structures. So, carrying on tradition, we consider only properties closed under isomorphism and thus algorithms discussed in this thesis depend only on the relations of the input structure but not on any kind of particularity of its domain. Next we view words, graphs and trees as structures.

Words over the alphabet Σ define relational structures over the vocabulary:

$$\sigma_{words} = \{lab_a^1 \mid a \in \Sigma\} \uplus \{<^2, succ^2, start^1\}$$

Where for a word $w \in \Sigma^*$, the domain of the underlying structure S_w is $dom(w)$ and the relations named by this vocabulary are interpreted as follows: $start^w = \{0\}$, $succ^w = \{(i, i+1) \mid 0 \leqslant i < |w|\}$ and lab_a^w is the set of positions of w labeled by a.

The vocabulary for trees over the alphabet Σ is:

$$\sigma_{trees} = \{lab_a^1 \mid a \in \Sigma\} \uplus \{fc^2, ns^2, parent^2, desc^2, root^1, anc^2, <^2\}$$

The structure S_t defined by a tree $\mathrm{t} \in \mathcal{T}_\Sigma$ have domain $dom(\mathrm{t})$ and the relations named by *fc*, *ns*, *parent* *desc*, *anc*, $<$ are interpreted as the first-child, next-sibling, parent, descendant, ancestor and preorder relations of the tree domain of t (see Section 2.1.3). The *root* relation is interpreted as $root^\mathrm{t} = \{\varepsilon\}$. Note that not all these relations are necessary to define words structures under isomorphic, as one can restrict the vocabulary only to the

labels with first-child and next-sibling relations. However we add the other relations because for property testing (see Chapter 6) random objects needs efficient access to these relations.

Note that graphs were already defined in Section 2.1.4 as a set of nodes (its domain) and an edge relation over this set of nodes. Thus any graph $G = (V, E)$ naturally translates into a structure S_G over the vocabulary:

$$\sigma_{graphs} = \{node^1, edge^2\}$$

where the relations are $node^G = V$ and $edge^G = E$.

The structures S_w, S_t, S_G so defined and any other structures isomorphic to them are called **representations** of words, trees and graphs respectively. It is important to notice at this point that any word, tree or graph defines a single class of isomorphic representations. Hence hereinafter algorithms that decide properties of words, trees or languages will input any of their representations and are required to behave the same on all representations of the same object. However we still need to specify formal languages that allow us to talk about objects. This is done by using the vocabulary of their representations.

3.1.2 Logical languages: FO, MSO

Formal languages used to query information about objects or relational structures are defined based on some logic. Yet what is logic? To provide an informal answer to this question we paraphrase Crossley [2011], who follows the ancient Greek philosopher Aristotle: "logic is the correct rearranging of facts to find the information that we want". Getting down to a formal level, modern logic has two sides: **syntax** and **semantic**. In the syntactic side one specifies rules to construct valid formulas for a language. And in the semantic side one specifies the meaning of those formulas. Many logics have been studied for relational structures: Datalog, Modal and Fixed-point logics for example. First-order (**FO**) and Monadic second-order (**MSO**) logics are standard for expressing properties of relational structures in finite model theory. A detailed introduction to finite model theory, and more results about the field can be found in [Libkin, 2004; Grädel, Kolaitis, Libkin, Marx, Spencer, Vardi, Venema, and Weinstein, 2007]. Here we give syntaxes and semantics of these two logics on relational structures.

First-order logic

The valid FO formulas, for a relational vocabulary σ and a set of variable $\mathcal{V}$, can be defined with the following grammar:

$$\phi = r(x_1, \cdots, x_k) \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \neg\phi \mid \exists x.\phi \mid \forall x.\phi \mid x = x'$$

where r is a symbol of σ of arity $k \in \mathbb{N}$, $x_i \in \mathcal{V}$ for all integers $i \leqslant k$ and $x, x' \in \mathcal{V}$. The variables of a formula ϕ which are out of any scope of any quantifier are called *free*. Non-free variables are called *bounded*. A formula without free variables is called *closed* or a *sentence*. The set of FO formulas over a relational vocabulary σ is denoted by $\mathrm{FO}[\sigma]$. A $\mathrm{FO}[\sigma]$ formula ϕ with k free variables will also be denoted by $\phi(x_1, \cdots, x_k)$. An assignment of the k free variables $x_1, \cdots, x_k$ into elements of some set S, is an element of S^k. Next assignments of variables will be denoted by $\overrightarrow{a} = (a_1, \cdots, a_k)$, where for all $i \leqslant k$ the variable x_i is assigned a_i.

We then define the semantic of FO formulas over relational structures using Alfred Tarski's [Tarski, 1995] method for defining the interpretation (semantic) of any deductive science. A $\mathrm{FO}[\sigma]$ formula ϕ, on the relational vocabulary σ, is interpreted over a structure Γ using an assignment $\overrightarrow{a}$ of all the free variables of ϕ into elements of the domain of Γ. Hence the semantics of $\mathrm{FO}[\sigma]$-formulas are below defined using the satisfiability symbol $\Gamma, \overrightarrow{a} \models \phi$.

$$
\begin{array}{lll}
\Gamma, \overrightarrow{a} \models r(x_1, \cdots, x_k) & \text{iff.} & r(a_1, \cdots, a_k) \\
\Gamma, \overrightarrow{a} \models \phi \wedge \phi' & \text{iff.} & \Gamma, \overrightarrow{a} \models \phi \text{ and } \Gamma, \overrightarrow{a} \models \phi' \\
\Gamma, \overrightarrow{a} \models \phi \vee \phi' & \text{iff.} & \Gamma, \overrightarrow{a} \models \phi \text{ or } \Gamma, \overrightarrow{a} \models \phi' \\
\Gamma, \overrightarrow{a} \models \neg\phi & \text{iff.} & \Gamma, \overrightarrow{a} \not\models \phi \\
\Gamma, \overrightarrow{a} \models \exists x.\phi(x_1, \cdots, x_k, x) & \text{iff.} & \exists b \in dom(\Gamma) \text{ satisfying } \phi(a_1, \cdots, a_k, b) \\
\Gamma, \overrightarrow{a} \models \forall x.\phi(x_1, \cdots, x_k, x) & \text{iff.} & \text{for all } b \in dom(\Gamma), \phi(a_1, \cdots, a_k, b)
\end{array}
$$

Different signatures have been considered for the FO logic of relational structures. In particular, for trees over alphabet Σ, Libkin [2004] considers a signature containing only the label relations $((lab_a)_{a\in\Sigma})$ with the child $(child)$ and next-sibling (ns) relations. FO logic has been intensively studied in computer science and in mathematics. And in general one classical result is that FO logic can not express the transitive closure of relations [Libkin, 2004]. By this we mean for some relations $r \in \sigma$, it could be the case that there exits no $\mathrm{FO}[\sigma]$ formula $\phi(x_1, \cdots, x_k)$ such: $\Gamma, \overrightarrow{a} \models \phi(x_1, \cdots, x_k)$ iff. $r^*(a_1, \cdots, a_k)$. This result is obtained using a correspondence between the expressive power of FO, with the existence of a winning strategy of Ehrenfeucht-Fraïssé games [Fraïsé, 1984]. This limitation opens the door for using more powerful logics to express properties of objects.

Monadic second-order logic

The MSO logic is an extension of the FO logic that allows quantified second-order variables which are interpreted as sets over the domain of any relational structure. Hence the set $\mathcal{V}$ of variables is extended with second-order variables which are classically denoted by upper-case letters, say X. For vocabulary σ, the valid $\mathrm{MSO}[\sigma]$ formulas are defined by the following grammar:

$$
\begin{array}{rcl}
\phi & = & r(x_1, \cdots, x_k) \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \neg\phi \mid \exists x.\phi \mid \forall x.\phi \mid x = x' \mid \\
& & \exists X.\phi \mid \forall X.\phi \mid x \in X
\end{array}
$$

where $r \in \sigma_k$ and $x, x', x_1, \cdots, x_k, X \in \mathcal{V}$. The semantic of MSO is defined by extending the semantics of FO. The difference is essentially with assignments of second-order variables to subsets of the relational structure domain. For assignments of second order variables, we use upper-cased symbols $\overrightarrow{A}$. Thus the satisfiability symbol for MSO is obtained by extending the one of FO with the following additional rules:

$$\begin{array}{lll} \Gamma, \overrightarrow{a} \models \exists X.\phi\,(x_1, \cdots, x_k, X) & \text{iff.} & \exists B \subseteq dom(\Gamma),\ \phi\,(a_1, \cdots, a_k, B) \\ \Gamma, \overrightarrow{a} \models \forall X.\phi\,(x_1, \cdots, x_k, X) & \text{iff.} & \text{for all } B \subseteq dom(\Gamma),\ \phi\,(a_1, \cdots, a_k, B) \\ \Gamma, (a, B) \models x \in X & \text{iff.} & a \in B \end{array}$$

In opposition to FO, the transitive closure of relations are expressible in MSO. And more generally the expressive power of MSO over trees and words is clearly determined by the relation with automata [Büchi, 1960; Doner, 1970; Thatcher and Wright, 1968b].

Formal languages are defined to formally talk about objects, to query objects and decide *properties* of objects. We didn't yet define formally what those properties was. In the next section we provide how the concept of property is handled using formal languages.

3.2 Properties

The concept of properties is philosophically very complex [Sta, 2011]. But formally, hereinafter a property is just a class of relational structures $\mathcal{P}$. A relational structure Γ has a property $\mathcal{P}$ if it is a member of $\mathcal{P}$. Thus properties are designated using closed formulas of logical languages (ex. FO, MSO); with the following meaning. A closed formula ϕ is identified with the class of relational structures for which ϕ holds. That is the property denoted by ϕ is the class of structures Γ such that $\Gamma \models \phi$. More generally, in the database community, a formula ϕ is interpreted as a query which evaluates on relational structures. And the query ϕ evaluated on a relational structure Γ returns assignments of the free variables of ϕ into elements of Γ. The evaluation of ϕ on Γ is then the set of assignments for which ϕ holds. Queries without free variables i.e closed formulas are called boolean queries. Thus properties can be identified with boolean queries over relational structures. And relational structures which has property ϕ are those which validate or satisfy the boolean query ϕ. A relational structure which satisfies a boolean query ϕ is called a model of ϕ.

Identifying properties with boolean queries (or closed formulas) ϕ raises two main problems studied in finite model theory. The **Satisfiability** and **Model checking** problems.

The *satisfiability* problem is the problem of deciding whether, given a formula ϕ of some logical language, there exits a relational structure which is a model of ϕ. The satisfiability problem is already known undecidable

for FO formulas of arbitrary finite structures [Church, 1936; Turing, 1937; Trahktenbrot, 1963]. However for relational structures representing trees the satisfiability is proven to be decidable both for FO and MSO formulas [Rabin, 1968].

In the *model-checking* problem, one is concerned about deciding, given a relational structure Γ and some formula ϕ, whether Γ is a model of ϕ. The model checking problem is then deciding whether some relational structure Γ has property ϕ. So the model checking checking problem is mainly what concerns us in this thesis.

3.3 Model checking: exact and approximate

The model checking problem is the problem of deciding whether some relational σ-structure Γ satisfies some property ϕ of some logical language over the signature σ. Therefore, we also refer to the model checking problem as the *property checking* problem.

3.3.1 Property checking: exact model checking

The model checking problem has been intensively studied in computer science: in database theory, automated validation for example. Thus many results about the complexity of the model checking problem for FO and MSO were obtained. We give below some of the known results.

Theorem 3.1 (Stockmeyer [1974]; Vardi [1982]). *The model checkin problem for* FO *and* MSO *logics is* PSPACE *complete.*

This holds for any class of structures that contain at least one structure with two elements in his domain.

This theorem shows that model checking is hard. But as dealing with harness is custom for computer scientists, more studies of model checking have then led to neater results about the time complexity of this problem. We recall that the inputs of the model checking problems are a property ϕ with some σ-structure Γ. Thus the time complexity of the model checking is parametrized with the size of ϕ: $|\phi|$ and the size of Γ: $|\Gamma|$. The size of a sentence in some logic being just its number of symbols. We next provide known results about the time complexity of property checking using the notion of *fixed-parameter tractable (ftp) problems*. A property checking problem is ftp if there is a computable function f, and a polynomial p, and an algorithm solving the problem with time complexity: $f(|\phi|) \cdot p(|\Gamma|)$. One other result that confirms the hardness of model checking is that: unless P = NP, the model checking for MSO is not ftp for graph structures. This is mainly because 3-colourability of graphs is expressible in MSO. Using parametrized complexity classes, a similar result were obtained by Downey, Fellows, and

Taylor [1996]: unless AW[$*$] = FPT, the model checking problem for FO is not ftp. From these additional results, one sees that in general the model checking problem is untractable. However, on the tracks of tractable restrictions of MSO-model checking a celebrated result is the following:

Theorem 3.2 (Courcelle and Mosbah [1990]). *Model checking for* MSO *is solvable in time* $f(|\phi|) \cdot |\Gamma|$ *for the following class of structures:*

- *words*
- *trees*
- *graphs with bounded tree-width*

Yet even with this nice result, where the time complexity is linear when the property ϕ is fixed, the history is not finished. Indeed, it is proven that the function f in the previous theorem can not be elementary and is more precisely of the form:

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^{|\phi|}}}}}}\right\} \text{height } \Theta(|\phi|)$$

This means that algorithms obtained from this theorem are generally infeasible in practice. But for the class of structures with bounded degree, there exists better FO-model checking algorithms with complexities at most: $2^{2^{2^{|\phi|}}} \cdot |\Gamma|$. Another approach to obtain better algorithms is to consider restrictions of the logical languages. This is mainly what is done when considering FO[k] logics, $k \in \mathbb{N}$, that is FO sentences with a bounded number of variables.

The hardness of model checking, which culminated with all the previously mentioned efforts also opens the door for other methods. Approximability and Randomization are rightfully the best alternatives to hard exact and deterministic computation, as noticed by Papadimitriou [1994]. However while approximation is clear for computations of functions which values are in the set of reals or integers (using a norm for example), it remains to know what would be such thing for a model checking problem. Indeed the answers of algorithms for a model checking problem are boolean and can be only *true* or *false*. In the next section we explain how this is done in property testing which combines approximation and random access to structures domains.

3.3.2 Property testing: approximate model checking

The notion of efficient algorithm has evolved with the amount of information processed in computer systems. Traditionally linear time algorithms were considered feasible, but with the internet and now social networks, huge data need to be processed. This as a result has shifted the notion of efficiency, as now *sublinear* algorithms which inspect a minuscule fraction of their input

data are the most realistic algorithms [Rubinfeld and Shapira, 2011]. However in most situations accuracy has to be traded for efficiency, because exact decision making is mainly impossible without a complete inspection of the whole object (structure) for which a property is being checked. Nevertheless while exactness is known not always possible, one has to guaranty an accuracy as good as possible. In property testing, one provides an approximate answer to the model checking problem, with algorithms that randomly access the domain of their input relational structures. Besides the fact that randomness is needed for most sublinear algorithms, so that every part of the input data can effectively be accessed, it is also known that in many cases randomized algorithms turn out to be very efficient. The traditional notion of approximation, where the output of an algorithm need to be close to some value, is not appropriate for decision problems with only *true* and *false* as answers. Thus a tailored version of approximation using edit distances between input objects is introduced in Property Testing. Property testing approach to approximation of decision problems transforms languages into *promise problems* [Even, Selman, and Yacobi, 1984], excluding some inputs.

Blum, Codenotti, Gemmell, and Shahoumian [1995] where the first to consider problems of this kind and the general notion of property testing was first formulated by Rubinfeld and Sudan [1996] for program checking. Definitions were given by Goldreich, Goldwasser, and Ron [1998] in their seminal paper, where they considered property testing as a framework for studying combinatorial objects such as graphs. Since then many properties were studied in the flavour of property testing [Goldreich and Ron, 2011; Alon and Shapira, 2008; Alon et al., 2009; Onak, Ron, Rosen, and Rubinfeld, 2012]. For example, property testing was applied to study properties of hypergraphs [Newman and Sohler, 2011], boolean functions [Alon and Shapira, 2002; Ron, Rubinfeld, Safra, and Weinstein, 2011], and geometric functions; see surveys of Goldreich [1998], Ron [2000], Fischer [2001] and Czumaj and Sohler [2010] for example. Ron [2008] also emphasis property testing connection to learning theory. Various aspects of property testing were intensively studied through the years: starting from the study of particular properties [Goldreich et al., 1998], passing through general characterizations of testable properties [Alon et al., 2009], and recently to the introduction of new property testing frameworks [Halevy and Kushilevitz, 2007; Balcan, Blais, Blum, and Yang, 2012].

In the next section we first begin by explaining which kind of approximation is considered in property testing and discuss some of the field results.

Approximation for model checking

We recall that trees, words, and graphs are the objects for which one would like to check their properties. Those objects were then represented as relational structures over some vocabulary. And under some logic (ex. FO,

MSO), their properties are designated by closed formulas or boolean queries. The problem of deciding whether some word, tree or graph have some property, was then reduced to the model checking problem (see Sections 3.1 and 3.3.1). Hence, the set of words, trees or graphs, which have some property denoted by a sentence ϕ, of some logical language over their representations, is identified with the class of representations or relational structures which are models of ϕ: $\{\Gamma \mid \Gamma \models \phi\}$. We intentionally use here set notations for a class which might not be a set, but without difficulties this class can be identified with a set of isomorphic classes of structures. One isomorphic class per a represented object. Thus now the interesting question is what would it be for a word, tree, or graph to approximately satisfy a property ϕ? The idea is to say that an object approximately satisfies ϕ, if it is very similar to another object which is known to satisfy ϕ. However a similarity notion between objects is formally a distance measure (see Section 2.2). Therefore, under some edit distance between objects (ex. words, trees or graphs), one says that an object approximately satisfies some property ϕ: if it is at a small distance to another object (possibly the same) which satisfies ϕ. Before taking further this idea, let us notice that distance measures defined in Section 2.2 for trees, graphs or words generalize to distance measures between their relational structures. This is mainly done by considering that the distance between two structures is exactly the distance between the two objects they represent. In the same way one defines the size of any relational structure isomorphic to some object (word, tree or graph) as the size of this object. Now getting the discussion to a more concrete level, let us consider only words and show why the above idea must be tuned a bit in order to have approximate model checking which could be efficiently solved using randomized algorithms.

Consider the alphabet $\Sigma = \{a, b\}$, the first-order property $\phi = \forall x.lab_a(x)$ and the problem of approximately checking whether some word $w \in \Sigma^*$ have the property ϕ; under the Hamming distance between words. Note that words that has property ϕ are those whose positions are labelled only with a. If we understand "approximate model checking" as we did above: then we would like, given an additional parameter $l \in \mathbb{N}$, to efficiently decide whether $d_h(w, \mathcal{P}_\phi) < l$; where $\mathcal{P}_\phi = \{w' \mid S_{w'} \models \phi\}$. By efficiently one means providing a (BPP) randomized algorithm which solves this problem with complexity depending only on l and $|\phi|$; and independently of the size of w. Would this be possible? the answer is clearly no, as shown in the following. Indeed, we use Yao's principle to show, on this example, that if l is a constant independent of the size of w, then there is no BPP algorithm which solves this problem. The idea behind is that to distinguish words with the property ϕ, from those which are at distance l of having ϕ, the only possibility is to randomly look at some positions of the input word w and see whether one of them is labelled b. But for big enough words, the probability to discover one such position can be made as small as possible if l is not related to the

size of w. Let $n \in \mathbb{N}$, $\mathcal{N}$ be the set of words of size n, for which there is exactly l positions labelled b. And let $\mathcal{P} = \mathcal{P}_\phi \cup \Sigma^n$ be the set containing the only word of size n which has the property ϕ. Note that $|\mathcal{N}| = \binom{n}{k}$, where $\binom{n}{k}$ denotes the binomial coefficient. Any BPP algorithm that may solve this problem must separate, with high probability, words in $\mathcal{N}$ from the one in $\mathcal{P}$. That is it should not err with probability greater than some (say) $\nu = 1/3$. Now let p be the probability distribution over $\mathcal{P} \cap \mathcal{N}$ such that: $p(w) = 1/2$ if $w \in \mathcal{P}$ and $p(w) = 1/2|\mathcal{N}|$ for $w \in \mathcal{N}$. Let us see how deterministic algorithms that may solve this approximate model checking behave on inputs chosen according to p. Note also that any element of $\mathcal{N}$ is uniquely identified with the set of positions that are labelled b. Then, for $w \in \mathcal{N}$, any deterministic algorithm that does not read the positions labelled b in w, will provide the same answer as if it was input the element of $\mathcal{P}$. Thus any deterministic algorithm reading only $k \stackrel{def}{=} f(l)$ positions will err with probability $1/2 \cdot \binom{n-k}{l} / \binom{n}{l}$, on input words chosen according to p, where f is a function of l.
Note that:

$$\binom{n-k}{l} / \binom{n}{l} \geqslant (1 - k/n)^l \qquad (1)$$

So for l and f fixed, this ratio can be made greater than 2ν as n grows; therefore any deterministic algorithm reading k positions will err at least with probability 2ν. Applying Yao's principle we conclude that there is no randomized algorithm for solving this problem.

The example above shows that, in order to have efficient algorithms, the notion of "approximately satisfying a property" must be related to the size of the object being considered. Thus the notion of "approximately satisfying a property" is defined hereinafter in the following way:

Definition 3.1 (ϵ-close, ϵ-far). *Let σ be a relational vocabulary, ϕ a sentence of some logical language over σ (ex.* FO*,* MSO*), $1/2 > \epsilon > 0$ a precision parameter and d a distance measure over the set of relational σ-structures. A σ-structure Γ is ϵ-close to have property ϕ if and only if $d(\Gamma, \mathcal{P}_\phi) \leqslant \epsilon|\Gamma|$, where $\mathcal{P}_\phi = \{\Gamma' \mid \Gamma' \models \phi\}$. Structures that are not ϵ-close to a property ϕ are said to be ϵ-far from ϕ.*

Property testing

Now the approximate model checking (*or property testing*) consist of deciding, for a property ϕ, a relational structure Γ and a precision parameter ϕ, whether Γ is ϵ-far from ϕ or it satisfies ϕ. Furthermore for structures ϵ-closed to ϕ, the decision is not required to be correct. BPP algorithms that solve the approximate model checking (or property testing) for ϕ: are called *testers* for the property ϕ. Moreover a tester is not input the structure Γ, instead it is input an oracle access to Γ. In Chapter 4 we detail how one formally defines

random access to relational structures. But for now on one should just see this as a set of functions which can generate elements of the domain of the relational structure and can tell information about the structure relations. In some situations the oracle also provides the sizes of the structure relations. For example for words, the oracle can access the label of some word at a provided position. And the oracle access of words can also provide the size of the word.

Definition 3.2 ((d, ϕ, f)-tester, query complexity). *Let σ be a relational vocabulary, $f : \mathbb{N}^2 \times \mathbb{R} \to \mathbb{N}$, ϕ a sentence of some logical language over σ and d a distance measure over the set of relational σ-structures. A (d, ϕ, f)-tester is any randomized algorithm that inputs: a precision parameter $1/2 > \epsilon > 0$, an oracle rdm_Γ that access the structure Γ; and which behave as follows:*

1. *if Γ satisfies ϕ then the answer is* CLOSE *with probability at least $\frac{2}{3}$, and*
2. *if Γ is ϵ-far from ϕ then the answer is* NO *with probability at least $\frac{2}{3}$*
3. *Furthermore the algorithm accesses (or reads) Γ at most $f(|\phi|, |\Gamma|, \epsilon)$ times*

The query complexity of a (d, ϕ, f)-tester is the maximum, on all its inputs, of the number of time it uses its input oracle.

Coming back to the above example which made us adapt our first idea for approximate model checking, note that the argument in this example does not hold any more. Indeed for $l = \epsilon n$ and $f(l) = 1/\epsilon$, the upper bound $(1 - k/n)^l$ in (1) is close to 0 as n grows. And one can even see that the algorithm which uniformly selects $f(l)$ positions and which answers according to the labels of the selected positions, have a small error probability. Indeed the error probability of such algorithm is:

$$1/2 \cdot \binom{n-k}{l} / \binom{n}{l} \leqslant 1/2 \cdot (1-\epsilon)^l \leqslant 1/2 \cdot \exp(-\epsilon l)$$

A property ϕ is said to be testable with query complexity $O(f)$ if there is a function $g = O(f)$ and a (d, ϕ, g)-tester for ϕ. And we say that ϕ is simply *testable* without mention of a query complexity if it has a tester whose complexity can be bounded by a function independent of the size of the structure accessed by the input oracle. Such tester will also said to have constant query complexity even though it depends on the precision parameter and the property size. One should have noticed that in Definition 3.2, a tester for a property ϕ may err on two sides: on far structures (item 1) and on structures satisfying ϕ (item 2). Testers that never err on structures satisfying ϕ are called **one-sided** (that is we replace the probability 2/3 of item 2 in Definition 3.2 by the probability 1). The efficiency of testers is usually measured by their query complexities. Note that the query complexity

is an information complexity, in the sense that it measures the number of times a tester requires information about the structure being tested; using its input oracle. While good query complexity may not necessarily imply good time complexity, fortunately in many (natural) cases it does. Further details are provided by Shapira [2006] who 'cook up' unnatural properties for which good query complexities does not imply good time complexities. Mainly testers solving such properties needs to do hard computations using the size of the structure which oracle access is input. It is important at this point to notice that the query complexity of testers is strongly related to which kind of information can be provided by the input oracle access. This point has lead to distinguishing two type of testers: **oblivious** testers which do not require their input oracle to provide the size of the accessed structure and other testers which require such size access. In the case of graphs for example, oracle access was represented as adjacency matrices or bounded list of adjacent nodes. This has lead to two different models: the *dense model* and the *bounded degree model*. And many results in property testing under the *dense model* are in contrast with those of the *bounded degree model* [see e.g. Ron, 2008], when the query complexity is considered. Oblivious testers capture the essence of property testing as most applications of property testing involves properties of networks such as the internet whose size is unknown [see e.g. Fischer, 2001; Ron, 2000]. Shapira [2006] also noticed that most testers for dense graphs were indeed oblivious.

Areas where property testing have been applied are for example the analysis of large graph and code theory. In the study of large graphs, the important clusterability problem [see e.g. Raghavan, 1997; Anderberg, 1973], is known NP-complete [Fowler, Paterson, and Tanimoto, 1981; Megiddo and Zemel, 1986]. And this problem can be efficiently decided in the framework of property testing [Alon, Dar, Parnas, and Ron, 2003]. And in code theory, linear code have been studied. As discussed by Ahlswede, Cai, Li, and Yeung [2000]; Koetter and Médard [2003]; Sanders, Egner, and Tolhuizen [2003]; Tan, Yeung, Ho, and Cai [2011], linear coding increases the transfer rate and the code construction complexity of network multicast. Locally testing codes was explicitly defined by Friedl and Sudan [1995]; Rubinfeld and Sudan [1996]; Arora [1994] and has gained attention over the years do to its relation with Probably Checkable Proof (PCP) [see e.g. Blum et al., 1995]. Locally testing of linear codes is therefore an important challenge that has been studied by Alon, Kaufman, Krivelevich, Litsyn, and Ron [2005]; Bhattacharyya, Kopparty, Schoenebeck, Sudan, and Zuckerman [2010], for the case of Reed-Muller codes. Roughly speaking testing a Reed-Muller code, corresponds to testing whether a function of some vectorial space over some finite field is a polynomial of degree $n \in \mathbb{N}$, or it is a function far from being so. In all works addressing this problem, the distance between functions is the Hamming distance, that is the size of the set of vectors in which the two functions differ. The tester provided by Bhattacharyya et al. [2010] is

optimal and has query complexity $\Theta(2^n + 1/\epsilon)$, where $\epsilon > 0$ is the precision parameter. Note that the query complexity is independent of the size of the vectorial space.

Testers may also be useful as a preliminary step of an exact model checking problem. This way ϵ-far structures can be quickly disregarded. Below we formally define testers for a class of properties denoted by some logical languages (ex. FO, MSO).

Definition 3.3 ((d, f)-tester, query complexity). *Let σ be a relational vocabulary, $f : \mathbb{N}^2 \times \mathbb{R} \to \mathbb{N}$ and d a distance measure over the set of relational σ-structures. A (d, f)-tester for the approximate model checking problem, of some logical language over σ and under the distance d, is any randomized algorithm that inputs: a sentence ϕ, a precision parameter $1/2 > \epsilon > 0$, an oracle rdm_Γ to access the structure Γ; and which behave as an (d, ϕ, f)-tester which inputs ϵ and rdm_Γ. That is:*

1. *if Γ satisfies ϕ then the answer is* CLOSE *with probability at least $\frac{2}{3}$, and*
2. *if Γ is ϵ-far from ϕ then the answer is* NO *with probability at least $\frac{2}{3}$*
3. *Furthermore the algorithm accesses (or reads) Γ at most $f(|\phi|, |\Gamma|, \epsilon)$ times*

The query complexity of a (d, f)-tester is the maximum, on all its inputs, of the number of time it uses its input oracle.

To have an idea about how this approximate model checking might be useful, let us informally discuss what it means for a structure Γ to be ϵ-far from a property ϕ under some edit distance d. In fact, Γ is ϵ-far to ϕ if it has to be "drastically changed" to become a structure satisfying ϕ. And a drastic change for a structure is here that the number of edit operations one has to use for changing Γ into a structure satisfying ϕ is greater than $\epsilon|\Gamma|$. Thus in model checking situations when one is promised input structures that either have some property or are ϵ-far from that property, a randomized algorithm that solves the corresponding approximate model checking problem might be preferable for efficiency reasons. As represented in 3.1, a (d, ϕ, f)-tester will separate with high probability the class of structures satisfying ϕ from the class of structures ϵ-far from this class. Thus the approximate model checking will also be called *approximate membership testing*. To further see why the name "approximate membership testing" is appropriate, consider one-sided testers that solve an approximate model checking problem. Such testers are required to answer CLOSE for structures Γ satisfying ϕ; when input a property ϕ, a precision parameter ϵ and an access rdm_Γ. And therefore whenever they answer NO for a structure, this means that they found a "witness" that Γ does not satisfy ϕ. Thus with high probability they can tell whether a structure can be "easily repaired" to satisfy a property. Therefore properties that can

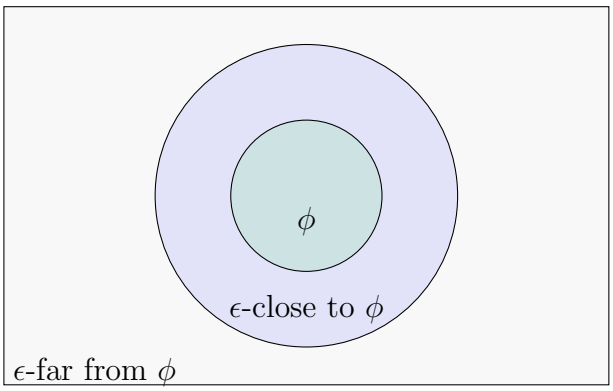


Figure 3.1: Picture representing how a tester for a property ϕ approximately separates far structures from the one satisfying ϕ.

be tested with one-sided testers must somehow be such that ϵ-far structures can be easily "*locally witnessed*" with small oracle accesses. The relation between property testing and *local repairability* was investigated by Austin and Tao [2009]. They indeed related property testing of hereditary dense graph properties to a notion of *local repairability*

Next we survey results for the approximate model checking of FO and MSO logical languages for graphs.

Approximate model checking for graphs

All the studies of approximate model checking surveys in the section are under the edit distance d_e (see Section 2.2.4). Approximate model checking of FO properties on dense graph structures was initiated by Alon, Fischer, Krivelevich, and Szegedy [2000a]. This study was taken further by Fischer [2005]. They showed that all properties expressible with first-order formulas with quantifiers alternations of the form "$\exists\forall$" are testable, whereas there exists properties with quantifiers alternations of the form "$\forall\exists$" which are not testable with query complexity independent of the size of the graph. These results mean that there is no tester (with constant query complexity) for the whole class of graph properties expressible with FO sentences. Thus it is also clear from these results that for properties defined by MSO sentences there is no tester with constant query complexity.

Knowing that not all graph properties can be efficiently tested, the remaining question was how to characterize graph properties that can be efficiently tested. Such characterisation was obtained by Alon, Fischer, Newman, and Shapira [2009] for dense graphs with oracle access designed as adjacency matrices (the dense model). This characterisation connects property testing to *external graph theory*. In fact, Alon, Fischer, Newman, and Shapira [2009] noticed that all previous testers for properties of dense graphs were obtained

by use of the *Szemerédi Regularity Lemma* [Szemerédi, 1976, 2013], or else by use of the *Removal Lemma* for graphs (the proof of the second lemma is obtained using an application of the first one). Then their characterization of testable graph properties essentially tells that testable graph properties are characterized by constant *regularity instances*. The Regularity Lemma allows to precisely approximate large graphs by random graphs, and the removal lemma which is an application of the previous lemma asserts that given a fixed graph G, any graph that have few copy of G can be made G-free by removing few edges. From the Removal Lemma it follows that hereditary properties of dense graphs are all testable [Alon, Fischer, Newman, and Shapira, 2009; Austin and Tao, 2009].

Further link between property testing of dense graphs and external graph theory is obtain by Lovász and Vesztergombi [2013], with a characterisation of testable properties using convergent sequences of graphs. Indeed, Lovász and Vesztergombi [2013] introduced a limit object (*graphon*, i.e. real valued functions $[0,1]^2 \to [0,1]$), for such convergent sequence of graphs; and testable properties were proven to be exactly those which also have their closures (thus graphon properties) testable. This result can be interpreted as a proof of $NP = P$ in the context of property testing of dense graphs, that is a property is testable if and only if it is *non-deterministically* testable [see e.g. Gishboliner and Shapira, 2013]. A property is non-deterministically testable if one can supply a "certificate" to each graph such that once the certificate of a graph is provided, one can test if the graph actually has the desired property.

3.4 Finite automata

Known results in approximate model checking of logical languages for trees and words are surveys in this section. However these results will be presented using automata for representing properties of words and trees and not a sentence of the logical languages introduced above. Automata are states machines that process words or trees using the notion of runs (defined below) and the property defined by such machine is just the set of words or trees that ends in final states when processed by the automata. Most efficient algorithms in model checking were in fact obtained using representations of properties by automata. This is due to two important results: the class of word properties expressible with a MSO sentence is exactly those that can be represented with automata [Büchi, 1960; Elgot, 1961] and the same relation holds between MSO sentences on trees and tree automata [Thatcher and Wright, 1965, 1968a]. And a MSO sentence can be effectively converted into an automaton recognizing the same property. The reason why representing properties by automata is helpful to obtain efficient algorithms in model checking is probably because automata come alone with a simple process for

checking whether a word or tree satisfies the property they denote. Whereas logical formula are kind of syntactical or too declarative. Automata (specially tree automata) has regained interest in the context of XML, as shown in the surveys by Neven [2002a,b] and Schwentick [2007].

Below we define automata for words and trees. Before getting further, we mention that definitions we provide for words and trees will also apply for the class of relational structures representing them.

3.4.1 Word automata

A non-deterministic finite automaton (NFA) is a tuple $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$, where Σ is an alphabet of letters, Q a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ a transition relation, and $\mathit{init}, \mathit{fin} \subseteq Q$ the subsets of initial and final states. If $(q, a, q') \in \Delta$, we say that $q \xrightarrow{a} q'$ is a transition or rule of A. Furthermore, we write $\xrightarrow{a}_A$ for the relation $\{(q, q') \mid (q, a, q') \in \Delta\}$ and $\rightarrow_A$ for the one-step reachability relation $\cup_{a\in\Sigma} \xrightarrow{a}_A$. The k-reachability relation is defined inductively by $\rightarrow_A^k = \rightarrow_A^{k-1} \circ \rightarrow_A$ and $\rightarrow_A^0 = \{(q, q) \mid q \in Q\}$. The reachability relation of A is the union of all k-reachability relations $\rightarrow_A^* = \cup_{k\geqslant 0} \rightarrow_A^k$.

A *quasi-run* of an NFA A on a fragment F of a word $w = a_1 \dots a_n$ is a function $r : \mathit{dom}(F) \to Q$ such that $r(i-1) \xrightarrow{a_i} r(i)$ is a transition of A for all $i \in F$. A quasi-run is called total if its domain is $\mathit{dom}(w)$. Note that $q \rightarrow_A^* q'$ if and only if there exists a word $w \in \Sigma^n$ and a total quasi-run r on w by A such that $r(0) = q$ and $r(n) = q'$.

A run r of A on a fragment F of w is a quasi-run of A on this fragment such that $r(0) \in \mathit{init}$. A run is called successful if it is total and satisfies $r(n) \in \mathit{fin}$. As usual, we say that w is recognized by A if there exists a successful run of A on w. The language $L(A) \subseteq \Sigma^*$ is the set of all words w recognized by A. We call a language $L \subseteq \Sigma^*$ regular if it is equal to $L(A)$ for some NFA A. The size of an NFA A is the sum of the number of its states and rules $|A| = |Q| + |\Delta|$.

A NFA $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$ is called a deterministic automaton (DFA) if the transition relation Δ is a function from $Q \times \Sigma$ to Q. A classical result is that the class of words languages defined by NFA is exactly the one defined by DFAs. However the size of the DFAs that recognize the same language as an NFA A can be $2^{|A|}$. Another important result is that the class of languages recognized by automata corresponds to the class of regular word languages (Kleene theorem). We recall that a language is regular if it can be recognized by a regular expression. And regular expressions over an alphabet Σ are defined with the following grammar:

$$e := a \mid e \cdot e \mid e + e \mid e^* \mid \varepsilon$$

where $a \in \Sigma$ and ε is the empty word. The language defined by the regular expression e is denoted $L(e) \subseteq \Sigma^*$. For a complete introduction to word

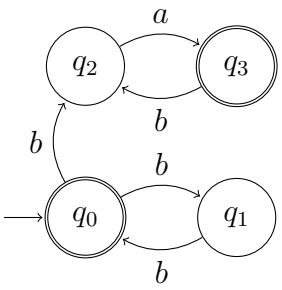


Figure 3.2: An NFA for $L = (bb)^*(ba)^*$.

automata and regular languages one is referred to [Sudkamp, 2006]. An example of automaton is given Figure 3.2

3.4.2 Tree automata

The notion of automata have been generalized for trees by Doner [1965, 1970], and Thatcher and Wright [1965, 1968a] to study the decidability of second order logic. In this section we introduce automata for ranked trees and hedge automata for unranked trees. Note that unranked trees can be converted into binary trees using the encodings of Section 2.1.3. However in the context of approximate model checking, the distance between two trees does not always translate well to their encodings (see Chapter 6 for details). Thus one can not in general use such encodings to reduce the study of approximated model checking of general trees to the one of ranked trees. Yet in the case of the edit distance with moves, such approach works fine as shown by Magniez and de Rougemont [2007].

Bottom-up automata for Ranked Trees

A bottom-up non-deterministic finite tree automaton ($\uparrow$NFTA) is a tuple $A = (\Sigma^r, Q, \mathit{fin}, \Delta)$, where $\Sigma^r = (\Sigma, \mathit{arity})$ is a ranked alphabet, Q is a set of states, $\mathit{fin} \subseteq Q$ is a subset of final states and Δ is a set of transition rules of the form:

$$a(q_1, \cdots, q_k) \to q$$

where a is a symbol of Σ^r of arity $k \in \mathbb{N}_0$ and $q_1, \cdots, q_k, q \in Q$.

Note that in contrast with word automata there are no initial states, instead for $k = 0$ the above transition rule is of the form $a \to q$. A $\uparrow$NFTA runs on a ranked tree $\mathrm{t} \in \mathcal{T}^r_\Sigma$ starting from its leaves and moving upward assigning a state to any node of the tree. Thus a run of the $\uparrow$NFTA $A = (\Sigma^r, Q, \mathit{fin}, \Delta)$ on $\mathrm{t} \in \mathcal{T}^r_\Sigma$ is a function $r : \mathit{nod}_\mathrm{t} \to Q$ such that: for all nodes $w \in \mathit{nod}_\mathrm{t}$ with k children

$$b\,(r(w \cdot 1), \cdots, r(w \cdot k)) \to r(w) \in \Delta, \text{ where } \mathit{lab}_\mathrm{t}(w) = b$$

And a run r of t is called successful if $r(\varepsilon) \in \mathit{fin}$, that is the root of t is assigned a final state. A tree t is recognized by a $\uparrow$NFTA A, if A has a successful run on t. The language $L(A) \subseteq \mathcal{T}_\Sigma^r$ is the set of all trees recognized by A.

A $\uparrow$NFTA $A = (\Sigma^r, Q, \mathit{fin}, \Delta)$ is called a bottom-up deterministic finite tree automaton ($\uparrow$DFTA) if for every symbol a of Σ^r of arity $k \in \mathbb{N}_0$ and every states $q_1, \cdots, q_k$, there is at most one $q \in Q$ such that: $a(q_1, \cdots, q_k) \to q$ is a rule of Δ. As in the case of words, $\uparrow$NFTAs and $\uparrow$DFTAs have the same expressibility, that is they recognize the same set of tree languages. And the size of a $\uparrow$DFTAs can be exponentially bigger than the size of its equivalent $\uparrow$NFTAs.

Top-down automata for ranked trees

Many other types of automata have been considered for ranked trees. In here we mention top-down non-deterministic finite tree automata ($\downarrow$NFTA). These automata runs on ranked trees from their root to their leaves. A $\downarrow$NFTA is a tuple $A = (\Sigma^r, Q, \mathit{init}, \Delta)$, where $\Sigma^r = (\Sigma, \mathit{arity})$ is a ranked alphabet, Q is a set of states, $\mathit{init} \subseteq Q$ is a subset of initial states and Δ is a set of transition rules of the form:

$$q \to a(q_1, \cdots, q_k)$$

where a is a symbol of Σ^r of arity $k \in \mathbb{N}_0$ and $q_1, \cdots, q_k, q \in Q$.

A run of the $\downarrow$NFTA $A = (\Sigma^r, Q, \mathit{init}, \Delta)$ on $\mathrm{t} \in \mathcal{T}_\Sigma^r$ is a function $r : \mathit{nod}_\mathrm{t} \to Q$ such that: for all nodes $w \in \mathit{nod}_\mathrm{t}$ with k children

$$r(w) \to b\,(r(w \cdot 1), \cdots, r(w \cdot k)) \in \Delta, \text{ where } \mathit{lab}_\mathrm{t}(w) = b$$

And a run r of t is called successful if $r(\varepsilon) \in \mathit{init}$. Note that the definition of $\downarrow$NFTAs is very similar to the one of $\uparrow$NFTAs, the only difference is the arrow direction of transition rules. Thus with no surprise $\downarrow$NFTAs and $\uparrow$NFTAs, recognize the same class of tree languages. Next one can define top-down deterministic finite tree automata ($\downarrow$DFTAs) in the same way as we did for $\uparrow$DFTAs. However $\downarrow$DFTAs does not have nice properties since they are less expressive than $\downarrow$NFTAs.

Hedge automata

As previously noticed, unranked trees has gained much attention as a model for XML. Thus the notion of automata was extended to work directly on unranked trees and not on their binary encodings. We define such automata in the following. A non-deterministic finite hedge automaton (NFHA) is a tuple $A = (\Sigma, Q, \mathit{fin}, \Delta)$, where Σ is an alphabet, Q is a set of states, $\mathit{fin} \subseteq Q$ is a subset of final states and Δ is a set of transition rules of the following form:

$$a(R) \to q$$

where $a \in \Sigma$, $q \in Q$ and $R \subseteq Q^*$ is a regular language over the set of states Q. The regular languages R that appear in the rules are called the *horizontal languages* and they can be defined with word automata or regular expressions. Without lost of generality, one can consider only *normalized* NFHA, that is hedge automata such that for all $a \in \Sigma$, $q \in Q$, there is at most one rule of the form $a(R) \to q$.

A run r of a NFHA $A = (\Sigma, Q, \mathit{fin}, \Delta)$ on an unranked tree $\mathrm{t} \in \mathcal{T}_\Sigma$ is a function $r : \mathit{nod}_\mathrm{t} \to Q$ such that: for all nodes $w \in \mathit{nod}_\mathrm{t}$, of label $a \in \Sigma$ and with k children, there is a rule $a(R) \to r(w)$ such that $r(w \cdot 1) \cdots r(w \cdot k) \in R$. The r is successful if $r(\varepsilon) \in \mathit{fin}$, in such case t is said to be recognized by A. The language $L(A) \subseteq \mathcal{T}_\Sigma$ is the set of trees recognized by A.

A NFHA A is said to be a deterministic finite hedge automaton (DFHA) if for all transition rules $a(R) \to q$ and $a(R') \to q'$, one has $R \cap R' = \varnothing$ or $q = q'$. The set of tree languages recognized by NFHAs is exactly the set of tree languages that DFHAs recognize. However the size of a NFHA A can be exponentially smaller than the size of an equivalent DFHA A' for which $L(A') = L(A)$. Another useful result is that an unranked tree language is regular if and only if the set of its tree binary encodings is regular.

The size of all this kind of automata is denoted $|A|$ and defined as the size of its set of states and the size of its transitions: $|A| = |Q| + |\Delta|$.

For a complete introduction to tree automata one is pointed to [Comon et al., 2007a]. We recall that a classical result is that properties expressible by MSO sentences on words and tree structures are exactly those which can be recognized by automata (for words, ranked and unranked trees). Therefore the model checking problem which was defined using closed formulas, can naturally be reduced to model checking problem where properties are represented by automata. Hence in all membership testing problems previously defined (see Section 3.3), one can replace the input logical sentence with a automaton. For simplicity reasons the notions of automata and runs was defined on words and trees; but one should notice that they naturally generalize to any relational σ_{words}-structures and σ_{trees}-structures. Next we provide known results for membership testing with automata as input properties.

3.4.3 Exact membership for words and trees

We recall that the model checking (or membership) problem is the following:

inputs:	An automaton A and a σ_{words}-structure (or σ_{trees}-structure) Γ
question:	Is Γ recognized by A?

For all the automata introduced above, the membership problem can be solved with a combined time complexity as stated in the following theorem.

Theorem 3.3. *Membership of a σ_{words}-structure (resp. σ_{trees}-structure) Γ, to the class $L(A)$ of σ_{words}-structure (resp. σ_{trees}-structure) recognized by an automaton A, can be decided with time complexity $O(|A||\Gamma|)$. This holds for all the automata introduced above.*

The idea is that an automaton A recognize a structure Γ if it has a successful run on Γ. Thus to solve the membership problem, one has to decide if one of the possible runs of A on Γ is successful. This can be done efficiently by simulating all runs and checking whether at each step one can successfully continue.

Then membership problems for the classes of σ_{words}-structures and the class of σ_{trees}-structures can be efficiently decided using automaton. However the interest in approximately deciding membership to these properties is to provide algorithms that have time complexity independent of the size of their input structures, or which are at most sublinear in the size of the input structure.

3.4.4 Approximate membership for words and trees

The approximate membership problem, under a distance measure d and for some class of structures, is the following:

inputs:	An automaton A, a random access rdm_Γ to some σ_{words}-structure (resp. σ_{trees}-structure) Γ, a precision parameter $0 < \epsilon < 1$
question:	Is Γ recognized by A or is it ϵ-far from being recognized by A?

A solution (tester) to this problem is any BPP randomized algorithm that answers CLOSE with probability (say) $\frac{2}{3}$ if Γ is recognized by A, and NO also with probability $\frac{2}{3}$ if Γ is ϵ-far from any structure recognized by A. And for structures that does not satisfy any of these conditions, the algorithm can return anyone of these answers. The solutions to this problem discussed below, are one-sided, that is they answer NO only when input random access to structures that are ϵ-far from the class $L(A)$ of structures recognized by A. A solution to this problem can be iterated a polynomial time to obtain one that does provide correct answer with probability as big as one want. Thus the probability $\frac{2}{3}$ can be replaced by any constant $1/2 < \nu < 1$.

There are many distances that can be defined on words and trees structures. However, for all distances measures d and d' one the same class of structures, if d' is weaker than d, then any solution to the approximate membership problem under d, is also a solution to the approximate membership problem under d'. In the case of words for example, any solution to the problem under d_h is also valid for d_l and d_m.

The query complexity of a tester (solution) is the number of time it accesses Γ through the oracle access to Γ. So the query complexity of testers is clearly

related to the access queries provided by random access of structures. Here we just give hints on the allowed access queries, and in Chapter 4, details on random accesses are provided. The random access to words provide its size and also the label at any position. It generates uniformly a position and also provide for some position the previous and next position. And for trees it generates uniformly some node; it also provides the depth, label, next-sibling and first-child of any node.

Approximate membership testing of words

Approximate membership testing of words was initiated by Alon, Krivelevich, Newman, and Szegedy [2000b], and a solution to this problem was provided. Alon, Krivelevich, Newman, and Szegedy [2000b] studied this problem under the Hamming distance between words and the complexity of their tester is stated in the following theorem.

Theorem 3.4 ([Alon et al., 2000b]). *Under the Hamming distance, the approximate membership problem can be solved with a tester whose query complexity is* $2^{O(|A|)} \cdot \log^3(1/\epsilon)/\epsilon$

Indeed, Alon, Krivelevich, Newman, and Szegedy [2000b], considered regular languages represented by deterministic automata over words and solved the approximate membership problem. They also showed that under the Hamming distance, solutions to the approximate membership problem requires at least $\Theta(1/\epsilon)$ query complexity. One other result of Alon, Krivelevich, Newman, and Szegedy [2000b] is that, under the Hamming distance, *context-free languages* can not be tested with constant query complexity. Since the Levenshtein distance and the edit distance with moves are weaker than the Hamming distance, therefore the tester of Alon, Krivelevich, Newman, and Szegedy [2000b] also applies to these distances. However in Chapter 5, we provide tester for the Levenshtein distance with better query complexity (the dependence in the automaton size is made polynomial). A tester specific to the approximate membership testing under the edit distance with moves was also provided by Fischer, Magniez, and de Rougemont [2006]. Fischer, Magniez, and de Rougemont [2006] use a statistical approach to solve the approximate membership problem. While their tester has larger query complexity, their method generalize to trees and to solving approximate equivalence problems.

Approximate membership testing of trees

The question of approximately testing membership of tree structures, under the standard edit distance, was raised by Chockler and Kupferman [2004]. This problem is still open since no tester with constant query complexity is known in the literature to solve it. Magniez and de Rougemont [2007],

approximate membership was studied in the context of the edit distance with moves and a solution was provided as stated in the following theorem.

Theorem 3.5 ([Magniez and de Rougemont, 2007]). *Under the edit distance with moves, approximate membership of trees can be solved with query complexity* $2^{O(2^{2|A|+1}/\epsilon)}$

Magniez and de Rougemont [2007] first provided a tester for ranked trees and then they proved that the edit distance with moves between the encodings of two unranked trees is in the same order of the distance between the unranked trees (this is not the case for the standard edit distance [Akutsu, 2006]). Moreover random access to nodes in the binary encoding of a tree t can be simulated from random accesses to t. Thus they generalized easily their tester of ranked trees to unranked ones. In this thesis we study the approximate membership problem with regard to the strong edit distance in Chapter 6. Under this distance, we show that one can not solve this problem with constant query complexity. This will give ideas why the problem raised in [Chockler and Kupferman, 2004] is hard and might not have any solution with constant query complexity. Indeed the situation in property testing is the following: the weaker is the distance, the better are chances that there is a tester with constant query complexity. Now $d_{move} \leqslant d_{stand} \leqslant d_{strong}$ and there is no tester for the strong edit distance while such tester exist for the edit distance with moves. However approximation with weak distances might not be satisfactory in applications involving XML for example. Indeed weak distances would not detect some structural differences between trees. In the next section we present XML. A direct application of membership testing of trees is approximately testing whether an XML document follows a DTD.

3.5 XML

The Extensible Markup Language (XML) is a standard data format for representing information exchanged by machines. It is designed by the XML-1.0 specification of the W3C consortium [Bray et al., 2008b]. In document processing, XML is used to represent documents and transformations (DocBook, SGML). It also enables to represent web pages in XHTML and many semi-structured databases are nowadays designed in XML (BaseX, Oracle Berkeley DB XML, xDB).

XML also comes with schema languages, query languages and transformation languages. Standard schema languages are *Document Type Definition* (DTDs) [Bray et al., 2008b], XML SCHEMAS [Fallside and Walmsley, 2004; Chidlovskii, 2000; Martens, Neven, Schwentick, and Bex, 2006a], and RELAXNG [van der Vlist, 2003]. The standard query language for retrieving information from XML documents is XPath [XPath]. And standard languages

for transformation are XSLT [Clark, 1999] and XQuery [Boag, Chamberlin, Fernández, Florescu, Robie, and Siméon, 2007].

An XML document is usually parsed into an unranked data tree that satisfies the XML data model (see [XPath]). This data model underlies all standard languages for XML processing and also all XML databases. An example of XML document with the tree modelling it can be found at Figure 3.3.

A prime task is to check whether some XML document is valid for some schema. Other tasks that are commonly performed on XML documents are *query answering* and *data transformation*. In the query answering task, one retrieves information which match some query, from an XML document. And the transformation task consist in transforming some XML document into another format. This later task is commonly used, for example, in Data Exchange one transforms XML documents satisfying some schema into XML documents that satisfy another schema. The efficiency of schema validation and query answering as well as type checking of transformation languages [Friese, 2011; Milo, Suciu, and Vianu, 2003; Maneth, Perst, and Seidl, 2007] has also been studied by such approach.

XML have gain much interest the last decade and the aforementioned tasks have been deeply studied [Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita, and Zhang, 2002; Vansummeren, 2005; Koch, 2006; Onder and Bayram, 2006; Janssen, Korlyukov, and Van den Bussche, 2007; Michiels, 2007; Schmidt, Scherzinger, and Koch, 2007; Debarbieux, Gauwin, Niehren, Sebastian, and Zergaoui, 2013]. Usually these tasks are formally studied using unranked data trees as models for XML documents and characterising all schema, query and transformation languages by means of tree automata or FO and MSO logics over tree structures [Wei, Li, Rundensteiner, and Mani, 2006; Benedikt, Libkin, and Neven, 2007; Onizuka, 2010; Gauwin and Niehren, 2011]. Using this approach the expressiveness of all the standard languages was characterized. RelaxNG, Core XPath 2.0 and XQuery are as expressive as MSO boolean queries on trees, FO n-ary queries on trees [Martens, Neven, Schwentick, and Bex, 2006b] and FO definable tree transformations [Benedikt and Koch, 2009] respectively.

However, lately, with the emergence of NoSQL databases (ex: JSON databases) and NoSql query languages (ex: SPARQL), XML is less successful these days. Note that graphs are more appropriate as data models of NoSQL databases.

In this thesis we are mainly concerned about the validation task, and more precisely we study an approximate version of the validation task inspired by property testing. We will be only concerned with structural differences of XML documents, thus one ignores data values. Below we mention some of the commonly used schema languages and relate those schemas languages to tree automata.

```
<collection>
  <book>
    <title>Principia Mathematica</title>
    <author>Russell</author>
    <author>Whitehead</author>
    <year>1913</year>
  </book>
  <book>
    <title>M.W.M.W.W.S</title>
    <author>Cavell</author>
    <year>1969</year>
  </book>
</collection>
```

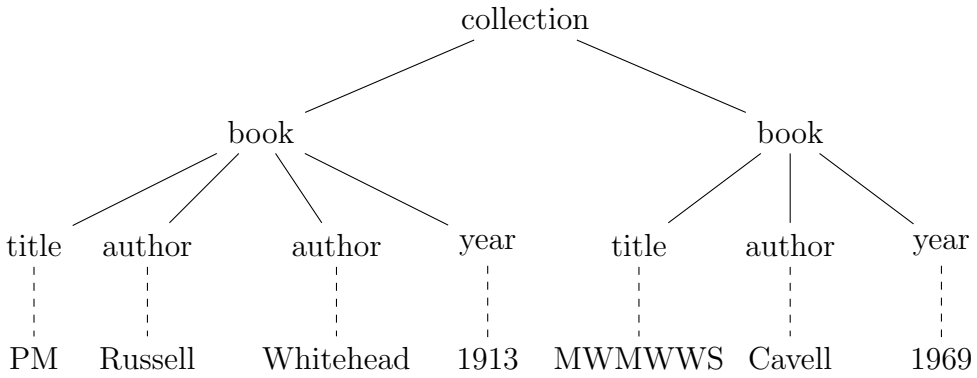


Figure 3.3: A XML document representing a collection of books with its tree model

3.5.1 Schemas

Many schemas have been used to denote collections of XML documents. These schemas differ in their syntax [Lee and Chu, 2000] but mostly they also do not have the same expressibility [Murata, Lee, Mani, and Kawaguchi, 2005b; Bex, Neven, and Van den Bussche, 2004], i.e, they do not denote the same collections of documents. The most commonly used schemas are the standards: *Document Type Definition* (DTDs) [Bray et al., 2008b], their extended version, and XML SCHEMAS [Fallside and Walmsley, 2004; Chidlovskii, 2000; Martens et al., 2006a]; all from the W3C consortium. Another standard is RELAXNG [van der Vlist, 2003] from the Oasis consortium. For a more detailed description of these schemas, the reader is redirected to [Murata, Lee, and Mani, 2001; Martens et al., 2006b; Schwentick, 2007; Comon, Dauchet, Gilleron, Löding, Jacquemard, Lugiez, Tison, and Tommasi, 2007b]. In the following we describe DTDs in more details and provide relations between the aforementioned schemas and regular tree languages. Indeed, in this thesis, we mainly study DTDs in the context of property testing. There are two reasons for that, the first reason is that DTDs are commonly used and the second is that they describe local restrictions. One should also note that all

the standard schemas can be modelled by extended DTDs.

Document type definition and extended DTD

A document type definition (DTD) is a set of restriction rules that defines a collection of XML documents. The restriction rules relate the label of any node in a valid XML document, with the word formed by the labels of the node sequence of children. We recall that XML documents are modelled by unranked trees. DTDs are the most commonly used schemas for XML [Bex et al., 2004]. Formally a DTD D over an alphabet Σ is any tuple $(\Sigma, \mathit{init}, \mathit{rul})$, where $\mathit{init} \in \Sigma$ is the start symbol and rul is a set of rules of the form $a \to e$, i.e, rul is a function mapping each symbol $a \in \Sigma$ to a regular expression e over Σ. In what follows we will denote by $D(a)$ the regular expression $\mathit{rul}(a)$, for $a \in \Sigma$. The W3C also recommends deterministic context models for DTDs, that is all regular expressions e in DTDs rules must be *deterministic*.

A DTD $D = (\Sigma, \mathit{init}, \mathit{rul})$ inductively defines for each letter $a \in \Sigma$, a tree language $L_a(D) \subseteq \mathcal{T}_\Sigma$ such that: $L_a(D)$ is the smallest set of trees satisfying

$$L_a(D) = \{a(t_1, \cdots, t_k) \mid t_i \in L_{a_i}(D), a_1 \cdots a_k \in L(\mathit{rul}(a)), \text{ for } 1 \leqslant i \leqslant k\}$$

The language of D-valid trees, for a DTD $D = (\Sigma, \mathit{init}, \mathit{rul})$, is the language of its initial state: $L(D) = L_{\mathit{init}}(D)$.

DTDs are less expressive than general tree automata. Indeed, in valid trees, the label of a node completely defines the labels of its children and this independently of the node context [Papakonstantinou and Vianu, 2000]. In fact the set of trees recognized by DTDs corresponds to the set of *local* tree languages [Murata, Lee, Mani, and Kawaguchi, 2005a]. Thus DTDs are less expressive than general regular tree languages. An example of DTD which validate the tree of Figure 3.3 is given Figure 3.4.

More powerful schemas was introduced by Papakonstantinou and Vianu [Papakonstantinou and Vianu, 2000] under the name *specialized* DTDs. Those schemas are now commonly called Extended DTDs (EDTDs). EDTDs enhance the power of DTDs by adding a typing mechanism. More formally an extended DTD $\mathcal{E}$ over an alphabet Σ is a tuple $\mathcal{E} = (\Sigma, Q, D, f)$, where Q is a finite set of states, D is a DTD over Q and $f : Q \to \Sigma$ is a function from Q to Σ. A tree $t \in \mathcal{T}_\Sigma$ is a member of the language of $\mathcal{E}$ if there is a tree $t_1 \in \mathcal{T}_Q$ such that $f(t_1) = t$; where $f(t_1)$ is obtained from t_1 by relabelling all nodes $w \in \mathit{nod}_{t_1}$ with $f(\mathit{lab}_{t_1}(w))$. EDTDs have the same expressive power than regular tree automata. As expressive power brings harder schema validation problems, restriction of EDTDs was also studied. A EDTD $\mathcal{E} = (\Sigma, Q, D, f)$ is *single-typed* if f is injective, that means there is no competitive types. In other words for $q, q' \in Q$: $f(q) = f(q')$ iff. $q = q'$. And q is a valid type for $a \in \Sigma$ if $f(q) = a$. Single-typed EDTD corresponds to XML SCHEMA according to [Martens, Neven, and Schwentick, 2005; Martens et al., 2006a]. Another restriction defines *restrained competition* EDTDs. A EDTD $\mathcal{E} = (\Sigma, Q, D, f)$ is

Standardized Syntax:

```
<!DOCTYPE collection[
  <!ELEMENT collection     (book*)>
  <!ELEMENT book           (title, author+, year)>
  <!ELEMENT title          (#PCDATA)>
  <!ELEMENT author         (#PCDATA)>
  <!ELEMENT year           (#PCDATA)>
]>
```

Syntax used in this thesis: (the initial element being collection)

$$\begin{array}{lcl} \text{collection} & \rightarrow & \text{book}^* \\ \text{book} & \rightarrow & \text{title} \cdot \text{author}^+ \cdot \text{year} \\ \text{title} & \rightarrow & \varepsilon \\ \text{author} & \rightarrow & \varepsilon \\ \text{year} & \rightarrow & \varepsilon \end{array}$$

Figure 3.4: A DTD for the collection of books of Figure 3.3. In the standardized syntax note that ',' is used for the concatenation in regular expressions. There is also the '#PCDATA' symbol which corresponds to textual content. However as we are only interested in the structure of XML documents, this symbol is replaced in our syntax by the empty word ε.

restrained competition, if for all different competitive types $q, q' \in Q$, there exists no words $w, w_1, w_2 \in Q^*$ and type $q_1 \in Q$ such that: both $w \cdot q \cdot w_1$ and $w \cdot q' \cdot w_1$ are members of $D'(q_1)$. *Restrained competition* EDTDs can be efficiently translated into top-down deterministic automaton on the fcns encodings of trees [Champavère, Gilleron, Lemay, and Niehren, 2009]. That is, for a restrained competition EDTD $\mathcal{E}$, the automata obtained with such translation recognizes exactly the fcns encodings of trees in the language $L(\mathcal{E})$.

XML SCHEMA

Against the expressiveness limitations of DTDs (locality, no typing mechanism), the W3C provides the XML SCHEMAs specification [Fallside and Walmsley, 2004]. In this XML-based schema language, one has a typing mechanism and more sophisticated constraints on the content of valid XML documents. For instance, one can specify: enumerated types, constraints about the number of occurrence of an element (*minOccurs, maxOccurs*), and also integrity constrains (*ref, unique, key*); for valid XML documents. When only the structural aspects of valid XML documents are considered, XML SCHEMAs have the expressibility of single-typed EDTDs. Thus XML

Schemas are less expressive than general tree automata and they denote tree languages whose fncs encodings can be recognized by top-down deterministic tree automata [Martens et al., 2005, 2006a].

RelaxNG

The RelaxNG schema language is an alternative to Xml Schema and is developed by the Oasis consortium [Clark and Murata, 2001]. RelaxNG has a syntax close to regular tree automata formalism; like edtds. It also has an xml-based syntax. RelaxNG is as expressive as general edtds and thus it is as powerful as general tree automata. Therefore in contrast with dtds and Xml Schema, in RelaxNG there is no requirements for deterministic content models and RelaxNG permits better support of unordered content models. However no determinism makes schema validation harder.

3.5.2 Schema validation

A Schema Validation problem consist in checking whether some xml document satisfies some Schema of some specified schema language. As one is generally concerned only about structural differences of xml documents, one assimilates xml documents with their tree models, and Schemas with tree automata. Thus a Schema Validation problem becomes a model checking problem i.e a membership problem. This way one can consider also approximate schema validation. In this approach, schema validation for all schemas specified in dtd, edtd, Xml Schema, and RelaxNG schema languages reduces to the membership problem of trees in languages specified by tree automata. This approach yields schema validation algorithms that read their input xml documents as trees and transform their input schemas into tree automata. Then the schema validation problem, is solved with model checking algorithms for trees with properties represented by tree automata. Since we already mentioned results about the membership problem of trees for properties denoted by tree automata (Section 3.4), we indicate in this section the complexity of transforming a schema into a tree automata. dtds and Xml Schema can be translated into restrained competition edtds in PTime; and restrained competition edtds can also be transformed into top-down deterministic automata for the binary encodings of trees [Martens et al., 2005, 2006a]. As RelaxNG have an automata-based syntax, they also translate to tree automata. Thus, using this approach, schema validation algorithms for all these schemas have almost the same complexity than the membership problem of trees for regular tree languages (one must add the cost of such transformations).

Using this approach for approximate schema validation yields one difficulty: we can not generally use binary encodings of trees as the distance is not always kept. However for the edit distance with moves, this is the

case and Magniez and de Rougemont [2007] could then use their tester for DTDs. And DTDs are the only schemas studied in approximate schema validation so far. However in the case of DTDs, which is the only schema we study in this thesis, one can efficiently transform such automata into hedge automata. This is what is done in Chapter 6 where we consider approximate membership of XML documents (or trees) for DTDs.

Yet, in practice, usually a more direct approach is used in schema validation tools. Indeed, in most object oriented languages, the approach consist at creating a binding between objects and types defined in the schema on which one wants to validate the input XML documents. Then the input document is read within SAX processor for example and using the defined binding one can validate the document. This is the approach in the JAXP implementation (in java) for example.

3.6 Alternatives to property testing

We recall that our objective in this thesis is to study relational structures (specially words and trees) in the context of property testing. And this way one aims to probabily and approximately decide properties of huge (XML) databases. However there are other approaches that were considered in computer science for designing algorithms that may probably and efficiently decide properties of huge databases. As such approaches one would like to mention the use of Streaming and Sketches. In such approach, huge databases are read in a streaming manner and one computes statistics (or sketches). Then sketches are used to approximately answer any query on huge databases with high probability. Sketches are intensively studied lately for estimating frequency moment of streaming data [Bhuvanagiri, Ganguly, Kesh, and Saha, 2006; Alon, Matias, and Szegedy, 1999], for answering biased quantile queries [Cormode, Korn, Muthukrishnan, and Srivastava, 2006] and aggregated queries over streaming data [Alon, Gibbons, Matias, and Szegedy, 2002; Bar-Yossef, Jayram, Kumar, Sivakumar, and Trevisan, 2002; Jayram, Kale, and Vee, 2007; Cormode and Garofalakis, 2007]. For more details on streaming and sketching, one is referred to the surveys by Muthukrishnan [2005, 2011] and Cormode, Garofalakis, Haas, and Jermaine [2012].

4 Our property testing framework

Contents

In the previous sections we introduced words, trees and graphs as the main objects we study. We explained how we study them as relational structures (their representations), under some relational vocabulary. Vocabulary which is then used to formally talk about properties of relational structures, by means of logical languages. Properties was then designated by sentences of logical languages or automata. Model checking was relaxed into property testing using distance measures. By this relaxation, we aim to design algorithms that approximately decide properties of words, trees and graphs; through random access to relational structures which represent their input objects (words, trees, graphs). Then we mentioned that the random access to structures is done using queries to an oracle which accesses a relational structure. Therefore, in property testing, the query complexity of algorithms strongly depends on which kind of queries are provided by the oracle. Thus to complete our framework, we detail in this chapter how we formally design oracles that access relational structures. A random access to a σ-structure Γ will be called a random object of Γ; where σ is a vocabulary. And to design random objects, we first separate the symbols of σ in three categories: the *size* vocabulary, the *deterministic* vocabulary and the *random* vocabulary. And an oracle access will be just a set of functions. Any symbol r of the size vocabulary corresponds to a function which returns the size of the relation r. The deterministic vocabulary is for boolean functions that tells whether a relation holds for a tuple of elements of the structure domain. And the

random vocabulary will "usually" correspond to a "uniform" generation of some element satisfying a relation. We detail this in the following.

4.1 Random objects of relational structures

Let σ be a relational vocabulary and Γ be a σ-structure, where Γ is the representation of some word, tree or graph. Furthermore we suppose that the vocabulary σ is the union of three vocabularies (not necessarily distinct). $\sigma = \sigma_{det} \cup \sigma_{size} \cup \sigma_{rand}$. We will require random objects for Γ to be able to answer: boolean queries about the relations with names in σ_{det}, size queries for relations in σ_{size}; and to be able to generate some elements of the domain of Γ which satisfy a relation in σ_{rand}, under some fixed distribution. In previous studies of property testing, the generation of elements is taken care by the algorithms (testers). However, in this thesis we prefer to make this generation part of the oracle access, so we can study which kind of randomization of words, trees or graphs is needed to have efficient testers. Before we formally define oracles access to a relational structure, we fist define the notion of randomization of structures.

We remind that for any set S, $S^0 = \{()\}$ is the set containing only the empty sequence. In what follows we sometime denote the empty sequence with $(e_1, \cdots, e_{k-1})$ for $k = 0$. In such case we abuse notation by identifying the sequence (e) with $(e_1, \cdots, e_{k-1}, e)$.

Definition 4.1. *Let σ be a relational vocabulary, Γ be a σ-structure and $\sigma_{rand} \subseteq \sigma$ be a subset of σ. A σ_{rand}-randomization of Γ is a pair $(\mathcal{D}, \perp)$ of a collection of distribution $\mathcal{D}$ with an element $\perp \notin dom(\Gamma)$ such that: for all relation symbols $r \in \sigma_{rand}$ of arity $k \in \mathbb{N}$, and for all sequence $s = (e_1, \cdots, e_{k-1})$ of $dom(\Gamma)^{k-1}$, there is a probability distribution $\mathcal{D}_{r,s} \in \mathcal{D}$ satisfying*

- *if $k = 1$ then $\mathcal{D}_{r,s}$ is a distribution over $R_s = \{e \mid r(e)\} \uplus \{\perp\}$, else*
- *if $k \geqslant 2$, then $\mathcal{D}_{r,s}$ is a distribution over $R_s = \{e \mid r(e_1, \cdots, e_{k-1}, e)\} \uplus \{\perp\}$,*
- *and furthermore, for all $k \in \mathbb{N}$, if $R_s \neq \{\perp\}$ then $\mathcal{D}_{r,s}(\perp) = 0$*

Moreover $\mathcal{D}$ contains only the distributions $\mathcal{D}_{r,s}$ which corresponds to some $r \in \sigma_{rand}$ and a sequence of elements $s = (e_1, \cdots, e_{k-1}) \in dom(\Gamma)^{k-1}$.

The randomisation $(\mathcal{D}, \perp)$ is called uniform if and only if the probability distributions $\mathcal{D}_{r,s}$ have the same probability on all elements of $R_s \backslash \{\perp\}$.

The randomisation is called f-weighted, where $f : dom(\Gamma) \to \mathbb{N}$, if for all $r \in \sigma_{rand}$ of arity $k \in \mathbb{N}$ and $s = (e_1, \cdots, e_{k-1}) \in dom(\Gamma)^{k-1}$ satisfying $R_s \neq \{\perp\}$, it holds for all $e_k \in R_s \backslash \{\perp\}$ that:

$$\mathcal{D}_{r,s}(e_k) = \frac{f(e_k)}{\sum_{e \in R_s \backslash \{\perp\}} f(e)}$$

For f-weighted randomizations of structures, notice that all probability distributions are determined by the same function. This kind of randomization will only be used for weighted words introduced in the last chapter for testing trees. The idea behind such randomization is that some elements of the structure domain are more important than others, and thus they must somehow be generated with higher probability by random objects which we define below. Note also that a randomization is a set of distributions $\mathcal{D}_{r,s}$ indexed by relations $r \in \sigma_{rand}$ of arity $k \in \mathbb{N}$ and sequence of elements $(e_1, \cdots, e_{k-1})$ of $dom(\Gamma)^{k-1}$. Randomization of structures are used below to design random objects. These distributions specify for a random object, the probability of selecting an element e such that $r(e_1, \cdots, e_{k-1}, e)$; when we use a random query to the random object with parameter r, $e_1, \cdots, e_{k-1}$. Below we define random objects of structures using random functions of randomizations of relational structures. We recall that the random function of an indexed collection of distributions $S = \{p_e \mid e \in S'\}$ is the function which inputs elements of S' and returns the random choice $X \sim p_e$ (see Section 2.3 for more details).

Definition 4.2. *Let $\sigma = \sigma_{det} \cup \sigma_{size} \cup \sigma_{rand}$ be a vocabulary, Γ a σ-structure and $(\mathcal{D}, \perp)$ a σ_{rand}-randomization of Γ. A $(\sigma_{det}, \sigma_{size}, \sigma_{rand}, \mathcal{D}, \perp)$-random object, for a σ-structure Γ, is a collection $rdm_\Gamma = \mathcal{B} \cup \mathcal{S} \cup \{\mathcal{R}\}$ of three sets of boolean functions ($\mathcal{B}$), integer valued functions ($\mathcal{S}$) and a random function ($\mathcal{R}$) such that:*

- *$\perp \notin dom(\Gamma)$*
- *for all relations $r \in \sigma_{det}$ of arity $k \in \mathbb{N}$, there is a boolean function $\mathcal{B}_r : dom(\Gamma)^k \to \mathbb{B}$ in $\mathcal{B}$ satisfying for all tuples $(e_1, \cdots, e_k) \in dom(\Gamma)^k$, $\mathcal{B}_r(e_1, \cdots, e_k) = true$ iff. $r(e_1, \cdots, e_k)$ holds.*
- *for all relations $r \in \sigma_{size}$ of arity $k \in \mathbb{N}$, there is an integer valued function $\mathcal{S}_r : dom(\Gamma)^{k-1} \to \mathbb{N}_0$ satisfying for all $(e_1, \cdots, e_{k-1}) \in dom(\Gamma)^{k-1}$, $\mathcal{B}_r(e_1, \cdots, e_{k-1}) = |\{e \mid r(e_1, \cdots, e_{k-1}, e)\}|$.*
- *$\mathcal{R}$ is the random function of the σ_{rand}-randomization $\mathcal{D}$.*

$\mathcal{B}$ is called the deterministic access to Γ, $\mathcal{S}$ is called the size access to Γ and $\mathcal{R}$ is the random access to Γ.

From this definition, Note that a $(\sigma_{det}, \sigma_{size}, \sigma_{rand}, \mathcal{D}, \perp)$-random object for a structure Γ have queries that can tell whether $r(e_1, \cdots, e_k)$ holds, for relations $r \in \sigma_{det}$ and elements $e_1, \cdots, e_k$ of the domain of Γ. This can be done using the function $\mathcal{B}_r$. And such random objects can also generate elements from the domain of Γ; it suffices to use the random function $\mathcal{R}$. In what follows we will use the same notations for the random, deterministic and size accesses to all structures. A $(\sigma_{det}, \sigma_{size}, \sigma_{rand}, \mathcal{D}, \perp)$-random object

is called *oblivious* if σ_{size} is empty. Generally we will only consider random objects with a uniform σ_{rand}-randomization, thus when this is the case we will ease up notations and talk only about $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random objects. And when also the vocabularies are fixed, we omit them.

Our objective in designing random objects this way is to provide testers that work fine independently of the relational structure which is the representation of a word or tree. In other words our aim is designing testers for all relational structures isomorphic to words, trees and graphs. In property testing previous works, the representation of words is an array of positions, and testers use the fact that positions are represented by integers so they can randomly draw positions and get the label of any position. In here we do not require any kind of restriction about how a position should be designated (they should just be distinguishable). Instead we require that the oracle, used by testers, can answer queries about those positions and also uniformly generate a position. Therefore in our framework, we may ask which kind of queries are required for a property to be testable? We do not provide a complete answer to this question, but instead we provide oracle access to words, trees and graph structures that allow to design testers for such structures. So we only provide sufficient queries to oracles in order that some properties (like regular word languages) became testable. We believe using our framework, we will be able to formally talk about properties that can be tested independently of the relational structure representing an object (trees, words, graphs).

Next we define precisely the random objects we use for words, trees and graphs. We will show that nearly all queries that can be made when a word is represented by an array can also be made by oracles accessing words. This allows us to discuss property testing of words using their representations by arrays, with the guaranty that all obtained results also apply for any relational structure isomorphic to some word and its appropriate random object.

4.2 Random objects of words

In the previous chapter, Section 3.1.1, we explained how a word $w \in \Sigma^*$ over the alphabet Σ defines (or can be represented) by the σ_{words}-structure S_w; where $\sigma_{words} = \{(lab_a, 1) \mid a \in \Sigma\} \uplus \{(<, 2), (succ, 2), (start, 1)\}$ and $dom(S_w) = dom(w)$. Then we emphasised the fact that we want testers designed in this thesis to work on any input structures isomorphic to S_w. Usually in property testing, the word w is represented by an array which stores the labels at any position. Then the positions of such array corresponds to the elements of $pos(w)$, and testers can uniformly generate positions and query their labels. While, for simplicity, we will continue to represent words with arrays in the next chapter (which presents our results about approximate

membership for regular words languages), in this section we present oracles that can answer all queries that the array representation provide. This way it should be clear that algorithms presented in next chapter works fine for any representation (i.e relational σ_{words}-structure isomorphic to S_w) of w. We present two oracles that only differs on their size access vocabularies. One oblivious oracle access and another with a none empty size vocabulary.

The deterministic vocabulary for word structures is:

$$\sigma_{det} = \sigma_{words}$$

and the random vocabulary is:

$$\sigma_{rand} = \{<^2, succ^2, start^1\}$$

Unless for weighted words (see Chapter 6 Section 6.5), we consider only uniform σ_{rand}-randomizations of word structures, thus we will omit to mention them in the definitions of the oracle accesses of word structures. Considering uniform randomizations is essentially what is done in property testing, as uniform distributions are easily implementable.

4.2.1 Oblivious random objects

Now we explain queries answered by $(\sigma_{det}, \varnothing, \sigma_{rand})$-random objects of word structures. And from the reader should have an intuition of which kind of words properties can be efficiently tested using oblivious random objects of word structures. Let Γ be a σ-structure isomorphic to some word $w \in \Sigma^*$; where Σ is some alphabet. Thus each element of the domain of w correspond to some unique element of Γ and vice versa. We recall that 0 is an element of the domain of w but is not a position, and Γ is called a representation of w. To have the element of Γ which corresponds 0, we can ask the query $\mathcal{R}(start)$ to the $(\sigma_{det}, \varnothing, \sigma_{rand})$-random object rdm_Γ of Γ. Below are other example of queries with what they correspond in w.

- $\mathcal{R}(start)$: Selects 0 in w.
- $\mathcal{R}(<, \mathcal{R}(start))$: Uniformly select a position of w, or $\perp$ if w is the empty word.
- $\mathcal{R}(succ, e)$: Select $i + 1$; where i is the position of w corresponding to the element $e \in dom(\Gamma)$.
- $\mathcal{R}(<, e)$: Select uniformly an position $i' > i$, where i is the position of w corresponding to the element $e \in dom(\Gamma)$; or returns $\perp$ if such position does not exist.
- $\mathcal{B}_<(e, e')$: Tells whether e' corresponds to $i + 1$; where i is the position of w corresponding to the element $e \in dom(\Gamma)$.

- $\mathcal{B}_{lab_a}(e)$: Tells whether i is labelled $a \in \Sigma$; here i is the position of w corresponding to the element $e \in dom(\Gamma)$;

However there are some information that can not be queried efficiently with the oblivious random object of Γ. Consider, for example, the question of knowing whether some element e of $dom(\Gamma)$ corresponds to the i^{th} position of w (we suppose here that $|w| > i$ and $i \in \mathbb{N}$). This can only be done with an oblivious random objects by first querying the element e' which corresponds to the i^{th} position of w and then checking whether $e = e'$. And querying e' can only be done by starting from the start element and using i times the query $\mathcal{R}(<, .)$. Thus this will require many queries to the random object. Indeed this is because with oblivious random objects, it is difficult to know which is the position corresponding to some generated element of the domain of Γ. Yet in the context of property testing with the Hamming distance, this is usually required (see next chapter). So, under the Hamming distance, the approximate membership problem for word regular languages is generally hard when we are allowed to use only oblivious random objects. In the next section, we will see that this difficulties disappear with non-oblivious random objects of word structures.

4.2.2 Random objects with size access

For non-oblivious oracle accesses, we consider along with the above mentioned vocabularies, a size vocabulary:

$$\sigma_{size} = \{(<, 2)\}$$

Let Γ be a σ-structure isomorphic to some word $w \in \Sigma^*$; where Σ is some alphabet. Essentially what non-oblivious $(\sigma_{det}, \sigma_{rand}, \sigma_{rand})$-random objects rdm_Γ add to oblivious ones (see above) is the capacity of knowing for any element of the domain of Γ, the position of w corresponding to it. Below we give example of queries we can ask to rdm_Γ, and explain their correspondent information about w.

- $\mathcal{S}_<(\mathcal{R}(start))$: This returns the size of w.
- $\mathcal{S}_<(\mathcal{R}(start)) - \mathcal{S}_<(\mathcal{R}(e))$: This returns the position corresponding to $e \in dom(\Gamma)$ in w.
- $\mathcal{S}_<(\mathcal{R}(e')) - \mathcal{S}_<(\mathcal{R}(e))$: This returns the number of positions between i and i'; where i and i' are the positions in w which correspond to $e, e' \in dom(\Gamma)$.

Like oblivious random objects of word structures, it is still hard to query the element of Γ which corresponds to the i^{th} position of w. However using

non-oblivious random objects of words so defined, we can design efficient testers for the approximate membership problem for words regular languages; under the Hamming distance (see next chapter).

Notice that random objects are defined for a representation Γ of w under some fixed isomorphism $\theta : \Gamma \to S_w$ which makes correspondence between elements of the domain of Γ and positions in w. We will not mention this isomorphism; and abusing notations, the random objects of all representations isomorphic to some word w will simply be denoted by rdm_w.

4.3 Random objects of trees

Let Σ be an alphabet and $\mathrm{t} \in \mathcal{T}_\Sigma$ a tree over Σ. We recall that a representation of t is any σ_{tree}-structure Γ such that there exists an isomorphism $\theta : dom(\Gamma) \to nod_{\mathrm{t}}$ from Γ to S_{t}. Where the signature of tree structures is:

$$\sigma_{trees} = \{lab_a^1 \mid a \in \Sigma\} \uplus \{fc^2, ns^2, parent^2, desc^2, root^1, anc^2, <^2\}$$

and the structure S_t have domain nod_t (see Section 3.1.1).

Random objects for tree structures Γ are designed with the following alphabets. The deterministic alphabet is exactly the whole signature, i.e

$$\sigma_{det} = \sigma_{trees}$$

The size alphabet is:

$$\sigma_{size} = \{anc^2\}$$

and the random alphabet is:

$$\sigma_{rand} = \{fc^2, ns^2, parent^2, desc^2, root^1\}$$

Thus in this thesis we consider only $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random object of tree structures (in Chapter 6). As we have done with words we now give examples of queries answered by the random objects of Γ: rdm_Γ. We give what they correspond to the tree t and lately we will simply denote random objects of all tree structure isomorphic to S_t simply by rdm_t.

- $\mathcal{R}(root)$: This returns the root of t (or e such that $\theta(e) = \varepsilon$).
- $\mathcal{R}(desc)$: Generates uniformly a descendent of $\theta(e)$.
- $\mathcal{R}(fc, e)$: Returns $\theta(e) \cdot 1$ if $\theta(e)$ has a child or $\perp$.
- $\mathcal{B}_{parent}(e, e')$: Tells whether $\theta(e')$ is a parent of $\theta(e)$.
- $\mathcal{B}_{<}(e, e')$: Tells whether $\theta(e')$ after $\theta(e)$ in the preorder relation.
- $\mathcal{S}_{anc}(e)$: returns the depth of the node $\theta(e)$.

Note that a random object rdm_t does not efficiently generate a child of some node e. In order to generate a child of e, we can uniformly generate a descendent e' of e and then using repetitively the query $\mathcal{R}(parent, .)$ until we reach e. However the number of necessary queries can be equal to $d(t)$.

We recall that in approximate model checking (or property testing), we want to detect farness of tree structures from some properties while using only local inspections of its domain. These local inspections are performed in our framework using the notion of random objects, which are defines using the vocabulary of tree structures. We might want to use a smaller set of relations in the vocabulary of trees. However as the vocabulary impose in our framework the set of relations efficiently accessible in our structures, this will affect the efficiency of testers. Indeed, which relations are used in the signature of trees random objects are important depending on the distance measure for which approximate model checking is considered. For instance the preorder relation is important for the standard and strong edit distances, whereas it is less important for the edit distance with moves. The reason is that the move operation can be used to rearrange subtrees at low cost, whereas for tree edit distances without moves such rearrangement become expensive and thus, under edit distances without moves, to efficiently detect farness of tree structures from a property, knowing the order of nodes in the whole tree structure helps. We can restated that in the following way, using move operations nearly allow to nearly ignore the preorder relation of nodes.

In the previous sections we have seen that random objects for words might be implemented using an array representing the word. With trees, the main difficulty to implement an oracle access in to efficiently answer the uniform generation of a descendent of any node. In the following we show how this can be done when trees are stored in relational databases or in XML databases.

4.3.1 Random objects in relational databases

Storing a tree t in relational databases can be done by storing each node as a tuple composed by its label with the numbers of its opening and closing tag in a streaming traversal of t. The well known downfall of this representation is that inserting a new node to the tree may require modifying the closing and opening tag number of an important set of nodes (in the worst case, this implies modifying the numbers of all nodes). We concretely explain this with the tree of Figure 4.1, which represents a collection of books. We represent in a table all nodes with identifiers and their tuples above discussed.

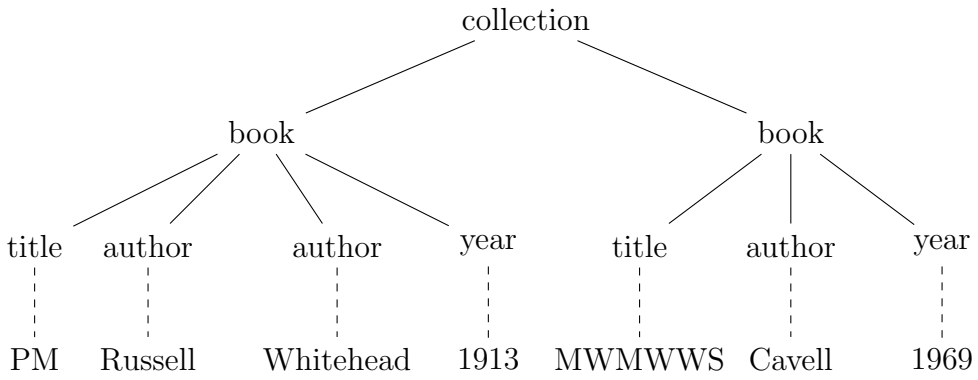


Figure 4.1: A tree example

Id	*Label*	*Op* (*opening*)	*Cl* (*closing*)
id1	collection	0	33
id2	book	1	18
id3	title	2	5
id4	PM	3	4
id5	author	6	9
id6	Russel	7	8
id7	author	10	13
id8	Whitehead	11	12
id9	year	14	15
id10	1913	16	17
id11	book	19	32
id12	title	20	23
id13	MWMWWS	21	22
id14	author	24	27
id15	Cavell	25	26
id16	year	28	31
id17	1969	29	30

Hence using simple SQL queries, any relational database could efficiently implement our random object for trees. For example for finding the root of the tree, we have to select the identifier of the node such the opening number (cl) is 0, (i.e *select identifier from Tree where op = 0*). The depth of a node with identifier *id*, can be obtained by:

```
Select count(identifier)
From Tree
Where identifier = id
      and cl < (Select cl From Tree Where identifier = id)
      and op > (Select op From Tree Where identifier = id)
```

And to uniformly generate a descendent of a node with identifier *id*, we use the query:

```
Select identifier
From Tree
Where identifier = id
      and cl > (Select cl From Tree Where identifier = id)
      and op < (Select op From Tree Where identifier = id)
```

and then uniformly return one of the results. Note also that any other query of the random oracle can efficiently be answered by a database storing trees this way.

4.3.2 Random objects in xml databases

In XML databases, the situation is rather simpler because trees are stored as XML documents. Thus to implement random objects, it suffices to guaranty (usually this is the case) that any node has an identifier and then translate any query to the random object into a XPath query to the database. We consider again the XML document of Figure 3.3 which represents a collection of books. We add identifiers as in the previous section.

```
<collection identifier="id1">
  <book identifier="id2">
    <title identifier="id3">Principia Mathematica</title>
    <author identifier="id4">Russell</author>
    <author identifier="id5">Whitehead</author>
    <year identifier="id5">1913</year>
  </book>
  <book identifier="id6">
    <title identifier="id7">M.W.M.W.W.S</title>
    <author identifier="id8">Cavell</author>
    <year identifier="id9">1969</year>
  </book>
</collection>
```

And for example, the root node can be obtained with the expression:

$$/@\text{identifier}$$

and the descendants of a node with identifier *id* can be queried with:

$$//*[./@\text{identifier} = \text{id}]/\textit{descendent} :: */@\text{identifier}$$

then for a descendent of the node with identifier *id* can then be uniformly selected. Using the *count* function of XPath, the depth of any tree can also be efficiently queried.

4.4 Random objects of graphs

We recall that a graph might be seen as a relational structure over the vocabulary:

$$\sigma_{graphs} = \{node^1, edge^2\}$$

We give an oblivious and non oblivious random object of graphs using this signature.

4.4.1 Oblivious random objects

We consider the deterministic vocabulary to be:

$$\sigma_{det} = \{edge^2\}$$

the size vocabulary is empty:

$$\sigma_{size} = \varnothing$$

and the random vocabulary is:

$$\sigma_{rand} = \{node^1\}$$

Thus with an oblivious $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random objects, we can only uniformly query a node and look the subgraph induced by the queried nodes, and this without knowing the size of the graph.

4.4.2 Non-oblivious random objects

To allow graphs random objects to query the size of the graph they represent, we consider the same random and deterministic vocabularies as for oblivious (i.e $\sigma_{det} = \{edge^2\}$ and $\sigma_{rand} = \{node^1\}$), and a size vocabulary which contains only the relation node.

$$\sigma_{size} = \{node^1\}$$

Hence non-oblivious random objects add only the possibility to query the size of the overall graph. We query the size of a graph structure through its random object with $\mathcal{S}_{node}()$.

Approximate membership testing of graphs was intensively studied since the seminal paper of Goldreich, Goldwasser and Ron [Goldreich et al., 1998]. Two models were proposed and mainly those models differ in the kind of oracle queries they allow on the graph representation. Most important is that those models are more or less appropriate depending on whether the considered graphs are dense or with bounded adjacency degree (or simply

sparse). Indeed it is appropriate to represent dense graphs by an *adjacency-matrices*, whereas for sparse graphs an *adjacency-lists* representation is more appropriate. And it was shown that some properties (like bipartiteness) are testable in one model and not in the other (see [Ron, 2000]). We do not really study property testing of graphs in this thesis, however we have provided in this section random objects of graphs. It would be interesting as future work to compare our framework with the aforementioned ones. One straightforward fact is that every property which can be tested in the dense model will also be testable when we provide a random graph object to the tester. This is due to the fact that queries used in the dense model can be answered by random objects of graphs. However in the adjacency-list model, nodes adjacent to some node v, are ordered and this order is not part of our vocabulary for graphs structures. Therefore testers that might benefit from such order can not be efficiently translated in our model.

5 Efficient tester for word regular languages

Contents

5.1 Introduction

In the previous chapters we have formally defined words, trees and graphs and stated our interest in approximately studying their properties in the area of property testing which we also called approximate membership testing. We have represented words, trees and graphs as relational structures and we also defined random objects as oracles that can randomly access relational structures. In the case of words, we have discussed how non-oblivious random objects for some word w can essentially be seen just as an array containing w. Therefore in this chapter, where we study property testing

for word regular languages, we can assimilate word random objects with arrays containing words (without lost of generality, and for simplicity of the presentation). Thus in all our proofs we replace random objects with array representations of words. However we will write all our algorithms and theorems using random objects, whereas in our proofs sometimes we might talk about words as represented by arrays.

The area of property testing was initiated by Rubinfeld and Sudan [1996] for program checking. Then Goldreich, Goldwasser, and Ron [1998] were the first to apply property testing to study properties of combinatorial structures; more precisely they studied graph properties. More recently, property testing was applied to study properties of hypergraphs [Newman and Sohler, 2011], boolean functions [Alon and Shapira, 2002; Ron et al., 2011], and geometric functions; see surveys of Fischer [2001], Goldreich [1998], and Ron [2008] for details. In approximate membership testing of words [Alon et al., 2000b; Newman, 2002; Fischer et al., 2006], one wants to test whether some word belongs to a regular language or to some language in another class.

Approximate membership testing for non-deterministic finite automata (NFAs) is the following problem. Given a word w, a precision value ϵ and an NFA A, the problem is to discover nonmembership to the language of A if the word w is ϵ-far from it. Intuitively, being ϵ-far from a language means that the correction of an ϵ-fraction of w is insufficient to turn it into a member of the language. Which corrections are permitted depends on the distance notion on words that is chosen. So far, the Hamming distance and the edit distance with moves were considered, but one can also choose the usual edit distance without moves. Since these distances can be ordered decreasingly, any approximate membership tester for the Hamming distance also applies to the edit distance, and any approximate tester for the edit distance can be used for the edit distance with moves.

Membership testers whose efficiency does not depend on the size of the input word are most relevant for processing large texts. It means that the tester needs only to inspect a fragment of constant size of any input word when fixing the error precision and the NFA. In order to provide access to the word's letters, without having to read the word entirely, the tester inputs a random object of some σ_{words}-structure isomorphic to S_w, where w is the word being tested (see Section 3.3.2 and Section 4.2). And as already mentioned above (see Section 4.2), such random object can be seen as a reference to an array that contains the word to test, together with its length. In this way, all positions of the word can be drawn from a uniform distribution.

Exact membership testing can be sped up by preprocessing with an approximate membership tester, since whenever the latter returns NO, the exact tester can adopt this decision with high probability (or even precisely for one-sided testers) without having to read the entire word. When testing membership for a collection of words, it may thus be sufficient to read only few of them entirely.

Alon, Krivelevich, Newman, and Szegedy [2000b] showed that approximate membership testing with constant query complexity is indeed possible for regular languages modulo the Hamming distance, for fixed automata and error precision. As argued above, their algorithm can equally be used to test approximate membership for larger distances, such as the edit distance or the edit distance with moves. The edit distance with moves is the natural distance notion obtained if considering words as directed graphs with deletion and addition of edges but may be too relaxed for some applications. Fischer, Magniez, and de Rougemont [2006] proposed another approximation algorithm for regular languages modulo the edit distance with moves, which may be exponential in the inverse of the error precision though.

The state of the art leaves two main problems open. First, all existing testers may require exponential time in the size of the input automaton or the inverse error precision. It is unknown whether polynomial time algorithms exist. Second, input automata are assumed to be deterministic, which may induce another exponential blow up for determinization. The question is whether there exists an algorithm that can also deal with nondeterministic automata in polynomial time. For these two reasons, the previous algorithms fail to be efficient which limits their potential for use in practice.

Our first contribution is an approximate membership tester for NFAs modulo the Hamming distance, which runs in polynomial time depending on the size of the input NFA and the inverse error precision, independently of the size of the input word. The new algorithm is obtained by reformulating Alon, Krivelevich and Newmann's algorithm [Alon et al., 2000b], based on the notion of infeasible fragments of words that we introduce. In order to decide feasability, we have to decide k-step reachability of NFA states in polynomial time, but independently of k. As we show, this is indeed possible under the assumption that addition and multiplication on natural numbers can be done in constant time. Here we apply ideas from algorithms for computing the Chrobak normal form of NFAs over a single letter alphabet [Gawrychowski, 2011].

As our second contribution, we show that approximate membership for NFAs modulo the edit distance can even be tested more efficiently. Our new tester is based on the notion of blocking fragments that we introduce. In order to decide whether a fragment is blocking, we have to decide reachability for NFA states. The reason for which we can relax k-reachability as for the Hamming distance is that errors can be corrected by letter insertion for the edit distance. Reachability can be decided in linear time in the size of the NFA, and thus more efficiently than k-step reachability. Therefore, the degree of the polynomial for the complexity of approximate membership testing can be reduced by 3. We summarize the new state of the art including our results in Figure 5.1.

precision ϵ, NFA A over Σ with k strongly connected components, inverse precision $\delta = 1/\epsilon$, polynomially bounded functions: $p^{i,j,l}(\delta, k, \lvert A\rvert) =_{\text{def}} \delta k^2 \lvert A\rvert^i \log^j(\delta k \lvert A\rvert^l)$		
Distance	Query complexity	Time complexity
Hamming distance [Alon et al., 2000b]	$O(p^{2,3,2}(\delta, k, \lvert A\rvert))$	$O(2^{\lvert A\rvert^2} + \delta^k)$
Hamming distance [Ndione et al., 2013]	$O(p^{2,3,2}(\delta, k, \lvert A\rvert))$	$O(p^{5,3,2}(\delta, k, \lvert A\rvert))$
Edit distance [Ndione et al., 2013]	$O(p^{1,3,1}(\delta, k, \lvert A\rvert))$	$O(p^{2,3,1}(\delta, k, \lvert A\rvert))$
Edit distance with moves [Fischer et al., 2006]	$\ln \lvert\Sigma\rvert \ \lvert\Sigma\rvert^{2\delta} \ \delta^4$	$O(\ln \lvert\Sigma\rvert \ \lvert\Sigma\rvert^{2\delta} \ \delta^4 + \lvert A\rvert^{O(\delta)})$

Figure 5.1: Summary of results on approximate membership testing for regular word languages.

Outline In Section 5.2 we start with preliminaries on approximate membership testing. Section 5.3 introduces informally the notions of infeasible and blocking fragments of words and illustrates their relevance for approximate membership testing by example. Section 5.4 presents the formal definitions of blocking and infeasible fragments and presents polynomial time algorithms to decide these properties. We also discuss in this section why one can restrict the discussion to array reference of words. The next two sections present our approximate membership tester for NFAs modulo the edit distance. In Section 5.5, we treat the case of strongly connected NFAs based on the notion of blocking intervals, and in Section 5.6, we generalize this algorithm to arbitrary NFAs with multiple strongly connected components, while relying on blocking fragments. In Section 5.7, we sketch an analogous approximate membership tester for NFAs modulo the Hamming distance, in which the notion of feasible fragments becomes essential.

5.2 Preliminaries

We have recalled preliminaries on words and finite automata in Sections 2.1.2 and 3.4.1 respectively. In particular, we introduced the notions of a fragment (and interval) of a word. We have formally defined distance notions in Section 2.2, and specially we defined three edit distances between words, namely: the Hamming distance (d_h), the edit distance (d_l) and the edit distance with moves (d_m). In this section, we recall these distances and approximate membership testing for regular word languages represented by NFAs. We also show how to run a finite automaton on a fragment, using

random objects of word structures or simply an array representing the word contain the fragment. Fragments will be used later on to witness ϵ-farness.

We recall that a distance between words with alphabet Σ is a binary real-valued function $d \subseteq \Sigma^* \times \Sigma^* \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$, and a word language L is just a subset of Σ^*. We had extended any distance notion d between words into a distance between a word w and any language L in this way:

$$d(w, L) = \inf\{d(w, w') \mid w' \in L\}$$

For the empty language, the distance is infinite. A word w is called ϵ-far from L with respect to the distance if $d(w, L) > \epsilon|w|$, and ϵ-close otherwise. Note that ϵ-farness and ϵ-closeness are defined with respect to the relative distance normalized by the length of the word. we have already discussed in Section 3.3.2 why it is important to use such normalization if one wants to efficiently test approximate membership of words to some language.

The Hamming distance $d_h(w, w')$ between two words $w = a_1 \dots a_n$ and $w' = a'_1 \dots a'_m$ is the least number of letter-exchange operations by which the two words become equal. For words of the same length $m = n$, this is the number of positions i such that $a_i \neq a'_i$, and for words of different length, this is ∞. The edit distance $d_l(w, w')$ between two words w and w' is the least number of insertion, deletion, and letter-exchange operations needed to transform w into w'. For instance, $d_h(001100, 110011) = 6$ and $d_l(001100, 110011) = 4$ since we can insert 11 at the beginning of the left word and delete 00 at its end. Clearly the edit distance between w and w' is always smaller or equal than their Hamming distance. We provide in this chapter testers for the approximate membership to regular languages defined by NFAs. As already discussed these testers also are valid for the edit distance with moves.

5.2.1 Random objects of words

We recall that in Section 3.1.1, we explained how a word $w \in \Sigma^*$ with alphabet Σ can be seen as the σ_{words}-structure S_w; where

$$\sigma_{words} = \{(lab_a, 1) \mid a \in \Sigma\} \uplus \{(<, 2), (succ, 2), (start, 1)\}$$

Then, in section 4.2, we defined oblivious $(\sigma_{det}, \varnothing, \sigma_{rand})$-random objects and non-oblivious $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random objects that provide accesses (queries) to any representation of w (i.e any σ_{words}-structure isomorphic to S_w). Where $\sigma_{det} = \sigma_{words}$, $\sigma_{rand} = \{<^2, succ^2, start^2\}$ and $\sigma_{size} = \{(<, 2)\}$. We discussed queries provided by random objects of words and explained that non-oblivious random objects can intuitively be assimilated with an array containing a word.

In this Chapter we consider property testing of regular words languages using non-oblivious random objects. Next we recall approximate membership

in the special case of word properties denoted by NFAs. We will simply denote random objects of some word $w \in \Sigma^*$ by rdm_w; independently of which representation is considered.

5.2.2 Approximate membership of regular word languages

We recall the problem of approximate membership testing to regular languages defined by NFAs modulo some distance.

Definition 5.1. *An approximate membership tester for* NFA*s A with alphabet Σ and a distance function $d : \Sigma^* \times \Sigma^* \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ is a terminating probabilistic algorithm that reads as input a precision value $\epsilon \in \mathbb{R}_{>0}$, and a random object rdm_w to a word $w \in \Sigma^*$ and outputs* CLOSE *or* NO *such that:*

- *if $w \in L(A)$ then with probability $\frac{2}{3}$ it outputs* CLOSE*, and*
- *if $d(w, L(A)) > \epsilon|w|$ then with probability $\frac{2}{3}$ the output is* NO*.*

When ϵ-close non-members of the language are received as input, approximate membership testers are allowed to answer both CLOSE or NO without any particular requirement. Given a tester M, we can obtain another tester M' with higher probability than $2/3$ by repeating M sufficiently often. Also note that all testers presented in the chapter will be one-sided in that they will always answer CLOSE for all words of the language. Hence, NO-answers of one-sided testers are always correct (not only with high probability).

Since the edit distance between w and w' is always smaller or equal than their Hamming distance, any approximate membership tester modulo the Hamming distance will also be an approximate membership tester modulo the edit distance.

The *query complexity* of a tester is the maximal number of queries to the input random object it may make until termination. We are particularly interested in testers with constant query complexity in the size of the input word, so that the number of read operations may only depend on the NFA and the error precision. The *time complexity* of a tester is the maximal number of computation steps it might require until termination. Here, we assume that all arithmetic operations, take time in $O(1)$.

5.3 Examples

We illustrate informally how to witness farness with respect to the Hamming distance or to the edit distance by large collections of small "infeasible" or respectively "blocking" fragments. The existence of such witnesses can then be tested by probabilistic algorithms. The precise definitions will be given in the next section.

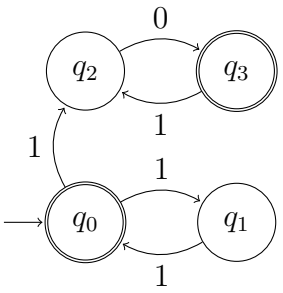


Figure 5.2: An NFA for $L = (11)^*(10)^*$.

We consider the regular language $L = (11)^*(10)^*$ over the alphabet $\Sigma = \{0, 1\}$. This language is recognized by the NFA with 4 states in Figure 5.2. This NFA has $k = 2$ strongly connected components, namely $\{q_0, q_1\}$ and $\{q_2, q_3\}$ which correspond to the two Kleene star operators in the definition of L.

5.3.1 Intervals

We consider words $u_m = (01)^m$ with $m > 0$ which do not belong to L. They can be corrected, however, to become a member of L by flipping all 0s to 1. It is apparent that corrections with fewer relabeling operations do not exist, so the Hamming distance of u_m to L is equal to m. In other words, u_m is at least $1/2$-far from L with respect to the Hamming distance. With only 2 insertion operations, however, we can correct u_m so that it belongs to L, by adding a 1 in front and a 0 at the end of u_m. Therefore the edit distance of u_m to L is at most 2, so u_m is $1/m$-close to L for the edit distance.

We next present a collection of intervals that witnesses for the Hamming distance that u_m is far from L. We consider the intervals $]i, i+2]$ of u_m where $0 \leqslant i < 2m - 1$ and i is even. These intervals are infeasible for the NFA in the following sense: First note that the factor of u_m at interval $]i, i + 2]$ is equal to 01. Words of even length can reach states q_0 and q_3 only. When proceeding from there with word 01 all possible runs of the NFA get stuck. This shows that no word with factor 01 at any even position may belong to the language. Hence, at least 50% of all intervals of length 2 are infeasible for this NFA. An analogous argument for odd positions shows that all of them are infeasible (but a fixed percentage will be enough for our algorithm).

When it comes to farness for the edit distance, the precise position of an interval does not matter but only the factor that it defines. The reason is that new letters can be inserted for correction before or after the factor. Therefore, the notion of infeasible intervals can be weakened to the notion of blocking intervals. These are intervals that do permit to run the NFA under consideration. Indeed, none of the intervals $]i, i + 2]$ of u_m is blocking in this sense. If i is odd, then the NFA can be run on the i-factor 10 from states q_0

or q_3 – even though no word of odd length can reach these states – and if i is even, the NFA can be run on the i-factor 01 from states q_1 and q_2 – even though no word of even length will end up there.

5.3.2 Fragments

We next show that collections of intervals are not always sufficient to witness farness. In order to see this, we consider the words $v_m = (10)^m(11)^m$ with $m > 0$. Again, no v_m belongs to L but all of them can be corrected by flipping 0s to 1. There is no better way even not if permitting insertion and deletion operations. Therefore, v_m is 1/4-far from L both with respect to the edit and the Hamming distance. Notice that $]2m-1, 2m+1]$ is the only blocking or infeasible interval of v_m, hence most of the 'small' collections of intervals of length 2 doesn't witness 1/4-farness of v_m.

We now consider the fragments $]i, i+2] \cup]j, j+2]$ where $0 \leqslant i < 2m-1 \leqslant j < 4m-1$. The pair of factors of v_m at these fragments are all equal to $(10, 11)$, and thus blocked in the following sense: Before reading the first factor arbitrary states are reachable, but when continuing with the first factor 10, only state q_3 can be reached. From there, we can reach states q_2 and q_3 over arbitrary words, given that we do not make any assumptions on the word between the two factors (even not on its length). From there, if we try to continue with the second factor 11, we get blocked in state q_2. This shows that all above fragments are blocking, and they constitute roughly 1/4th of all fragments with 2 intervals of length 2.

5.3.3 Sampling algorithms

All our algorithms will be based on probabilistic sampling for testing whether a large fraction of "short" fragments is blocking or infeasible. Most typically, we might want to test whether a word contains an ϵ-fraction of blocking fragments of some fixed length, say 2 for instance. This can be tested by a probabilistic algorithm as follows: parametrized by a positive real c, it draws randomly c/ϵ fragments of length 2 of the input word from a uniform distribution, and answers NO if at least one of them is blocking, and YES otherwise. This algorithm will detect some blocking fragment with probability at least $1-(1-\epsilon)^{c/\epsilon} \geqslant 1- \ e^c \xrightarrow{c\to\infty} 1$. Therefore, it answers correctly with NO with high probability, for instance, with probability 0.9999 if we choose a parameter $c \geqslant \ln(0.0001)$.

This kind of sampling algorithm can be used with $\epsilon = 1/4$, for instance, in order to show that v_m is far from L modulo the edit distance. It can also be applied to show that u_m is far from L modulo the Hamming distance, by considering infeasible intervals of size 2 and choosing $\epsilon = 1/2$. For testing membership of other words to L, one might expect that c increases with the

distance from L. How to choose c for given NFA A and error precision ϵ, is less obvious though.

5.4 Blocking and infeasible fragments

We next define the notions of blocking and infeasible fragments formally and show how to decide these properties in polynomial time in the size of the fragment and the NFA, but independently of the size of the word.

Since intervals are particular fragments, we also reintroduce infeasible intervals, which were called "infeasible runs" in [Alon et al., 2000b]. All other notions are original to this thesis.

For all what follows, we fix an NFA $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$, a length $n \in \mathbb{N}_0$ and a word $w \in \Sigma^n$. Furthermore, without lost of generality, we suppose that Σ does not contain symbols which are not used in any transition of A. Indeed any automaton that does not satisfy this requirement can be modified to an equivalent one that does satisfy it in linear time.

From this point it is important to recall that testers will input random objects of some structure isomorphic to S_w, and not an array containing the word w. The main difference between an array and a random object of some σ_{words}-structure Γ isomorphic to S_w is the following. Let $\theta : \Gamma \to S_w$ be the isomorphism between Γ and S_w. The elements of the domain of Γ are not necessarily integers and to know the label at the position $\theta(e)$, for some element $e \in \Gamma$, we may use $|\Sigma|$ queries to the random object of Γ (with calls to $\mathcal{B}_{lab_a}(e)$, where $a \in \Sigma$). A fragment (resp. interval) of Γ is any subset of $F \subseteq \mathit{dom}(\Gamma)$ such that $\theta(F)$ is a fragment (resp. interval) of w. Since the difference stated above does not affect the complexities of algorithms discussed in this section for fragments of w; then for simplicity, in all discussions below, we will restrict ourself only to w and algorithms that inputs: a reference to an array containing w and the length of w. Yet all definitions and results of this section also apply to any representation of w, with algorithms that input a random object of such representation.

Now we explain why complexities of algorithms of this section are unchanged when we consider them to input random object of any representation Γ of w. For any fragment $F \subseteq \mathit{dom}(\Gamma)$, one can compute the positions $\theta(e) \in \mathit{dom}(w)$ corresponding to elements $e \in F$, with the query $\mathcal{R}(<, \mathcal{R}(\mathit{start})) - \mathcal{R}(<, \mathcal{R}(e))$ of rdm_Γ. The label of $\theta(e)$ can also be computed with $|\Sigma|$ ($\leqslant |A|$) queries. Thus with an overall number of $|F||A|$ queries to rdm_Γ, we can compute the part of the array representation of w which corresponds to F. Since all algorithms discussed here with array representation of words have complexities at least $O(|F||A|)$, then, using queries of rdm_Γ, computing the part of the array representation needed in these algorithms can be done with preprocessing step of time $O(|F||A|)$.

However the query complexity of testers presented in this chapter will be

affected with the expensive way of querying the label of elements (positions). Therefore in the rest of this chapter we suppose that random objects of words has in addition the query $lab : dom(\Gamma) \to \Sigma$ which access the label of any input element. With such function non-oblivious random objects corresponds exactly to array representation of words. And results presented in Figure 5.1 are given with such supposition. Without this label function it is important to see that only the query complexities in Figure 5.1 got multiplied with $|\Sigma|$.

Definition 5.2. *A run r of an* NFA *A on a fragment F of w (of size n) is* blocking *if:*

1. *there exist elements $i, j \in dom(F)$ with $i < j$ such that $r(i) \not\to_A^* r(j)$ – in this case we can choose i, j such that $]i, j]$ is disjoint from F –, or*
2. *the maximal element m of $dom(r)$ satisfies $r(m) \not\to_A^* fin$, or $m = n$ and $r(m) \notin fin$.*

We call a fragment F of w blocking *for A if all runs of A on F are blocking.*

Since intervals are fragments without holes, this definition introduces the notion of blocking intervals as a special case. Furthermore, note that no fragment of a word in $L(A)$ is blocking.

Proposition 5.1. *Whether a fragment F of a word w is blocking for an* NFA *A can be decided in time $O(|F||A|)$ by an algorithm that receives as inputs the* NFA *A and a reference to an array containing the word w, and the length of w.*

Note that the whole word w cannot be read in time $O(|F||A|)$. Therefore, a reference to an array is passed as input that contains the word, and in addition, the length of this word.

Proof. We define a non-deterministic evaluator for A on fragments F of the word. It reads the positions of F in increasing order, applies automata transitions between subsequent positions, and whenever meeting a hole of F then it jumps to all states that are reachable from the current state set P. This set is denoted by $\to_A^* (P)$. For all words $w = a_1 \dots a_n$, elements $i \in pos(w)$, non-empty fragments $F \subseteq \{i, \dots, n\}$, and state sets $P \subseteq Q$ we define:

$$eval_A(F) = eval'_A(0, init, F)$$

$$eval'_A(i-1, P, F) = \begin{cases} eval'_A(i, \xrightarrow{a_i}_A (P), F \backslash \{i\}) & \text{if } i = \min(F) \\ eval'_A(j-1, \to_A^* (P), F) & \text{if } j = \min(F) > i \end{cases}$$

$$eval'_A(i, P, \varnothing) = \begin{cases} P & \text{if } i = n \\ \to_A^* (P) & \text{if } i < n \end{cases}$$

Note that $eval_A(F) \cap fin = \varnothing$ if and only if F is blocking for A. Furthermore, $eval_A(F)$ can be computed in time $O(|F||A|)$ by an algorithm that receives as inputs a fragment F, a reference to an array containing w, and the length n of w. This can be done by computing the least fixed point of a ground Datalog program of size $O(|F||A|)$, and thus in this time [see e.g. Gottlob and Koch, 2002]. Note that for all P the set of reachable states $\rightarrow_A^* (P)$ can be computed in time $O(|A|)$. Note also that the computation time is independent of the length of w. □

Definition 5.3. *A run r of an* NFA *A on a fragment F of w (of length n) is called* infeasible *if:*

1. *there exist elements $i, j \in dom(F)$ such that $i < j$ and $r(i) \not\rightarrow_A^{j-i} r(j)$ – in this case, we can choose i, j such that $]i, j]$ is disjoint from F –, or*
2. *the maximal element m of $dom(r)$ satisfies $r(m) \not\rightarrow_A^{n-m} fin$.*

We call a fragment F of w infeasible *for A if all runs of A on F are infeasible.*

Note that blocking runs are infeasible, and thus blocking fragments are infeasible too. Furthermore, no successful run is infeasible, so no word in $L(A)$ may be infeasible nor blocking.

The following proposition for NFAs will help us get rid of the exponential dependency on the automaton size in Alon et. al.'s algorithm.

Proposition 5.2. *Whether a fragment F of a word w is infeasible for an* NFA *A can be decided in time $O(|F||A|^3 + |A|^5)$, when receiving as input a reference to an array containing w and its length (so that the whole word does not need to be read).*

Proof. The decision procedure for infeasibility of fragments is similar to deciding whether a fragment is blocking, except that holes in fragments need to be evaluated more strictly. Therefore, we define a strict evaluator for fragments which behaves like the previous evaluator, except that it respects the number of missing positions in holes of fragments. For any word $w = a_1 \dots a_n$, element $i \in pos(w)$, non-empty fragment $F \subseteq \{i, \dots, n\}$, and state set $P \subseteq Q$ we define:

$$s\_eval_A(F) = s\_eval'_A(0, init, F)$$
$$s\_eval'_A(i-1, P, F) = \begin{cases} s\_eval'_A(i, \xrightarrow{a_i}_A (P), F \setminus \{i\}) & \text{if } i = \min(F) \\ s\_eval'_A(j-1, \rightarrow_A^{j-i} (P), F) & \text{if } j = \min(F) > i \end{cases}$$
$$s\_eval'_A(i, P, \varnothing) = \rightarrow_A^{n-i} (P)$$

Note that $s\_eval_A(F) \cap fin = \varnothing$ if and only if fragment F of w is infeasible for A. We can now test whether fragment F of w is infeasible for A by computing $s\_eval_A(F)$ and checking whether $s\_eval_A(F) \cap fin = \varnothing$. The

following Lemma 5.3 implies that $s\_eval_A(F)$ can be computed recursively along its definition, with $O(|F|)$ recursive calls each of which costs $O(|A|^3)$, after a preprocessing time of $O(|A|^5)$. So the overall computation time is in $O(|F||A|^3 + |A|^5)$ as stated by the proposition. □

Lemma 5.3. *For any* NFA A *we can compute in preprocessing time* $O(|A|^5)$ *an algorithm that receives as inputs a subset* P *of states of* A *and a natural number* $m \in \mathbb{N}_0$ *and computes in time* $O(|A|^3)$ *the set* $\rightarrow_A^m (P)$*, so in time independent of* m*.*

Proof. For any pair of states $(q, q') \in Q$, we compute an NFA $A_q^{q'}$ with a single letter alphabet $(\{0\}, Q, \Delta_A^0, \{q\}, \{q'\})$ where $\Delta_A^0 = \{(q_1, 0, q_2) \mid q_1 \rightarrow_A q_2\}$. We then convert all $A_q^{q'}$ into their Chrobak normal form by using the algorithm in [Gawrychowski, 2011]. This takes time $O(|Q|^3)$ for each pair (p, p') and thus $O(|A|^5)$ all together. Recall that a Chrobak normal form of a single-letter NFA $A_q^{q'}$ is a single-letter NFA $B_q^{q'}$ that recognizes the same language, such that the digraph of $B_q^{q'}$ consists of a single path with at most $|Q|^2$ states, succeeded by a non-deterministic choice of a set of disjoint cycles whose total sizes is at most $|Q|$. One can thus precompute in time $O(|Q|^2)$ the length of the path and an array containing its states, and in time $O(|Q|)$ the length of each cycle and an array containing its states.

We show next – once having precomputed the sizes of the path and the cycles – that given a pair of states $(q, q') \in Q^2$ and $m \in \mathbb{N}_0$, we can check in time $O(|Q|)$ whether $q \rightarrow_A^m q'$. This property is equivalent to that $B_q^{q'}$ accepts some word of length m. If m is smaller than the length of the path of $B_q^{q'}$, this can be done by selecting the m'th state of the path in its array, and testing whether it is final for $B_q^{q'}$. Otherwise, we compute $l' = m - l$ where l is the length of the path, and for all cycles of $B_q^{q'}$ the state that is reached with l' steps. This can be done by computing the remainder of l' by division modulo the length of the cycle, and accessing the state of this remainder in the array of the cycle. It then remains to check whether any of the computed states is final. For each cycle, all these operations can be done in $O(1)$. Since the number of cycles is in $O(|Q|)$ the overall time is in $O(|Q|)$ too.

In order to compute $\rightarrow_A^m (P)$ for some m and P, we check whether $q \rightarrow_A^m q'$ for all pairs $(q, q') \in P \times Q$. This takes $|Q|^2$ time $O(|Q|)$, and thus can be done in time $O(|A|^3)$. □

5.5 Membership for strongly connected NFAs modulo the edit distance

We present an approximate membership tester with respect to the edit distance for regular languages defined by NFAs that are strongly connected.

These are NFAs such that $q \rightarrow_A^* q'$ for any two states q and q'.

Let $\epsilon > 0$, $\delta = 1/\epsilon$ and $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$ a strongly connected NFA containing some initial and some final states. Since A is strongly connected with initial and final states, then $L(A)$ is not empty. Clearly, there exists a run of A on any fragment of any word in $L(A)$, that is, no fragment or interval of any word in the language is blocking. In contrast, words that are ϵ-far from the language with respect to the edit distance must have many short blocking intervals:

Lemma 5.4. *Let $\gamma = 4\delta(|Q| + 1)$, $n \geqslant 8\gamma\lceil\log(\gamma)\rceil$ a natural number, and $w \in \Sigma^*$ a word of size at most n, and d the edit distance from w to $L(A)$. If $d > \epsilon n$ then there exists a power of two $l = 2^i$ in $[2, \gamma]$ such that the number of intervals of length $2l$ that are blocking for A is at least $n\beta_l$ where $\beta_l = l/(2\gamma\lceil\log(\gamma)\rceil)$.*

The assumptions of the lemma imply $|Q| \leqslant \epsilon n \leqslant |w| \leqslant n$, since $|Q| + |w| \geqslant \max(|Q|, |w|) \geqslant d$ holds generally for the edit distance for $L(A) \neq \varnothing$, and $|w| \geqslant d - |Q| \geqslant n\epsilon - |Q| \geqslant |Q|$ by these assumptions, so that $max(|w|, |Q|) = |w|$. In the application in this section, we will choose $|w| = n$, but for the general case in Section 5.6, the lemma will be applied to some large interval of a word of size n. Note also that the number of blocking intervals increases with l. Furthermore, the lower bounds β_l of the linear growth rate increase monotonically with l such that for all $l \leqslant \gamma$:

$$\beta_l \leqslant \beta_\gamma = 1/(2\log(\gamma)) \leqslant 1/4$$

Proof. Let w be a word of size at most $n \geqslant \gamma\lceil\log(\gamma)\rceil$ whose edit distance from $L(A)$ is at least $d > \epsilon n$. We consider the unique decomposition $0 = i_0 < \ldots < i_h = |w|$ such that all $I_j =]i_{j-1}, i_j[$ are maximal non-blocking intervals, where $1 \leqslant j \leqslant h$. Since A is strongly connected, we can repair w such that it becomes a word of $L(A)$ by first deleting the letters of w at positions i_j and then inserting words of length at most $|Q|$ after all i_j where $0 \leqslant j \leqslant h$. Without lost of generality, and to ease up the repair strategy presentation, we show this only in the case where all intervals I_j are not empty. Note that I_j is empty only if the letter at position i_j does not appear in any word of $L(A)$. Then let r_j be some run on interval I_j of w. The word inserted at i_0 is chosen such that it has a total run from *init* to $r_1(i_0)$; this is possible since A is strongly connected and has an initial state. For all $1 \leqslant j < h$, the word inserted after i_j is chosen such that it has a total quasi-run from $r_j(i_j)$ to $r_{j+1}(i_j)$ by A. The word inserted after i_h must have a total quasi-run from $r_h(i_h)$ to *fin* by A'. This is possible since we assume that *fin* is non-empty. Clearly, the repaired word belongs to $L(A)$. The overall correction costs in

terms of letter insertion and deletion operations is $h+1+|Q|(h+1)$, so that $d \leqslant h+1+|Q|(h+1)$. Hence $\epsilon n \leqslant h+1+|Q|(h+1)$ so that:

$$h \geqslant \frac{4n}{\gamma} - 1 \geqslant \frac{3n}{\gamma}$$

The last inequation follows from $n/\gamma \geqslant 2$, which in turn is a consequence of $n \geqslant \gamma \log(\gamma)$ and $\gamma \geqslant 4$ (so that $\log \gamma \geqslant 2$).

Let $S = \{I_j \mid 0 \leqslant j \leqslant h-1\}$. We call an interval of w small if its size is at most γ and big otherwise. We next estimate the numbers of small intervals I in S such that I is blocking for A. For all $2 \leqslant l \leqslant 2\gamma$, let S_l be the subset of S of small intervals whose size belongs to $]l/2, \min(l,\gamma)]$. The number of big intervals of w is at most $|w|/\gamma \leqslant n/\gamma$, so that the number of small intervals in S is at least $h - n/\gamma$. Since $\cup_{i=1}^{\lceil \log(\gamma) \rceil} S_{2^i}$ is a partition of the set of small intervals of S, the above lower bound for h implies:

$$\sum_{i=1}^{\lceil \log(\gamma) \rceil} |S_{2^i}| \geqslant 2n/\gamma$$

Therefore, there exists a number $l = 2^i$ with $2 \leqslant l \leqslant \gamma$ and $|S_l| \geqslant 2n/\gamma\lceil\log(\gamma)\rceil$. We fix such an index l.

We are now interested in the cardinality of the set W_{2l}, which contains all intervals of w of size in $]l, 2l]$ that are blocking for A. We next estimate the cardinality of W_{2l}. Let i_1, i_2, i_3, i_4 be the two smallest and the two greatest indexes in $\{j \mid I_j \in S_l\}$ respectively. Every interval $I_j \in S_l$ where $j \notin \{i_1, i_2, i_3, i_4\}$ is subsumed by $[l, n-l]$ and thus belongs to l intervals of W_{2l}. Conversely, every interval of W_{2l} may contain at most 3 intervals from S_l, since the latter are non-overlapping and of size at least $1 + l/2$. Hence:

$$\begin{array}{rcll} |W_{2l}| & \geqslant & \frac{l\ (|S_l|-4)}{3} & \\ & \geqslant & \frac{l}{3}(\frac{2n}{\gamma\lceil\log(\gamma)\rceil} - 4) & \text{since } |S_l| \geqslant 2n/\gamma\lceil\log(\gamma)\rceil \\ & \geqslant & \frac{l}{3}(\frac{2n}{\gamma\lceil\log(\gamma)\rceil} - \frac{n}{2\gamma\lceil\log(\gamma)\rceil}) & \text{since } n \geqslant 8\gamma\lceil\log(\gamma)\rceil \\ & = & \frac{nl}{2\gamma\lceil\log(\gamma)\rceil} \ = \ n\beta_l & \end{array}$$ □

The above lemma tells us that we can test approximate membership for strongly connected NFAs with respect to the edit distance by selecting sufficiently many small intervals randomly and testing whether they are all non-blocking, and it provides estimations for the sizes and numbers of small intervals that needed to be considered. However, we cannot know the precise size $l = 2^i$ of small intervals, so we have to try out all possible sizes.

Proposition 5.5. *Algorithm membership_edit_connect$_A$ in Figure 5.3 is a one-sided approximate membership tester for words w in regular languages recognized by strongly connected* NFA*s A with respect to the edit distance. The query complexity is in $O(\delta|A|\log^2(\delta|A|))$ and the time complexity in $O(\delta|A|^2\log^2(\delta|A|))$, where $\delta = 1/\epsilon$ is the inverse precision value.*

```
fun membership_edit_connect_A(ε, rdm_w)
  // rdm_w is a random object of some representation of w ∈ Σ*
  // 0 < ε < 1 precision value
  // A = (Σ, Q, Δ, init, fin) strongly connected NFA with init ≠ ∅ and fin ≠ ∅
  let δ = 1/ε          // inverse precision
  let γ = 4δ(|Q| + 1) // size bound for small intervals
  let n = S(<, R(start)) // size of w
  if n < 8γ⌈log(γ)⌉ then
     if w ∈ L(A) then return CLOSE else return NO
     // membership for small words w can be decided by running NFA A on w;
     // using rdm_w to access the word representation
     // this needs time O(γlog(γ) |Q|)
  else // word w is sufficiently long
     for i = 1 to ⌈log(γ)⌉ do
       let l = 2^i
       let β_l = l/2γ⌈log(γ)⌉ // fraction of blocking intervals of size 2l by
           Lemma 5.4
       for j_1 = 1 to 2/β_l
         let e = R(<, R(start))
         let I be the maximal interval of length at most 2l starting from e
         //I is queried using iteratively R(succ, .) at most 2l times
         if interval I is blocking wrt. A
            then return NO and exit else skip
            // this can be tested in time O(2l|Q|) by Proposition 5.1
       end
     end // no small blocking interval of w found
     return CLOSE
```

Figure 5.3: An approximate membership tester for strongly connected NFAs with respect to the edit distance.

Note that the upper bound for the time complexity $O(p^{2,2,1}(\delta, k, |A|))$ where $k = 1$ is slightly better than $O(p^{2,3,1}(\delta, k, |A|))$ as we promised in the introduction for the general case, and similarly for the query complexity.

Proof. Algorithm $membership\_edit\_connect_A(\epsilon, rdm_w)$, where rdm_w is random object of w, first checks whether word w is sufficiently long to apply Lemma 5.4, that is whether $|w| \geqslant 8\gamma\lceil\log(\gamma)\rceil$ with $\gamma = 4\delta(|Q| + 1)$. This can be done in time $O(|A|)$ without traversing the word, since its size $|w|$ can be computed with the query $\mathcal{S}(<, \mathcal{R}(start))$ of the random object. Furthermore, note that we assume that arithmetic operations can be done in size $O(1)$.

The algorithm always returns CLOSE if $w \in L(A)$, since in this case no interval of w is blocking. This shows that the algorithm is one-sided. We next assume that $d(w, L(A)) \geqslant \epsilon|w|$ and want to compute the probability that the algorithm answers NO. Note that to query an interval in the representation of w for which the random object is input, one uses at most $2l$ queries of the form $\mathcal{R}(succ, .)$. By Lemma 5.4 there exists a power of two $l \in [2, \gamma]$ such that the number of intervals of length $2l$ that are blocking for A is at least $|w|\beta_l$ where $\beta_l = l/2\gamma\lceil\log(\gamma)\rceil$. Our algorithm misses all these intervals with probability:

$$(1 - \beta_l)^{2/\beta_l} = (1 - 2/(2/\beta_l))^{2/\beta_l} \leqslant e^{-2} < 1/3.$$

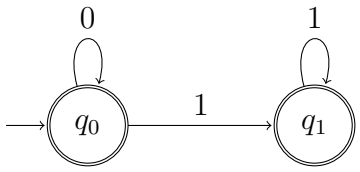


Figure 5.4: An NFA recognizing language 0^*1^*.

The algorithm will thus answer NO correctly with probability of at least 2/3. The query complexity of the algorithm for sufficiently long words is in $O(\sum_{\{l \mid l=2^i,\ 1 \leqslant i \leqslant \lceil \log(\gamma) \rceil\}} \sum_{j=1}^{2/\beta_l} 2l)$ and thus in $O(\delta|A| \log^2(\delta|A|))$. The time complexity is by a factor of $|A|$ higher, since NFA A is to be run on all selected intervals of w in order to check whether they are non-blocking (Proposition 5.1) and thus it is in $O(\delta|A|^2 \log^2(\delta|A|))$. □

5.6 Membership for general NFAs modulo the edit distance

We next treat approximate membership for general NFAs with respect to the edit distance. This problem is more difficult than the case of connected NFAs, since the correction algorithm in the proof of Lemma 5.4 fails. Indeed, this lemma fails for general NFAs, so what we need is a proper generalization.

For illustration, we consider the regular language $L = 0^*1^*$ which can be recognized by the NFA with 2 states in Figure 5.4. This NFA has $k = 2$ strongly connected components. We consider the collection of words $w_m = 1^m 0^m$ with $m \in \mathbb{N}$. A word w_m has length $n = 2m$ and edit distance m from L, so it is 1/2-far from L. An interval I of w_m is blocking for the above NFA if and only if the factor of w at interval I subsumes 10, that is, if $\{m, m+1\} \subseteq I$. Thus, for all $1 \leqslant l \leqslant m$ the number of blocking intervals of w_m of size l is equal to $l - 1$. This number is too small, in that it fails to grow linearly with n for any l in contrast to what Lemma 5.4 would predict for strongly connected NFAs. Similar to the example in Section 5.3.2, this problem can be solved by looking into fragments F with one hole. These are unions of two disjoint subsequent intervals I_1 and I_2 such that $F = I_1 \cup I_2$. Such fragments F are blocking for A if the factor of w_i at I_1 contains the letter 1 while the factor of w_i at I_2 contains the letter 0. Therefore, the number of blocking fragments of size $|F| = 2$ is m^2. This number grows linearly with the total number of fragments which are a union of two intervals ($\leqslant 4m^2$), so we can detect farness from 0^*1^* by inspecting sufficiently many fragments with 2 positions, which need not to be subsequent.

Let $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$ be an NFA. Without loss of generality, we assume that A is productive, i.e. that every state in Q is reachable from *init*

and co-reachable from *fin*. A connected component of A is a maximal subset of states of A that is strongly connected. Note that the strongly connected components of A partition Q. Let k be the number of strongly connected components of A. We also fix a precision value $\epsilon > 0$ and its inverse $\delta = 1/\epsilon$.

Definition 5.4. *A* component path of A *is a sequence* $\Pi = (Q_1, \ldots, Q_j)$ *of pairwise distinct strongly connected components of* A *such that for all* $1 < h \leqslant j$ *some state of* Q_h *is reachable from some state of* Q_{h-1}.

In this case, all states of later components of Π are reachable from all states of earlier components. Furthermore, note that the length of any component path is at most k, that is $0 \leqslant j \leqslant k$.

Let $Q' \subseteq Q$ be a subset of states of A. The restriction of A to Q' is the NFA $A(Q') = (\Sigma, Q', \Delta', \mathit{init}', \mathit{fin}')$ with state set Q', initial states $\mathit{init}' = Q'$, final states $\mathit{fin}' = Q'$, and transition relation $\Delta' = \Delta \cap (Q' \times \Sigma \times Q')$. The restriction of A to a component path Π is the automaton $A(\Pi) = A(Q_1 \cup \ldots \cup Q_j)$.

A decomposition of word w of length n along a component path of some automaton A, $\Pi = (Q_1, \ldots, Q_j)$, is a sequence of integers $J = (i_0, i_1, \ldots, i_j)$ such that $0 = i_0 < \ldots < i_j = n$. Notice that any decomposition depends on the length of the word and the length of the component path (which we sometimes leave implicit in the context).

Definition 5.5. *A word* $w \in \Sigma^n$ *is* ϵ*-far from a path* $(Q_1, \ldots, Q_j)$ *of* A *if for all decompositions* $0 = i_0 < \ldots < i_j = n$ *there exists some integer* h *such that:*

$$d_l(w]i_{h-1}, i_h], L(A(Q_h)) > \epsilon n$$

A word is called ϵ-close from Π if it is not ϵ-far from it. The next lemma shows that sufficiently long words far from $L(A)$ are also far from any component path of A.

Lemma 5.6. *Let* $\alpha = 2(k+1)|Q|\delta$ *where* k *is the number of strongly connected components of* A *and let* $w \in \Sigma^n$ *be a word of length* $n \geqslant \alpha$. *If word* w *is* ϵ*-far from* $L(A)$ *then it is* $\frac{\epsilon}{2k}$*-far from any component path of* A.

Proof. Let a word $w \in \Sigma^n$ with $n \geqslant \alpha$ be $\frac{\epsilon}{2k}$-close to some component path $\Pi = (Q_1, \ldots, Q_j)$ of A. We will then show that w is ϵ-close to $L(A)$. By assumption, there exists some decomposition $0 = i_0 < \ldots < i_j = n$ such that for all natural numbers $1 \leqslant h \leqslant j$:

$$d_l(w]i_{h-1}, i_h], L(A(Q_h))) < \frac{\epsilon n}{2k}$$

Hence, we can correct all factors $w]i_{h-1}, i_h]$ into some word $w_h \in L(A(Q_h))$ at the cost of at most $\frac{\epsilon n}{2k}$ edit operations. Let r_h be a successful run of $A(Q_h)$ on w_h. For $1 \leqslant h < j$, since $A(Q_h)$ is strongly connected and Π a component

path of A, there is a word v_h of length at most $|Q|$ with a total quasi-run from $r_h(|w_h|)$ to $r_{h+1}(0)$. Since A is productive, there also exists a word v_0 of length at most $|Q|$ with a total run from *init* to $r_1(0)$, and another word v_j of length at most $|Q|$ with a total quasi-runs from $r_j(|v_j|)$ to *fin*. Now consider the word $w' = v_0 \cdot w_1 \cdot v_1 \cdots v_{j-1} \cdot w_j \cdot v_j$. Clearly, $w' \in L(A)$. Furthermore:

$$d_l(w, w') \leqslant |Q| + \frac{\epsilon n}{2k} \cdot j + (j-1)|Q| + |Q| = \frac{\epsilon n j}{2k} + (j+1)|Q|$$

This inequality, $j \leqslant k$, and $n \geqslant \frac{2(k+1)|Q|}{\epsilon}$ yield that $d_l(w, L(A)) \leqslant \epsilon n$. □

By combining Lemmas 5.6 and 5.4, we can show for all ϵ-far words, that there is a lower bound on the number of intervals blocking for every component path and decomposition along this path.

Lemma 5.7. *Let $\gamma' = 16k\delta(|Q|+1)$ (that is $\gamma' = 4k\gamma$). Then for any word $w \in \Sigma^n$ of length $n \geqslant 4\gamma'\lceil\log(\gamma')\rceil$ that is ϵ-far from $L(A)$, there exists a power of two l in $[2, \gamma']$ such that for all component paths $\Pi = (Q_1, \ldots, Q_j)$ of A and decompositions $0 = i_0 < \ldots < i_j = n$, there exists some interval $]i_{h-1}, i_h]$ of w which contains at least $n\beta'_l$ subintervals of size $2l$ blocking for $A(Q_h)$, where $\beta'_l = l/(\gamma'\lceil\log(\gamma')\rceil)$.*

Proof. Let $w \in \Sigma^n$ be ϵ-far from $L(A)$ where $n \geqslant 4\gamma'\lceil\log(\gamma')\rceil$ and define $\beta'_l = l/(\gamma'\lceil\log(\gamma')\rceil)$ for all $l \in [2, \gamma']$. Note that $0 < \beta'_l \leqslant 1/4$. We consider an arbitrary component path $\Pi = (Q_1, \ldots, Q_j)$ of A. Since w is ϵ-far from $L(A)$, it follows with $\epsilon' = \epsilon/2k$ that w is also ϵ'-far from Π by Lemma 5.6 (which can be applied since $n \geqslant 4\gamma'\lceil\log(\gamma')\rceil \geqslant \gamma' \geqslant \alpha$). Hence, for any decomposition $0 = i_0 < \ldots < i_j = n$ there exists an index h such that:

$$d_l(w]i_{h-1}, i_h], L(A(Q_h))) > \frac{\epsilon n}{2k}$$

Since $n \geqslant 4\gamma'\lceil\log(\gamma')\rceil$, we can apply Lemma 5.4 to the factor $w' = w]i_{h-1}, i_h]$, NFA $A' = A(Q_h)$, precision value ϵ', and γ' (instead of w, A, ϵ, and γ). Note that the size of w' is at most n as required. The lemma shows that there exists a power of two l' in $[2, \gamma']$, such that at least $n\beta'_{l'}$ intervals of w' are blocking for $A(Q_h)$ of size $2l'$. Let l be the least l' for all component paths Π and decompositions $0 = i_0 < \ldots < i_j = n$. Since $\beta'_{l'}$ grows monotonically with l', the claim follows. □

Let w be a word of length n. For all natural numbers m, we define $\mathcal{S}(w, m)$ to be the set of all sequences $S = (i_1, \cdots, i_m)$ of m positions of w. For a natural number l, a sequence $S = (i_1, \ldots, i_m)$ of $\mathcal{S}(w, m)$ is called l-blocking for A if and only if the fragment $F^l_S = \cup_{1 \leqslant o \leqslant m} I_o$ is blocking for A; where for $1 \leqslant o \leqslant m$, $I_o = [i_o, \min(i_o + l, |w|)]$.

For component paths $\Pi = (Q_1, \ldots, Q_j)$ and decompositions $J = (i_0, \ldots, i_j)$, we say that the sequence S of positions l-matches (Π, J) if and only if there exists a non-blocking run of $A(\Pi)$ on F^l_S such that $r(i) \in Q_h$ for $1 \leqslant h \leqslant j$ and all $i \in]i_{h-1}, i_h] \cap dom(F)$. In this case, we write $(\Pi, J) \models_l S$.

Lemma 5.8. *Let $\gamma' = 16k\delta(|Q|+1)$. Then for any word $w \in \Sigma^n$ of length $n \geqslant 4\gamma'\lceil\log(\gamma')\rceil$ that is ϵ-far from $L(A)$ with respect to the edit distance, there exists a power of two l in $[2,\gamma']$ such that at least $\frac{4}{5}$ of the sequences in $\mathcal{S}(w,\alpha_l)$ are $2l$-blocking for A, where $\alpha_l = 6k\gamma'\lceil\log(\gamma')\rceil^2/l$.*

Proof. Let $w \in \Sigma^n$ be ϵ-far from $L(A)$ where $n \geqslant 4\gamma'\log(\gamma')$. For an interval I of the domain of w, a set $Q' \subseteq Q$, and a natural number l, we denote by $\mathfrak{B}(I,l,Q')$ the set of subintervals of size l of I that are blocking for $A(Q')$.

By Lemma 5.7, we can fix a power of two $l \in [2,\gamma']$, such that for all component paths $\Pi = (Q_1,\ldots,Q_j)$ of A and decompositions $J = (i_0,\ldots,i_j)$ for w and Π, there exists $1 \leqslant h \leqslant j$ such that the interval $I_h =]i_{h-1}, i_h]$ satisfies $|\mathfrak{B}(I_h,2l,Q_h)| \geqslant n\beta'_l$, where $\beta'_l = l/(\gamma'\lceil\log(\gamma')\rceil)$. In particular, note that the size of I_h is strictly greater than $n\beta'_l$ for such indexes h.

In order to obtain the lower bound in the lemma, we prove an upper bound on the number of $2l$-nonblocking sequences in $\mathcal{S}(w,\alpha_l)$. The next claim shows that to obtain the upper bound, we can restrict ourselves only to decompositions whose positions are multiples of $\lambda = n\beta'_l/4$, so that they belong to the set $\Lambda = \{\min(\lceil o\lambda\rceil, n) \mid 0 \leqslant o \leqslant 4/\beta'_l + 1\}$.

Claim 5.9. *For any $2l$-nonblocking sequence $S = (i_1,\cdots,i_{\alpha_l})$ of $\mathcal{S}(w,\alpha_l)$, there exists a strongly connected component Q_h of A and an interval $I =]i,i']$ with $i,i' \in \Lambda$ such that $\mathfrak{B}(I,2l,Q_h)$ contains at least 2λ intervals, but none of the intervals $I_o = [i_o, \min(i_o+2l,|w|)]$, for all $1 \leqslant o \leqslant \alpha_l$.*

Let $S = (i_1,\cdots,i_{\alpha_l})$ be a $2l$-nonblocking sequence in $\mathcal{S}(w,\alpha_l)$. By definition, there exists a component path $\Pi = (Q_1,\ldots,Q_j)$ and a decomposition $J = (i_0,\ldots,i_j)$ such that $(\Pi,J) \models_{2l} S$. That is $F_S^{2l} = \cup_{1\leqslant o\leqslant m} I_o$, the fragment consisting of the union of intervals $I_o = [i_o, \min(i_o+2l,|w|)]$ for all $1 \leqslant o \leqslant \alpha_l$, is none blocking for A. As argued above, there exists $1 \leqslant h \leqslant j$ such that the interval $I_h =]i_{h-1}, i_h]$ satisfies $|\mathfrak{B}(I_h,2l,Q_h)| \geqslant n\beta'_l = 4\lambda$. Since $(\Pi,J) \models_{2l} S$, none of the intervals I_o may belong to $\mathfrak{B}(I_h,2l,Q_h)$. Let I be the largest interval included in I_h with limits in Λ. By inclusion, no interval I_o may occur in $\mathfrak{B}(I,2l,Q_h)$ neither. Furthermore, the number of positions in I_h that are not in I is at most 2λ. Hence:

$$|\mathfrak{B}(I,2l,Q_h)| \geqslant |\mathfrak{B}(I_h,2l,Q_h)| - 2\lambda \geqslant 4\lambda - 2\lambda = 2\lambda$$

This concludes the proof of the claim.

For any interval I with limits in Λ and strongly connected component Q_h with $|\mathfrak{B}(I,2l,Q_h)| \geqslant 2\lambda$, the number of sequences $S = (i_1,\cdots,i_{\alpha_l})$ of $\mathcal{S}(w,\alpha_l)$ for which none of the intervals $I_o = [i_o, \min(i_o+2l,|w|)]$ belongs to $\mathfrak{B}(I,2l,Q_h)$ is bounded by $(n-2\lambda)^{\alpha_l}$; where $1 \leqslant o \leqslant \alpha_l$. To obtain the lower bound on the total number of $2l$-nonblocking sequences, we sum over all possible pairs of intervals and connected components. Their number is bounded by $|\Lambda|(|\Lambda|-1)k \leqslant 20k/\beta'^2_l$ so that the number of $2l$-nonblocking

sequences in $\mathcal{S}(w, \alpha_l)$ is at most $20k/\beta_l'^2 \cdot (n - 2\lambda)^{\alpha_l} \stackrel{def}{=} R$. For $k \geqslant 2$, the lemma now follows from the following estimation:

$$\begin{aligned}
R &\leqslant 20k/\beta_l'^2 \cdot (n - n\tfrac{\beta_l'}{2})^{\alpha_l} && \text{since } \lambda = \tfrac{n\beta_l'}{4} \\
&\leqslant 20k/\beta_l'^2 \cdot n^{\alpha_l} \cdot (1 - \tfrac{\beta_l'}{2})^{\alpha_l} \\
&\leqslant 20k/\beta_l'^2 \cdot |\mathcal{S}(w, \alpha_l)| \cdot (1 - \tfrac{\beta_l'}{2})^{\alpha_l} && \text{since } |\mathcal{S}(w, \alpha_l)| = n^{\alpha_l} \\
&\leqslant 20k/\beta_l'^2 \cdot |\mathcal{S}(w, \alpha_l)| \cdot e^{\frac{-\alpha_l \beta_l'}{2}} \\
&\leqslant |\mathcal{S}(w, \alpha_l)| \cdot 20k/\beta_l'^2 \cdot e^{-3k \log(\gamma')} && \text{definitions of } \alpha_l \text{ and } \beta_l' \\
&\leqslant |\mathcal{S}(w, \alpha_l)| \cdot \tfrac{20k\gamma'^2 \lceil \log^2(\gamma') \rceil}{2} \cdot (\tfrac{1}{\gamma'})^{3k} && \text{since } \beta_l' \geqslant \tfrac{2}{\gamma' \lceil \log(\gamma') \rceil} \\
&\leqslant |\mathcal{S}(w, \alpha_l)| \cdot 10k \cdot (\tfrac{1}{\gamma'})^{3k-4} && \text{since } \log(\gamma') \leqslant \gamma' \\
&\leqslant |\mathcal{S}(w, \alpha_l)| \cdot 10k \cdot (\tfrac{1}{\gamma'})^{2} && \text{since } k > 2 \\
&\leqslant \tfrac{1}{5} |\mathcal{S}(w, \alpha_l)| && \text{since } \gamma' > 10k > 5
\end{aligned}$$

The case $k = 1$ where A is strongly connected follows directly from Lemma 5.4. □

With this lemma and the fact that words from $L(A)$ have only feasible fragments, we obtain an approximated membership tester for NFAs with the expected performance.

Theorem 5.10. *Algorithm membership_edit$_A$ in Figure 5.5 is a one-sided approximate membership tester for words in languages defined by* NFAs *A with respect to the edit distance. The query complexity is in $O(\delta k^2 |A| \log^3(\delta k |A|))$ and time complexity is in $O(\delta k^2 |A|^2 \log^3(\delta k |A|))$; where $\delta = 1/\epsilon$ is the inverse precision value and k is the number of connected components in A.*

Proof. The case $n \leqslant 4\gamma' \lceil \log(\gamma') \rceil$ corresponds to an exact decision procedure so we can consider only the case $n > 4\gamma' \lceil \log(\gamma') \rceil$ in our argument. If $w \in L(A)$ then the path defined by any successful run is matched by all sequences of intervals in w. Therefore the algorithm always answers CLOSE for such words. Now, for each $i \in [1, \lceil \log(\gamma') \rceil]$, the algorithm chooses a sequence of $S \in \mathcal{S}(w, \alpha_l)$ and tests whether S is $2l$-blocking. Thus if w is ϵ-far from $L(A)$, then using Lemma 5.8 one has that with probability at least $\frac{4}{5}$, the algorithm finds a sequence $S \in \mathcal{S}(w, \alpha_l)$ which is $2l$-blocking for A. Therefore the algorithm answers NO with at least the same probability. The query complexity is obtained by counting the accesses I_j, which correspond to some $w]j', j' + 2l]$ in w, for every selected j. Hence it is bounded by:

$$\sum_{i=1}^{\lceil \log(\gamma') \rceil} 2l\alpha_l \leqslant 8k\gamma' \lceil \log(\gamma') \rceil^3 = O(\delta k^2 |A| \log^3(\delta k |A|))$$

For the time complexity the consuming part corresponds to testing whether the sequences S is blocking, using the result of Proposition 5.1 we know that

```
fun membership_edit_A(ε, rdm_w)
  // rdm_w is a random object of some representation of w ∈ Σ*
  // 0 < ε < 1 precision value
  // A = (Σ, Q, Δ, init, fin) NFA with init ≠ ∅ and fin ≠ ∅
  let k be the number of connected components of A
  let δ = 1/ε              // inverse precision
  let n = S(<, R(start)) //size of w
  let γ' = 16kδ(|Q| + 1) // size bound for small intervals
  if n < 4γ'⌈log(γ')⌉ then
     if w ∈ L(A) then return CLOSE else return NO
     // membership for small words w can be decided by running NFA A on w;
     // using rdm_w to access the word representation
     // this needs time  O(4γ'log(γ') |Q|)
  else // word w is sufficiently long
     for i = 1 to ⌈log(γ')⌉ do
       let l = min(2^i, γ')
       let α_l = 6kγ' log^2(γ')/l // number of intervals of size 2l
       for j = 1 to α_l
         let e_j = R(<, R(start))
         let I_j be the maximal interval of length at most 2l starting at e_j.
         //I_j is queried using iteratively R(succ,.) at most 2l times
       end
       if fragment F = I_1 ∪ ... ∪ I_{α_l} is blocking wrt. A
          then return NO and exit else skip
     end
     return CLOSE
```

Figure 5.5: An approximate membership tester for NFAs with respect to the edit distance.

this can be done in time $O(|A|2l\alpha_l)$. Summing up yields a time complexity of

$$O(|A| \sum_{i=1}^{\lceil \log(\gamma') \rceil} 2l\alpha_l) = O(\delta k^2 |A|^2 \log^3(\delta k |A|))$$

□

5.7 Membership for NFAs modulo the Hamming distance

We improve Alon et. al.'s approximate membership tester for DFAs [Alon, Krivelevich, Newman, and Szegedy, 2000b], so that it runs in polynomial time, while being extended to NFAs. The correctness argument follows the same schema as worked out there for the original algorithm. Therefore, we present only a sketch of the proof in which we point out the differences to before.

The main difference of our improved algorithm is that we rely on deciding feasibility of fragments. The original algorithm decided infeasibility for intervals in the case of DFAs with single strongly connected components. In

the general case, it relied on deciding infeasibility of fragments only implicitly, without having extracted that notion. It should also be noticed that we elaborated the same schema again for our new tester for NFAs with respect to the edit distance, with the main difference that infeasible fragments are used there instead of blocking fragments.

5.7.1 Strongly connected NFAs

We start with strongly connected NFAs A. For any word that is far from the language of A with respect to the Hamming distance, we show that it has many small infeasible intervals. This is stated by the following lemma, which is the analogous of our Lemma 5.4 in the case of the Hamming distance, except that it was only stated for DFAs in [Alon et al., 2000b].

Lemma 5.11 (Lemma 2.4 of [Alon et al., 2000b]). *Let A be a strongly connected* NFA *over Σ and $\delta = 1/\epsilon$. Then there exists a natural number $m \leqslant 3|Q|^2$ (called the reachability constant of A) such that for any word $w \in \Sigma^n$ of length $n \geqslant 64\delta m \log(4m\delta)$ such that $d_h(w, L(A)) \geqslant \epsilon n$, there exists a power of two $l \in [2, 4m\delta]$ such that the number of infeasible intervals of w of length $2l$ is at least $2l \cdot n/(\delta m \log(4m\delta))$.*

The proof is analogous to the proof of our Lemma 5.4 for the edit distance. With the Hamming distance, however, one must adapt the repair strategy carefully, such that it produces a word of the exactly the same size n. The fact that we lift this lemma from DFAs to NFAs does not matter, since the original proof did not depend on determinism.

According to Proposition 5.2, we can decide feasibility of intervals in polynomial time in the size of the interval and the NFA, and independently of the size of the word. In combination with Lemma 5.11 this allow us to construct an approximate membership tester with constant query complexity (in the size of the word) that runs in polynomial time for all NFAs that are strongly connected.

5.7.2 General NFAs

In the case of NFAs with many strongly connected components, we consider components path with decomposition of the word, as we did before for the edit distance. Let $A = (\Sigma, Q, \Delta, \mathit{init}, \mathit{fin})$ be an NFA with k strongly connected components. Without loss of generality, we assume that A is productive. Let $0 < \epsilon < 1$ and $\delta = 1/\epsilon$. For any state set Q' we denote by $A(Q', p, q)$ the NFA $(\Sigma, Q', \Delta', \{p\}, \{q\})$ where Δ' is the restriction of Δ to states in Q'. Let $w \in \Sigma^*$. A triplet $(\Pi, J, (p_o, q_o)_{1 \leqslant o \leqslant j})$ consists of a component path $\Pi = (Q_1, \ldots, Q_j)$, a decomposition $J = (i_0, \ldots, i_j)$ of w, where $p_o, q_o \in Q_o$ for all $1 \leqslant o \leqslant j$. A triplet is called admissible if $\mathit{init} \to_A p_0$, $q_j \to_A \mathit{fin}$ and

and for all $1 \leqslant o < j$ it holds that $q_o \rightarrow_A p_{o+1}$ and $p_o \rightarrow_A^{i_{o+1}-i_o-1} q_o$. The next lemma relates the general case to the case with one strongly connected component.

Lemma 5.12 (Lemma 2.7 in [Alon et al., 2000b]). *Let $(\Pi, J, (p_o, q_o)_{1 \leqslant o \leqslant j})$ be an admissible triplet for a component path $\Pi = (Q_1, \ldots, Q_j)$ and a decomposition $J = (i_0, \ldots, i_j)$ of a word w. It w is ϵ-far from $L(A)$ with respect to the Hamming distance, then there exists $1 \leqslant h \leqslant j$ such that:*

$$d_h(w]i_{h-1}, i_h[, L(A(Q_h, p_h, q_h))) \geqslant \frac{\epsilon|w|}{2k}$$

The proof of this lemma is similar to the proof of Lemma 5.6, except that the repair strategy needs to be adapted so that it copes with Hamming distance properly, as done before for Alon et. al.'s algorithm. We omit the details.

Lemma 5.13. *Let M be the maximal reachability constants m for all connected components in A (so $M = 3|A|^2$ in the worst case) and $\eta = 8Mk\delta$. For any word $w \in \Sigma^n$ of length $n \geqslant 16\eta \log(\eta)$ that is ϵ-far from $L(A)$ with respect to the Hamming distance, there exists a power of two l in $[2, \eta]$ such that at least $\frac{4}{5}$ of the elements in $\mathcal{S}(w, \frac{3k\eta \log^2(\eta)}{4l})$ are $2l$-infeasible for A.*

This lemma can be proven in the same way than Lemma 5.8. The only thing that changes is that blocking intervals or fragments are to be exchanged for infeasible intervals or fragments.

Theorem 5.14. *Algorithm membership_hamming$_A$ in Figure 5.6 is a one-sided approximate membership tester for* NFAs *A with respect to the Hamming distance. If k the number of strongly connected components of A and $\delta = 1/\epsilon$ the inverse precision, then its query complexity is in $O(\delta k^2 |A|^2 \log^3(\delta k |A|^2))$ and its time complexity in $O(\delta k^2 |A|^5 \log^3(\delta k |A|^2))$.*

Proof. Based on Lemma 5.13, we can argue as we did for the edit distance, that algorithm *membership_hamming$_A$* is a membership tester with respect to the Hamming distance. Its query complexity and running time are mainly due to deciding the feasibility of the randomly selected sequence of intervals. By Proposition 5.2, we can compute the feasibility of a fragment F in time $O(|F||A|^3)$ after a global precomputation (once for all fragments) in $O(|A|^5)$. □

5.8 Conclusion

We have shown that approximate membership testing for NFAs with constant query complexity can be done in polynomial time. It turns out that

```
fun membership_hamming_A(ϵ, rdm_w)
  // rdm_w is the random object of some representation of w ∈ Σ*
  // 0 < ϵ < 1 precision value
  // A = (Σ, Q, Δ, init, fin) NFA with init ≠ ∅ and fin ≠ ∅.
  let k be the number of connected components of A
  let M be the maximal reachability constant m of all strongly connected
      components of A
      // M can be computed in time O(|A|^2) as shown in the original
           algorithm
      // or else we can choose the worst case M = 3|Q|^2
  let δ = 1/ϵ // inverse error precision
  let η = 8Mkδ
  if n < 16η log(η) then
     if w ∈ L(A) then return CLOSE else return NO
     // membership for small words w can be decided by running NFA A on w;
     // this needs time O(ηlog(η) |Q|)
  else // word w is sufficiently long
     for i = 1 to ⌈log(η)⌉ do
       let l = min(2^i, η)
       for j = 1 to 3kη log^2(4η) / l
         let e_j = R(<, R(start))
         let I_j be the maximal interval of length at most 2l starting at e_j.
         //I_j is queried using iteratively R(succ,.) at most 2l times
       end
       if fragment F_S = I_1 ∪ ... ∪ I_{α_l} is infeasible wrt. A
          then return NO and exit else skip
     end
     return CLOSE
```

Figure 5.6: An approximate membership tester for NFAs with respect to the Hamming distance.

approximation modulo the edit distance leads to more natural algorithms with lower query and time complexity.

We recall that to test approximate membership for an NFA, the idea we exploit above is to randomly inspect the word which representation is provided to our algorithms. Thus we have showed that small witnesses of farness (blocking or infeasible fragments) are important; and then, by breaking down the approximate membership for an NFA into running the NFA on those small witnesses one provided efficient testers. This is possible mainly because of locality reasons. Indeed inspecting locally some parts of the word witnesses whether the word is far from an NFA.

One would like to use the same idea for regular tree languages. However, for general regular tree languages, we conjecture that this will not be possible without imposing serious locality restrictions, but this needs to be elaborated. Since document type descriptors (DTDs) for XML documents satisfy strong locality restrictions, one could hope for approximate membership testers for (DTDs) and thus for efficient approximate XML schema validation. First results in this direction exist already for the edit distance with moves [Magniez and de Rougemont, 2007; de Rougemont and Vieilleribière, 2007]. And we also study testability of DTDs in the next chapter for the strong edit distance between trees.

Another interesting problem would be to study approximate inclusion or equivalence checking with respect to the edit distance [Benedikt, Puppis, and Riveros, 2011]. So far, approximate inclusion checking has only been considered for the edit distance with moves [Fischer et al., 2006]. So the question is whether approximation can lead to realistic algorithms in this perspective.

6 Approximate DTD validity

Contents

6.1 Introduction

Validity checking for collections of large XML documents may quickly become time consuming. With today's technology, more than 10 minutes are needed to validate a single document of more than 20 giga bytes, so that the treatment of hundreds such documents may take days or weeks. This difficulty could be overcome by sublinear algorithms that can quickly detect invalid documents without reading them entirely.

Whether sublinear algorithms for schema validation exist is a principle question, which could become relevant to various applications. First, they could be used to speed up query answering algorithms for aggregate queries with schemas as filters. Second, they are relevant for Web servers that must check the validity of XML documents hosted by various clients. Since servers are subject to bandwidth limitations they may be forced to receive input documents one by one, so that they cannot be processed in parallel. Sublinear validity testers, in contrast, often allow to detect invalidity of documents stored at the client's site, while requiring to communicate only a small fragment to the server. A third application is the validation of views on XML databases produced by simple tree transducers without having to materialize them [de Rougemont and Vieilleribière, 2007].

Sublinear algorithms for DTD validation can also be obtained by streaming [Green, Gupta, Miklau, Onizuka, and Suciu, 2004; Martens et al., 2006b; Alur and Madhusudan, 2009; Konrad and Magniez, 2012]. This, however, works only for such XML streams that contain an error close to the beginning; otherwise a large prefix of the stream needs to be inspected yielding at least linear time algorithms. In contrast, our objective is to develop probabilistic approximation algorithms inspired by property testing [Rubinfeld and Sudan, 1996; Fischer, 2001; Goldreich, 1998; Fischer et al., 2006; Alon et al., 2000b; Czumaj and Sohler, 2010] which access a random fragment of constant size only, in order to detect invalidity with high probability, if the input structure contains many errors where so ever.

In the previous sections, we have modelled XML documents as trees which are represented as relational structures. We also provided how one could access these tree structures through their random objects without having to read them entirely. These random objects mainly permits to randomly generate any node, to access to parent, first-child and next-sibling of any node. Random objects also permit to generate some ancestor and some descendant of any node. All these accesses are realistic and can be implemented in Databases as discussed in Section 4.3. This is mainly done by storing for each node: the number of its opening and its closing event in the XML stream. So these accesses can also be implemented by DOM after SAX parsing, and which is supported by existing XML databases [see e.g. Arion, Benzaken, Manolescu, and Papakonstantinou, 2007]. Thus one can study approximate DTD validation in our property testing framework with a distance notion over

A DTD for **c**ollections of **b**ooks with **a**uthors:

```
c -> b*, b -> a+, a -> #PCDATA
```

Data model of an invalid XML document:

$$t_i = \begin{array}{ccccc} & & c & & \\ & \swarrow & & \searrow & \\ a & \underbrace{\cdots}_{i} & & & a \end{array}$$

The XML linearization:

$$w_i = \texttt{<c>}\underbrace{\texttt{<a/>}\cdots\texttt{<a/>}}_{i}\texttt{</c>}$$

Schema of XML linearizations of valid trees:

$$\texttt{<c>(<a/>}^{+}\texttt{)}^{*}\texttt{</c>}$$

Figure 6.1: The strong edit distance of tree t_i to the DTD is equal to i, while the edit distance of its linearization w_i to valid linearizations is only 2.

trees.

Now the question is which distance measures most the number of errors in XML document with respect to a given DTD. In Section 2.2, we have discussed the general idea of using edit operations to design edit distances between trees (XML documents). The usual tree edit distance [Pawlik and Augsten, 2011; Bille, 2005; Zhang and Shasha, 1989], for instance, supports node insertion, node deletion, and node relabeling. As already discussed in [Bernard, Boyer, Habrard, and Sebban, 2008; Polyzotis, Garofalakis, and Ioannidis, 2004], however, node insertion and deletion may change the structure of trees heavily, so that it leads to a large approximation to assign the low cost of 1 to them. An example is given in Figure 6.1. The trees $t_i = c(a^i)$ there were obtained from a library collection (c) by deleting the unique book node (b) with i author children (a). The usual tree edit distance of t_i to a valid library is thus equal to 1, even though i different edges are affected by this deletion. We propose to solve this problem by using the strong tree edit distance from Selkow [Selkow, 1977], which restricts the usual tree edit operations to leaf insertion, leaf deletion, and node relabeling. In our example, the strong edit distance of t_i to some valid library is indeed equal to i. Generally, it should be noticed that stronger distances with fewer edit operations are advantageous for property testing, in that any tester for a stronger distance is also correct for any weaker distance (although they might exist faster tester for the weaker distance).

To the best of our knowledge, no approximate membership tester for DTDs

modulo the usual tree edit distance exists so far. The question is also open for more general schema languages such XML SCHEMA or RELAXNG, and also for tree automata for unranked trees as noticed already in [Chockler and Kupferman, 2004]. The best result existing so far is the approximate membership tester for trees, presented by Fischer, Magniez and Rougemont [Fischer et al., 2006; Magniez and de Rougemont, 2007], which however applies to an even weaker tree edit distance, allowing for subtree moves in addition to node insertion, deletion, and relabeling. It should also be noticed that property testers for graphs are usually limited to local properties [see e.g. Ron, 2008; Newman and Sohler, 2011], which is insufficient to deal with regular expressions as in DTDs.

The main contribution in this chapter is an efficient probabilistic algorithm testing approximate DTD validity modulo the strong tree edit distance, whose run time depends only on the depth of the XML document but not of its size. Given as inputs an error precision $\epsilon > 0$, a DTD D, and an XML documents t that is ϵ-far (normalized by the size of t) from satisfying the DTD D modulo the strong edit distance, the algorithm returns NO with high probability. For valid trees it answers CLOSE and for all others either CLOSE or NO. The running time is polynomially bounded in the depth of t, $1/\epsilon$, and the mintree size m_D of the DTD, which is the maximum over element names $a \in \Sigma$ of the minimal sizes of a-labeled subtrees of D-valid trees. Even though m_D may grow exponentially with D, it seems to be close to the size of D for all practically relevant DTDs. Furthermore, m_D can be computed in quadratic time in the size of D, so unusual cases can be recognized efficiently and passed directly to exact DTD validity checking.

XML documents of depth 1 are essentially words, so that the recent approximate membership tester for non-deterministic finite automata (NFAs) modulo the edit distance on words from [Ndione, Niehren, and Lemay, 2012] can be applied. This tester improves on a previous tester by Alon, Krivelevich, and Newman [Alon et al., 2000b] for the Hamming distance, so that it runs in polynomial time in the size of the automaton and the inverse error precision, and still independently of the size of the input word. For XML documents of greater depths, the situation is more difficult. We will show that a membership tester for tree automata modulo the usual tree edit distance can be obtained by linearization of unranked trees into words (and thereby provide a partial answer to an open question from [Chockler and Kupferman, 2004]). This works since the usual edit distance between trees can be bounded in function of the edit distance on their linearizations [Akutsu, 2006]. The time complexity of this tester, however, will depend exponentially on depth of the input.

The next difficulty is that one cannot use the linearization approach for approximate membership testing modulo the *strong* tree edit distance. This is also illustrated in Figure 6.1, where w_i is the linearization of t_i. The edit distance of w_i to the language of linearizations of valid tree is only 2, since

the insertions of one opening tag and one closing tag around the factor <a/>^{i} of w_i are sufficient for correction, while the strong edit distance of t_i to the DTD is equal to i. Intuitively, the insertion of the tags and around the factor <a/>^{i} should be assigned a much higher cost, since the b-node inserted thereby is supposed to capture i a-nodes as its children.

To remedy the situation we study weighted words, i.e., words in which all positions are assigned to a weight. The edit distance on words is then lifted to weighted words, such that the costs of edit operations are given by these weights. We then present a polynomial time NFA membership tester for weighted words modulo the stong edit distance. It naturally induces an approximate DTD-membership tester for trees of depth 2 modulo the strong tree edit distance. A tester for trees of bounded depth could be obtained by recursively testing trees of depth 2. The running time, however, would then become exponential in the depth of the tree, since the error precision would be divided by a constant factor in all recursion steps. Furthermore, such an approach could only be applied to trees of bounded depth. So the remaining challenge is how to test DTD membership for trees without any depth restriction, while depending polynomially on the depth of the tree. To this end, we contribute a direct reduction from approximate DTD validity to approximate NFA membership of weighted words.

We complement our positive result by a negative result that illustrate the limitation of property testing modulo the strong tree edit distance. By a Yao-style argument [Yao, 1977] we will show that approximate DTD validity cannot be tested by any probabilistic algorithm whose running time does not depend at least linearly on the input tree depth.

6.1.1 Outline

In Section 6.2 we recall preliminaries on XML data models and schemas. In Section 6.3, we recall edit distances for trees and words. In Section 6.4, we present our main result. In Section 6.5, we introduce weighted words, lift the edit distance, and present our tester for membership of weighted words to regular languages modulo the edit distance. In Section 6.6, we prove the main result, and in Section 6.7 we present our negative result.

6.2 Data models and schemas

We recall preliminaries on the XML data model and on XML schemas. We start the discussion with word automata that were introduced in Section 3.4.1 which will be used for DTDs instead of regular expressions. We recall that we denote the root of any tree by ε. This sometimes overlap with the notation used in DTDs for the regular expression, which denotes the language containing only the empty word. However what is denoted by ε is always

clear depending on the context.

6.2.1 Words

In the preliminaries chapter (Chapter 2) we have introduced words, the notion of intervals and fragments. We represented words as σ_{words}-structures. Thus the notion of fragments and intervals have been extended to any structure representing a word. We also defined membership testing as approximate model checking of relational structures using distance measures and random access to relation structures

In this context, we studied approximate membership of words for NFA which designate regular languages (see Chapter 5). We have shown that, under the edit distance d_l, approximate membership for NFAs can be tested with constant time complexity. The tester essentially runs the NFA on fragments of the input word representation using its random-object. It then decides whether the fragment is blocking or not to answer the approximate membership problem. In Section 5.4, It has also been proven that: running an NFA A on a fragment F is possible in $O(|F||A|)$ time complexity. We have discussed in the same section that the query complexities of the testers of the previous chapter must be multiplied by the size of the alphabet if we do not use array representations (or equivalent random objects), but use instead the non-oblivious word random object of Section 4.2.2 (the time complexities remain unchanged). Thus in this chapter we will only consider non-oblivious random objects of words introduced in Section 4.2.2, but we will adapt the complexities of the previous chapter. We remind that those random objects allow to generate uniformly an element of the domain of their structure, they also generate the start element of the domain and the next-sibling of any element. Furthermore all satisfied relations can be known on tuples of elements.

The ideas we use for approximate DTD validity are very similar to the ones for regular word languages. However, as discussed in the introduction, we will need to generalize them for weighted words introduced below. Before going further we discuss our XML data model and define DTDs using automata.

6.2.2 XML data model

The XML data model essentially boils down to finite unranked data trees when ignoring details of attributes, processing instructions and comments. Since we only consider structural aspects of XML documents described by DTDs, we can safely ignore data values and thus simplify the XML data model further to finite unranked trees over a finite alphabet (fixed by the DTD).

In Section 2.1.3, formal definition of trees are provided and their encodings to unranked trees have also been discussed. Tree automata are introduced in Section 3.4.2. Trees are represented by relational σ_{tree}-structures over the

following vocabulary.

$$\sigma_{trees} = \{lab_a^1 \mid a \in \Sigma\} \uplus \{fc^2, ns^2, parent^2, desc^2, root^1, anc^2\}$$

In this vocabulary, one have relations for the label of nodes, their fist-child, next-sibling, parent, descendant, ancestor and the root relation is only satisfied by the element of the structure domain corresponding to the root. XML schemas, in particular DTDs, are designed to denote XML document languages i.e. XML document properties (see Section 3.5.1). As we model XML documents with unranked trees, in the next section we give equivalent definition of DTDs using word automata instead of regular expressions.

DTDs specify constraints over the word of any node in valid trees. Now we define the word associated with any tree as it follows. Let Σ be an alphabet. For any tree $\mathrm{t} \in \mathcal{T}_\Sigma$, we define the word $word(\mathrm{t}) \in \Sigma^*$ of t to be the sequence of labels of the children of the root of t. And the node of any node $v \in nod_\mathrm{t}$ is defined as $word(v) = word(\mathrm{t}_{|v})$. For instance, if $\mathrm{t} = c(b(a,a), b(a,a,a))$ then $word(\mathrm{t}) = bb$ and $word(\mathrm{t}_{|\varepsilon \cdot 2}) = aaa$.

6.2.3 Schemas

Various languages for defining schemas of XML documents were proposed in the literature. Document type descriptors (DTDs) are most basic, while XML SCHEMAs are more expressive. Both can be compiled into top-down deterministic tree automata on top-down binary encodings of unranked trees. RELAXNG captures even more expressive regular context-free grammars, i.e., tree automata enhanced by compression via named pattern. Our choice of DTDs is motivated by the fact that equally efficient membership testers for more expressive formalisms such as tree automata are difficult to find or may even not exist.

Standard DTDs define regular languages of unranked trees by using regular expressions. These can be compiled into NFAs in linear time, but only when permitting ε-transitions [Schnitger, 2006; Hagenah and Muscholl, 2000]. It should also be noticed that all regular expressions in DTDs are deterministic (see the W3C recommendation). Therefore they can be converted into deterministic finite automata in polynomial time. However, this conversion might require quadratic time if not fixing the alphabet [Brüggemann-Klein, 1993]. Therefore it might be advantageous using DFAs with ε-transitions. However as all algorithms discussed here are at least in quadratic time in the size of the DTD, then converting DTDs regular expressions into NFAs without ϵ-transitions will not affect our complexities. Thus, to comply with our previous definition of NFAs, which was without ϵ-transitions, we do not consider such transitions in the following. Yet one should notice that all complexity results obtain in the previous chapter for detecting whether fragments are blocking for some NFAs straightforwardly generalize when one allows

ϵ-transitions. Thus our choice is rather for simplicity of the presentation rather than a limiting issue. In some examples of DTDs we will use regular expressions for illustration nevertheless.

Definition 6.1. *A* DTD D *over an alphabet* Σ *is a tuple* $(\Sigma, init, (A_a)_{a\in\Sigma})$ *where* $init$ *is an element of* Σ*, and all* A_a *are* NFAs *with alphabet* Σ*.*

An unranked tree t over Σ is valid for a DTD D iff $lab_t(\varepsilon) = init$ and $word(t_{|v}) \in L(A_a)$ for all $v \in nod_t$ and $lab_t(v) = a$; where $a \in \Sigma$. We denote the set of all D-valid trees by $L(D)$.

For all labels $a \in \Sigma$ and DTD $D = (\Sigma, init, (A_a)_{a\in\Sigma})$, we denote by D_a the DTD $(\Sigma, a, (A_a)_{a\in\Sigma})$. The mintree size m_D is the maximum for all $a \in \Sigma$ of all minimal sizes of trees belonging to $L(D_a)$:

$$m_D = \max_{a\in\Sigma, L(D_a)\neq\varnothing} \min\{|t| \mid t \in L(D_a)\}$$

Note that one can compute m_D in quadratic time from D even though this number might be exponentially bigger than the size of D.

6.3 Edit distances

We recall the usual edit distances for words and trees, as well as the strong tree edit distance.

6.3.1 Edit operations

We remind that the (usual) edit operations on words permit to relabel, insert, and delete a letter at a given position. The edit distance between two words w and w' is the least number of usual edit operations needed to transform w into w'. It is denoted by $d_l(w, w')$. The standard edit operations on trees allow for node relabelling, node inserting, and node deletion [Pawlik and Augsten, 2011; Bille, 2005; Zhang and Shasha, 1989]. The standard edit distance on unranked trees t and t', denote by $d_{stand}(t, t')$, is the least number of usual edit operations required to transform t into t'. The strong edit operations [Selkow, 1977] restricts the usual edit operations to node relabelling, leaf insertion and leaf deletion. The strong edit distance between two trees t and t' is the least number of strong edit operation to transform t into t'. It is denoted by $d_{strong}(t, t')$. All these distances are formally define in the preliminary chapter (Chapter 2).

Since the strong edit distance between tree permits less edit operations than the standard edit distance, the following inequality always holds:

$$d_{stand}(t, t') \leqslant d_{strong}(t, t')$$

Furthermore, $d_{strong}(t, t') \leqslant |t| + |t'| - 1$ since one can first delete all nodes of t except the root, then relabel the root, and finally add all non-root nodes of t' one by one.

6.3.2 Farness and approximate membership

Any distance measure over a set S was extended to a distance from elements of S to subsets of S in Section 2.2. And in Section 3.3.2, Definition 3.1, we have generally defined the notion of ϵ-farness from any property of relational structures, with respect to a distance measure. In Definition 3.1, we used logical formulas to denote properties, which are understood as a class (or set) of structures. Since the property denoted by a DTD D (or a tree automata A), is assimilated with the set of trees $L(D)$ ($L(A)$ respectively), one obtain the notion of ϵ-farness from a DTD D, for tree structures.

Note that ϵ-farness from a DTD D with respect to the standard edit distance implies ϵ-farness from D with respect to the strong edit distance. Furthermore, since $d_{strong}(t, t') \leqslant |t| + |t'| - 1$ for any two unranked trees, it follows that $d_{strong}(t, D_a) \leqslant |t| + m_D - 1 \leqslant m_D\ |t|$ for all labels a in the alphabet of a non-empty DTD D (a tree always has a root, so $|t|, m_D \geqslant 1$). Since emptiness of DTDs is linearly decidable we only consider non empty DTDs in the rest of the chapter.

Then the approximate DTD validity in this context corresponds to property testing of tree structure modulo some distance ; see definitions in Sections 3.3.2 and 3.4.4. Approximate membership can also be applied for properties denoted by automata using random accesses to tree structures; see Section 3.4.4. The random object of tree structures have also been defined in Section 4.3 with the following vocabularies: $\sigma_{det} = \sigma_{trees}$, $\sigma_{size} = \{anc^2\}$ and the random vocabulary is $\sigma_{rand} = \{fc^2, ns^2, parent^2, desc^2, root^1\}$.

A $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random object of some representation of a tree provide queries for accessing the first-child, next-sibling, parent of any node. It also gives access to the root of the tree and can uniformly generate a descendent of any node. Therefore using such random objects one can navigate trough the tree domain. Furthermore the depth of any node can be accessed using a size query. See Section 4.3 for example of queries to random objects of trees. The intuition in using such random access for approximate DTD validity is the following: a DTD specifies constraints on the label of any node children using regular expressions. While these constraints are horizontal, any node label is also related to the label of all its ancestors, yielding a vertical constraints. Thus in order to efficiently detect DTD invalidity efficiently one needs both horizontal and vertical information. The vertical information is just here provided by the depth of nodes and the horizontal information is obtained by accessing the siblings and children. Hereinafter one could see how to exploit these informations to detect invalidity, once we expose our tester for DTDs.

We recall the definition of testers for DTD validity or more generally properties denoted by tree automata. We use the terminology property definition (or equivalently language definition) for DTDs and tree automata in the following definition. In this definition, we denote by $\mathcal{A}$ the class of property

definitions (class of DTDs or class of tree automata).

Definition 6.2. *An approximate membership tester, for trees and properties denoted by definitions in $\mathcal{A}$, is an algorithm (possibly randomized) that receives as inputs a random object rdm_t of some structure isomorphic to a tree t, an error precision $\epsilon > 0$ and a property definition $A \in \mathcal{A}$, and answers with probability $\frac{2}{3}$: CLOSE if $s \in L(A)$ and NO if s is ϵ-far from A.*

The *query complexity* of a tester is the number of times it uses rdm_t during the computation in dependence of the input (size). Its *time complexity* accounts for all other operation performed by the algorithm in addition to the query complexity. For more details one is referred to Chapters 3 and 4, where approximate membership is discussed (see Sections 3.3.2 and 3.4.4)

The first attempt to obtain a tester for tree is to consider their linearizations as done with the edit distance with moves [Magniez and de Rougemont, 2007]. In the next section, we exhibit relations that may allow such approach. Note that such relations do not exist for the strong edit distance as already discussed in the introduction.

6.3.3 Linearizations

The relationship from [Akutsu, 2006] between the standard edit distance on trees t and t′ and the edit distance of their respective XML linearizations w and w' depends on the minimal depth of the two trees $d = \min(d(\mathrm{t}), d(\mathrm{t}'))$:

$$\frac{d_l(w,w')}{2} \leqslant d_{stand}(t,t') \leqslant (2d+1)\ d_l(w,w')$$

These estimations are thight up to a constant factor. The upper bound shows for any tree t of depth d that, if t is ϵ-far from a DTD D modulo the standard tree edit distance, then its XML linearization w is $\frac{\epsilon}{2d+1}$-far from the linerizations of any D-valid tree. See Figure 6.2 for an example, where the difference grows linearly with the depth.

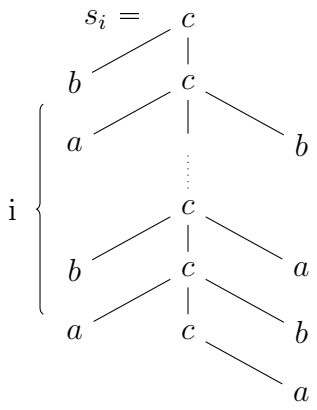
Note however, that the same upper bound does not hold for the strong tree edit distance $d_{strong}(t,t')$, as shown by counter example in the introduction, since only trees of depth 1 were used there. This indicates already, that we will need a more general method for testing DTD membership modulo the strong tree edit distance, than for testing membership for finite word automata modulo the edit distance.

Accessing linearizations through trees random objects

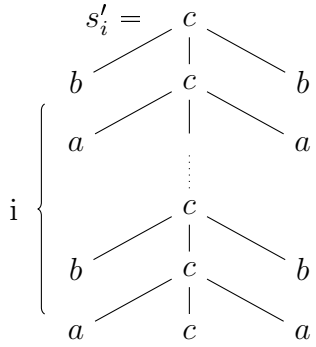
As we will explain in the following section, the first attempt to obtain testers for trees modulo the standard edit distance is to use the membership tester of Figure 5.5. For this to be possible, one must simulate all queries to

A DTD: $c \to aca \mid bcb \mid \varepsilon$

An invalid tree:



A valid tree:



The linearization of s_i:

v_i = (<c><b/><c><a/>)^i<c>
(<a/></c><b/></c>)^i</c>

The linearization of s'_i:

v'_i = (<c><b/><c><a/>)^i<c></c>
(<a/></c><b/></c>)^i

Figure 6.2: The tree s_i is at least 1/3-far from being valid even for the usual tree edit distance, since all third children of all its c-nodes must be relabelled to become valid. The linearization v_i of s_i however can be corrected to the linearization v'_i of the valid tree s'_i with only 2 edit operations (moving the closing tag of the lowest c-node to the end), so that v_i is $2/3i + 1$-close to a linearization of a valid tree.

word structures used in the tester of Figure 5.5 by the random object of the tree which linearization is considered. This way one can mimic the approximate membership tester of words on the linearization of trees to obtain an approximate membership of trees modulo the standard edit distance.

Let Σ be an alphabet and Γ be a tree σ_{trees}-structure isomorphic to some tree t $\in \mathcal{T}_\Sigma$. One considers the σ_{words}-structure Γ'_t such that:

$$
\begin{aligned}
dom(\Gamma'_t) &= \{e_i \mid e \in dom(\Gamma), i \in \{0,1\}\} \\
start^{\Gamma'_t} &= \{e_0 \mid root^\Gamma(e)\} \\
succ^{\Gamma'_t} &= \{(e_1, e'_0) \mid ns(e,e')\} \cup \{(e_0, e'_0) \mid fc(e,e')\} \cup \{(e_0, e_1) \mid \nexists e'.fc(e,e')\} \\
<^{\Gamma'_t} &= \{(e_i, e'_j) \mid < (e,e'), i,j \in \{0,1\}\} \cup \{(e_0, e_1) \mid e \in dom(\Gamma)\}
\end{aligned}
$$

Note that $<^{\Gamma'}$ is the transitive closure of $succ^{\Gamma'}$ and that Γ' is isomorphic to the σ_{words}-structure S_w, where w is the linearization of t. Now the tester of Figure 5.5 simply test if the size of the word which randomization is input is large, and when this is the case it simply uniformly generates elements

of the domain of the structure which random object is input and it runs an automaton on the selected fragment. It is straightforward that all these accesses can be simulated by the $(\sigma_{det}, \sigma_{size}, \sigma_{rand})$-random object of Γ. To generate and element of Γ' we only generate an element $e \in dom(\Gamma)$ and with probability $\frac{1}{2}$ we return e_0 or e_1. And to check whether the linearization of t is large enough we use the fc and ns relations to test whether the size of t is not small. Not however that the size access of the non-oblivious random object of w can not be simulated but this is not important to mimic the behaviour of the word membership tester of Figure 5.5; modulo the edit distance.

6.4 Main results

In this section we formulate results about our testers for approximate membership of tree structures modulo the standard edit distance and the strong edit distance.

6.4.1 Standard tree edit distance

We sketch how to test approximate membership for tree automata on unranked trees modulo the usual tree edit distance, based on tree linearization. Note that such tree automata subsume our DTDs.

The idea is to use the upper bound $d_{stand}(t, t') \leqslant (2d + 1)d_l(w, w')$ from [Akutsu, 2006], where w is the XML linearization of t and w' the XML linearization of t'. We want to test approximately whether an unranked tree t of depth d is recognized by a tree automaton B. If t is ϵ-far from B modulo the usual tree edit distance, then its linearization is $\frac{\epsilon}{2d+1}$-far from the language of linearizations of trees recognized by B of depth at most d. The tree automaton B can then be compiled into finite automata A of exponential size $|B|^d$ that accepts all these linearizations. This can be done by first compiling B into a nested word automaton [Alur and Madhusudan, 2009] in linear time, which in turn is compiled to a finite automaton by moving stacks up to depth d into states. One can then apply the polynomial time membership tester for finite automata modulo the edit distance on words from the previous chapter [see also Ndione et al., 2013]. In order to do so, one has to verify that the randomized data model of words can be simulated by a randomized data model of the corresponding tree, as we explain in the previous section. Since the tester of the previous chapter (Figure 5.5) never errs for correct words, and there is no requirement on close trees, this method gives indeed a valid tester. The query complexity of this test is in $O(|\Sigma|p(|A|, 1/\epsilon, d))$ where Σ is the alphabet of A and p is the polynomially bounded function that satisfies for all positive real numbers $\mathbf{a}, \mathbf{e}, \mathbf{d}$:

$$p(\mathbf{a}, \mathbf{e}, \mathbf{d}) = \mathbf{a}^3\ \mathbf{e}\ \mathbf{d}\ \log^3(\mathbf{a}^2\ \mathbf{e}\ \mathbf{d})$$

The time complexity is in $O(|A|\ |\Sigma|\ p(|A|, 1/\epsilon, d))$. Note that the alphabet size factor is only due to the way one access labels and when there is in addition a label function for random objects this factor disappear (see previous chapter). In combination we obtain the following result:

Theorem 6.1. *Whether an unranked tree t is approximatively recognized by a tree automaton B modulo the tree edit distance with error precision ϵ can be tested with query complexity $O\left(|\Sigma|\ p\left(|B|^{d(t)}, 1/\epsilon, d(t)\right)\right)$ and time complexity $O\left(|B|^{d(t)}\ |\Sigma|\ p\left(|B|^{d(t)}, 1/\epsilon, d(t)\right)\right)$. Where Σ is the alphabet of B.*

This theorem has three weaknesses. First of all, the finite automaton A constructed from the tree automaton B and the depth d may be of exponential size $O(|B|^d)$. Second, the test applies to the usual tree edit distance only; but not to the strong tree edit distance. And third, the query complexity of the tester depends exponentially on the depth of the tree.

6.4.2 Strong tree edit distance

Our main result is that all three problems can be solved for DTD membership modulo the strong tree edit distance, as stated in the following theorem.

Theorem 6.2. *Whether an unranked tree t is valid for a* DTD $D = (\Sigma, init, (A_a)_{a\in\Sigma})$ *modulo the strong tree edit distance with error precision ϵ can be tested with query complexity in $O(d^3\ |\Sigma|\ p(\boldsymbol{a}, d^2/\epsilon, m_D))$ and time complexity in $O(\boldsymbol{a}\ d^3\ |\Sigma|\ p(\boldsymbol{a}, d^2/\epsilon, m_D) + |D|^2)$, where $d = d(t)$ and $\boldsymbol{a} = \max_{a\in\Sigma} |A_a|$ is smaller than $|D|$.*

The dependency on the depth is reduced from exponential to polynomial. We will also show a lower bound, proving that the query complexity must depend at least linearly on the depth (Theorem 6.10). In contrast, approximate membership of NFAs can be done with constant query complexity [Ndione et al., 2013; Alon et al., 2000b; Fischer et al., 2006]. Nevertheless, as DTDs are naturally connected to NFAs, one might want to reduce approximate membership of the former to the one of the latter. We believe that this cannot be archieved. Instead, we will present a reduction to a more general property tester for so called weighted words that we will develop for this purpose.

6.5 Weighted words

We present an approximate membership tester for finite automata on weighted words modulo a weighted edit distance.

6.5.1 From trees to weighted words

A weighted word over an alphabet Σ is a word over the alphabet $\Sigma \times \mathbb{N}$. The idea for the introduction of weigthed words is as follows. To any node v of a tree t we assign the weight $|t_{|v}|$. The weighted word associated to a node v is then the word $word(t_{|v})$, in which each position is weighted by the weight of the corresponding child of v.

We next illustrate the close link from trees to weighted words by example. We consider the DTD D with rules $r \to ab^*, a \to a^*, b \to b^*$. For any $i \geqslant 0$, let a_i be the tree $a(a, \ldots, a)$ with i a-leaves and b_i the tree $b(b, \ldots, b)$ with i b-leaves. The tree $t = r(a_1, b_2, b_3, a_4)$ of depth 2 is clearly invalid for D. Its distance is $d(t, D) = 5$ since one must delete the whole last subtree to become valid and this subtree has size 5. However, if we consider the regular language below the root $L = ab^*$ and pick the word at the root $w = word(t_{|\varepsilon}) = abba$, then we have $d(w, L) = 1$ for the edit distance for words. One way to understand the problem is that we cannot simply ignore the sizes of the subtrees as we did. Instead, we should associate a weight to each position, and consider the weighted word $\boldsymbol{\omega} = (a, 2)(b, 3)(b, 4)(a, 5)$ for the above example. We also need to adapt the costs of deleting a weighted letter such as (c, i) to its weight i. In this way, the weighted distance of $\boldsymbol{\omega}$ to L becomes 5 which is equal to the distance of t to D.

Any weighted word has the form $w * p$ for some $w \in \Sigma^*$ and $p \in \mathbb{N}^*$, where w and p have the same length. We call w the word part and p the weight part of $w * p$. We will also say that $\boldsymbol{\omega}$ has at position i the weight $k \in \mathbb{N}$ and the label $a \in \Sigma$ if $\boldsymbol{\omega}[i] = (a, k)$. The weight $|\boldsymbol{\omega}|_*$ is the sum of the weights at all positions of $\boldsymbol{\omega}$. The word part of a weighted word is used to define its membership to word regular languages while the weight part is used to define weighted words edit distance. We say that a weighted word $w * p$ is recognized by an NFA A if and only if $w \in L(A)$. The set of weighted words recognized by A is denoted by $\mathcal{L}_*(A)$.

The notions of blocking fragment and interval (of Section 5.4) are lifted to weighted words by ignoring weights. For example, if A is a productive automaton recognizing $L = ab^*$, then the interval $I =]0, 2]$ of $\boldsymbol{\omega} = (a, 1)(a, 3)(b, 4)$ is blocking for A, since aa is the word part of the weighted word $\boldsymbol{\omega}I$ located at I, and after reading the first a, A cannot proceed with any second "a". The weight of a fragment $F \in dom(w)$ of $\boldsymbol{\omega} = w * p$ is the sum of the weights of all positions in F. That is $|F|_* = \sum_{i \in F} p[i]$.

6.5.2 Edit distance for weighted words

The edit operations for weighted words are essentially the same as for words, i.e, insertions, relabeling, and deletions. The only difference is that the costs of these operations depend on the weights of the letters that are edited. Given a weighted word $\boldsymbol{\omega} = \sigma_1 \ldots \sigma_n$ and a natural number $i \in [0, n]$, the insertion

of a weighted letter $\sigma \in \Sigma \times \mathbb{N}$ following position i in $\boldsymbol{\omega}$ yields the weighted word:

$$ins_{i,\sigma}(\boldsymbol{\omega}) = \sigma_1 \dots \sigma_i \sigma \sigma_{i+1} \dots \sigma_n.$$

The cost of this operation is the weight of σ. The deletion of position $1 \leqslant i \leqslant n$ of w yields the following weighted word:

$$del_i(\boldsymbol{\omega}) = \sigma_1 \dots \sigma_{i-1} \sigma_{i+1} \dots \sigma_n.$$

The cost of such deletion is the weight of the deleted letter σ_i. The relabeling at position $1 \leqslant i \leqslant n$ of $\boldsymbol{\omega}$ into a letter b changes only the letter at this position but not its weight. Let $\sigma_i = (a, k)$ then the relabeling operation at position i costs k and yields:

$$rel_{i,b}(\boldsymbol{\omega}) = \sigma_1 \dots \sigma_{i-1} (b, k) \sigma_{i+1} \dots \sigma_n$$

It should be noticed that the relabeling of a node in a tree has cost 1, but that one may also have to modify the whole subtree in the worst case to become valid with respect to the DTD. The objective is that the global editing cost of a modification of a tree is bounded by the editing cost of its corresponding weighted word with similar operations. We in the following denote the edit distance between weighted words by d_*, and as usual the distance is extended to a distance between a weighted word and any set of weighted words.

6.5.3 Random object of weighted words

We recall that for a weighted word $\boldsymbol{\omega} = w * p$, one says that it is recognized by the word NFA A, iff $w \in L(A)$. Thus membership of a weighted word to the weighted word language $\mathcal{L}_*(A)$ is somehow just membership of its word part to $L(A)$. Thus we design random objects $rdm_{\boldsymbol{\omega}}$ of the weighted word $\boldsymbol{\omega}$, just as a random object of the word part w. However since the edit distance of weighted words is defined according to their weight part, we use such weight part for the randomization of the random object $rdm_{\boldsymbol{\omega}}$. Thus we define a random object of the weighted word $\boldsymbol{\omega}$ just as a non-uniform random object to w. And the randomization considered for representations of w will be weighted by the weight part of $\boldsymbol{\omega}$. We remind that the vocabulary for structures representing words over the alphabet Σ is :

$$\sigma_{words} = \{(lab_a, 1) \mid a \in \Sigma\} \uplus \{(<, 2), (succ, 2), (start, 1)\}$$

The random and deterministic vocabularies of words over Σ was also defined as: $\sigma_{rand} = \{<^2, succ^2, start^1\}$ and $\sigma_{det} = \sigma_{words}$ respectively.

Now we define random objects of weighted words over the alphabet Σ. We next consider only such random objects for approximate membership of weighted words for NFAs. We remind that all words over Σ defines a relational σ_{words}-structure S_w with domain $dom(w)$ (see Section 3.1.1).

Definition 6.3. *Let Σ be an alphabet and $\boldsymbol{\omega} = w * p$ a weighted word over Σ. A random object for $\boldsymbol{\omega}$ is any oblivious $(\sigma_{det}, \varnothing, \sigma_{rand}, \mathcal{D}, \perp)$-random object of some structure Γ isomorphic to S_w, such that the σ_{rand}-randomization $(\mathcal{D}, \perp)$ of Γ is f-weighted. Where $f : dom(\Gamma) \to \mathbb{N}$ is the function defined with the isomorphism $\theta : \Gamma \to S_w$ in the following way: for all $e \in dom(\Gamma)$, if $\theta(e) \neq 0$ then $f(e) = p[\theta(e)]$ else $f(e) = 1$.*

We will abuse notations and denote all random objects of $\boldsymbol{\omega}$ by $rdm_{\boldsymbol{\omega}}$. Thus using random objects to a weighted word $\boldsymbol{\omega}$, one only accesses the word part of $\boldsymbol{\omega}$ without knowing its exact weight part. However one knows that all elements generated by randomized queries of $rdm_{\boldsymbol{\omega}}$ are done with probabilities defined by the weight part of $\boldsymbol{\omega}$. We now give examples of queries to the random object $rdm_{\boldsymbol{\omega}}$, where $\boldsymbol{\omega} = w * p$. These queries are essentially the same as for random objects of words (see Section 4.2.1). Note that $rdm_{\boldsymbol{\omega}}$ can access any structure Γ isomorphic to S_w. However to give a clear intuition of what queries of rdm_Γ do we describe them as if they access the structure S_w.

- $\mathcal{R}(start)$: Returns 0 in w.
- $\mathcal{R}(<, \mathcal{R}(start))$: Select a position i of w with probability $\frac{p[i]}{\sum_{j \in pos(w)} p[j]}$, or $\perp$ if w is the empty word.
- $\mathcal{R}(succ, e)$: Returns $i+1$; where i is the position of w corresponding to the element $e \in dom(\Gamma)$.
- $\mathcal{R}(<, e)$: Selects an position $i' > i$ with probability $\frac{p[i']}{\sum_{j \in pos(w), j > i} p[j]}$, where i is the position of w corresponding to the element $e \in dom(\Gamma)$; or returns $\perp$ if such position does not exist.
- $\mathcal{B}_<(e, e')$: Tells whether e' corresponds to $i+1$; where i is the position of w corresponding to the element $e \in dom(\Gamma)$.
- $\mathcal{B}_{lab_a}(e)$: Tells whether i is labelled $a \in \Sigma$; here i is the position of w corresponding to the element $e \in dom(\Gamma)$;

Since the randomization used for a weighted word $\boldsymbol{\omega}$ is weighted by its weight part, in the following we will say that one draws positions of $\boldsymbol{\omega}$ according to the weight distribution of $\boldsymbol{\omega}$, whenever one uses the query $\mathcal{R}(<, \mathcal{R}(start))$. Using this random object we next study approximate membership of weighted word for NFAs.

6.5.4 Testing weighted words

In approximate membership of weighted words for languages denoted by NFAs, the notion of ϵ-farness is adapted to the weight of the weighted word instead of using the length of its word part. This is defined in the following definition. We recall that any distance

Definition 6.4. *A weighted word $\boldsymbol{\omega}$ is ϵ-close to an NFA A iff and only $d_*(\boldsymbol{\omega}, \mathcal{L}_*(A)) \leqslant |\boldsymbol{\omega}|_*$. When $\boldsymbol{\omega}$ is not ϵ-close to A, we say that $\boldsymbol{\omega}$ is ϵ-far from A.*

We recall in the following definition the notion of approximate membership tester for weighted words and properties (language) denoted by NFAs.

Definition 6.5. *An approximate membership tester for weighted words and NFAs is an algorithm (possibly random) that inputs: a precision $\epsilon > 0$, an NFA A, and a random object $rdm_{\boldsymbol{\omega}}$ of some weighted word $\boldsymbol{\omega}$ and answers with probability $\frac{2}{3}$ CLOSE if $\boldsymbol{\omega}$ is close to A and NO if $\boldsymbol{\omega}$ is ϵ-far from A.*

Before going further in the approximate membership testing of weighted words for languages defined by NFAs, we mention that weighted words are just artefacts that we be used for trees approximate membership for languages denoted by DTDs, under the strong edit distance. Therefore one will simulate random objects of weighted words using the ones of trees. This will be fully explained in Section 6.6.

We next show that approximate NFA membership modulo the edit distance can be tested efficiently for weighted words. We will prove the following result for the randomized data model of weighted words defined above.

Theorem 6.3. *Let A be an NFAs with alphabet $|\Sigma|$ and k strongly connected components. Whether a weighted word $\boldsymbol{\omega}$ is approximately a member of $\mathcal{L}_*(A)$ modulo the weighted edit distance with error precision ϵ can be tested with query complexity $O(\frac{k^2|\Sigma||A|}{\epsilon}\log^3(\frac{k|A|}{\epsilon}))$ and time complexity $O(\frac{k^2|\Sigma||A|^2}{\epsilon}\log^3(\frac{k|A|}{\epsilon}))$ independently of the weight or size of $\boldsymbol{\omega}$.*

One way to notice that $\boldsymbol{\omega}$ does not belong to the language of A is to find a blocking factor, i.e. a factor on which A cannot be run from any state. A more general way is to find a subword blocking for A. Whenever another factor starts after a hole in the subword, then one must start with states that can be reached by some run on the previous factor, while jumping over the holes. As we will see we can test membership of a weighted word $\boldsymbol{\omega}$ approximately, by randomly drawing sufficiently many factors of $\boldsymbol{\omega}$ and running A on them in order to check whether the obtained weighted subword is blocking.

So far, the above ideas are the same as for usual words. What changes for weighted words is that the cost of errors can be concentrated at some

position of high weight. However since the random object of weighted words draws positions according to their weight, we show that with respect to such random object for weighted words, Theorem 6.3 is true.

As we have done for property testing of words in the last chapter, we will prove this in two steps. We first consider the case where the NFA is strongly connected (i.e. where all states are reachable from any other state), and secondly we study the general case of automaton with several strongly connected components.

6.5.5 Strongly connected automata

Let A be an NFA that is strongly connected. An approximate membership testing for weighted words can proceed as follows. The input is a randomized data model $rdm_{\boldsymbol{\omega}}$ for some weighted word $\boldsymbol{\omega}$. The tester then generates randomly sufficiently many positions of the word according to their weights, reads sufficiently long factors starting there, and returns NO if one of them is blocking. What "sufficient" here means can be deduced from the following Lemma.

Lemma 6.4. *Let $A = (\Sigma, Q, init, fin, \Delta)$ be an* NFA *that is strongly connected, $\boldsymbol{\omega} = w * p$ a weighted word, m a natural number bigger than $|\boldsymbol{\omega}|_*$, $\epsilon > 0$ an error precision, and $\gamma = \frac{8|Q|}{\epsilon}$. If $d(\boldsymbol{\omega}, A) > \epsilon m$ and $m \geqslant 8\gamma\lceil\log(\gamma)\rceil$, then:*

1. *Either the set of positions i so that $w[i]$ is blocking has overall weight at least m/γ, or*
2. *There is a length $l \in [2, \gamma]$ which is a power of 2 such that the number of intervals I of length $2l$ that are blocking for A is at least $m\beta_l$ with $\beta_l = l/(2\gamma\lceil\log(\gamma)\rceil)$.*

When applied with $m = |\boldsymbol{\omega}|_*$, observe that this Lemma only holds for suffiently heavy $\boldsymbol{\omega}$'s where $m \geqslant 8\gamma\lceil\log\gamma\rceil$. For other weighted words, one can simply check exact membership directly while reading them entirely with low costs. In fact, for sufficiently heavy weighted words that are ϵ-far, if for all $l \in [2, \gamma]$, we draw $O(1/\beta_l)$ positions with the weighted distribution, then with high probability either one of the selected positions is blocking or it starts a blocking interval of size $2l$.

Proof. We consider the collection of intervals obtained in the following way: $I_1 =]i_0, i_1]$ where $i_0 = 0$ and i_1 is the smallest index such that $]i_0, i_1]$ is blocking for A (possibly $i_1 = 1$). Then iteratively, $I_j =]i_{j-1}, i_j]$ such that again i_j is the smallest index that makes I_j blocking for A, until I_k where $i_k = |\boldsymbol{\omega}|$. For any j, let $I'_j =]i_{j-1}, i_j - 1]$ be the interval obtained by removing the last letter of I_j, by definition I'_j is either empty or non-blocking.

We consider the weighted word $\boldsymbol{\omega}'$ of $\mathcal{L}_*(A)$ obtained in the following way: $\boldsymbol{\omega}'_0 = \varepsilon$ and for each $1 \leqslant j \leqslant k$ if $\sigma_j = \boldsymbol{\omega}[i_j]$ is non blocking for A, then $\boldsymbol{\omega}'_j = \boldsymbol{\omega}'_{j-1} \cdot \zeta \cdot \boldsymbol{\omega} I'_j \cdot \zeta' \cdot \sigma_j$ where ζ and ζ' are inserted weighted words that allow $\boldsymbol{\omega}'_j$ to be non blocking: as A is strongly connected, there exists such ζ and ζ' of length and weight less than $|Q|$. Else if σ_j is blocking, this correction can not be done and σ_j is erased and then we pick $\boldsymbol{\omega}'_j = \boldsymbol{\omega}'_{j-1} \cdot \zeta \cdot \boldsymbol{\omega} I'_j$. For $\boldsymbol{\omega}_1$, ζ is also chosen such that $\boldsymbol{\omega}_1$ can be read from *init*. The weighted word $\boldsymbol{\omega}'$ is then $\boldsymbol{\omega}'_k \cdot \zeta'$ where ζ' makes $\boldsymbol{\omega}'$ a word of $\mathcal{L}_*(A)$ (again, $|\zeta'|_*$ can be less than $|Q|$).

If B is the subset of $\{i_1, \dots i_k\}$ containing all positions i such $\sigma_i = \boldsymbol{\omega}[i]$ that constitute blocking fragments, then obtaining $\boldsymbol{\omega}'$ from $\boldsymbol{\omega}$ costs:

$$(k+1)|Q| + |B|_* + (k - |B|)|Q|$$

The first part of this summation corresponds to insertion of weighted words ζ, the second to the deletion of elements of B, and the last to the insertion of weighted words ζ'. The real distance $d(\boldsymbol{\omega}, A)$ is smaller than this summation, but bigger than ϵm by hypothesis. So, $\epsilon m \leqslant |B|_* + |Q|(2k + 1 - |B|)$, and thus $2k + 1 - |B| \geqslant (\epsilon m - |B|_*)/|Q| = 8m/\gamma - |B|_*/|Q|$.

We distinguish two cases. If $|B|_* \geqslant m/\gamma$ then the lemma is true by condition 1. In the other case $|B|_* < m/\gamma$, and hence $|B| < m/\gamma$. Thus, $2k + 1 - |B| \geqslant 7m/\gamma$ and since $m \geqslant 8\gamma \log \gamma$, it follows in particular that $m \geqslant \gamma$. Hence $m/\gamma > 1$, so that $2k - |B| \geqslant 6m/\gamma$, and also $k \geqslant 3m/\gamma$.

We now want to estimate the number of 'small' intervals I_j: i.e. the ones whose length are less than γ. For a value l which is a power of 2, let $\mathcal{I}_l = \{I_j \mid 1 \leqslant j \leqslant k, l/2 < |I_j| \leqslant l\}$. Note that the number of 'big' intervals (whose length is more than γ) is at most $|\boldsymbol{\omega}|/\gamma \leqslant m/\gamma$, and so, the number of small intervals is at least $k - m/\gamma$. Since $\cup_{i=1}^{\lceil \log(\gamma) \rceil} \mathcal{I}_{2^i}$ is a partition of the set of small intervals I_j, the above lower bound on k implies:

$$\sum_{i=1}^{\lceil \log(\gamma) \rceil} |\mathcal{I}_{2^i}| \geqslant k - m/\gamma \geqslant 2m/\gamma$$

Therefore, there exists a number $l = 2^i$ with $2 \leqslant l \leqslant \gamma$ and $|\mathcal{I}_l| \geqslant 2m/\gamma \lceil \log(\gamma) \rceil$. We fix such an index l. We are now interested in the cardinality of the set W_{2l}, which contains all intervals of w of size in $]l/2, 2l]$ that are blocking for A. We next estimate the cardinality of W_{2l}. Every interval of $\mathcal{I}_l$, except the two leftmost and the two rightmost ones, is included in $[l, |\boldsymbol{\omega}| - l]$, and thus belongs to l intervals of W_{2l}. Conversely, every interval of W_{2l} may contain at most 3 intervals from $\mathcal{I}_l$, since the latter are non-overlapping and of size

at least $1 + l/2$. Hence:

$$\begin{array}{rcl} |W_{2l}| & \geqslant & \frac{l\ (|\mathcal{I}_l|-4)}{3} \\ & \geqslant & \frac{l}{3}(\frac{2m}{\gamma\lceil\log(\gamma)\rceil} - 4) \\ & & \text{since } |\mathcal{I}_l| \geqslant 2m/\gamma\lceil\log(\gamma)\rceil \\ & \geqslant & \frac{l}{3}(\frac{2m}{\gamma\lceil\log(\gamma)\rceil} - \frac{m}{2\gamma\lceil\log(\gamma)\rceil}) \\ & & \text{since } m \geqslant 8\gamma\lceil\log(\gamma)\rceil \\ & = & \frac{ml}{2\gamma\lceil\log(\gamma)\rceil} \ = \ m\beta_l \end{array}$$

This ends the proof of the Lemma as the condition 2 is true. □

6.5.6 General automata

We generalise the previous result to automata with multiple strongly connected components. We refine the results from Chapter 5 [see also Ndione, Lemay, and Niehren, 2013], which in turn adapt the schema from [Alon et al., 2000b], and prove that drawing positions from the weight distribution of ϵ-far weighted words, yields a blocking fragment with high probability.

For integers l, α, weighted word $\boldsymbol{\omega}$ and a sequence $S = (i_1, \cdots, i_\alpha)$ of α positions in $\boldsymbol{\omega}$, we denote by F^l_S the fragment $\cup_{1\leqslant j\leqslant\alpha}[i_j, \min(i_j + l, |\boldsymbol{\omega}|)]$. And we define $\mathcal{S}(\boldsymbol{\omega}, \alpha)$ as the set of sequences of α positions (not necessarily distinct) of $\boldsymbol{\omega}$. We say that $S \in \mathcal{S}(\boldsymbol{\omega}, \alpha)$ is l-blocking for an NFA A if the F^l_S is blocking for A, and we say that S is l-nonblocking for A otherwise.

Lemma 6.5. *Let A be a productive* NFA *with state set Q and k strongly connected components. Let $\epsilon > 0$, $\gamma' = \frac{16k|Q|}{\epsilon}$ and $\boldsymbol{\omega}$ be a weighted word of weight greater than $8\gamma'\lceil\log(\gamma')\rceil$. If $\boldsymbol{\omega}$ is ϵ-far from A, then there exists a power of two $l \in [2, \gamma']$ such that: with probability $\frac{5}{6}$, drawing $\alpha_l = 30k\gamma'\lceil\log(\gamma')\rceil^2/l$ positions with the weight distribution of $\boldsymbol{\omega}$ yields some $2l$-blocking sequence $S \in \mathcal{S}(\boldsymbol{\omega}, \alpha_l)$. That is a sequence $S = (i_1, \cdots, i_{\alpha_l})$ such that the fragment $F^{2l}_S = \cup_{1\leqslant j\leqslant\alpha}[i_j, \min(i_j + 2l, |\boldsymbol{\omega}|)]$ is blocking for A.*

The idea of the proof of Lemma 6.5 is to characterize $2l$-nonblocking fragments of $\mathcal{S}(\boldsymbol{\omega}, \alpha_l)$ by strongly connected components and then bound their probability of being selected by the drawing process. This is done in Claims 6.6 and 6.7 in the following. This idea is essentially the same that we used for approximate membership of words in NFAs (see Section 5.6).

Let $A = (\Sigma, Q, \mathit{init}, \mathit{fin}, \Delta)$ be a productive NFA. A connected component of A is a maximal subset of states of A that is strongly connected. Note that the strongly connected components of A form a partition of Q. Let k be the number of strongly connected components of A. We define a component path as a sequence $\Pi = (Q_1, \ldots, Q_j)$ of pairwise distinct strongly connected components of A such that for all $1 < h \leqslant j$ some state of Q_h is reachable

from some state of Q_{h-1}. note that the length of any component path is at most k, that is $0 \leqslant j \leqslant k$.

We denote by $A(Q') = (\Sigma, Q', \mathit{init}', \mathit{fin}', \Delta')$, where $Q' \subseteq Q$, the restriction of A on Q', defined as $\mathit{init}' = Q'$, $\mathit{fin}' = Q'$ and $\Delta' = \Delta \cap Q' \times (\Sigma \uplus \{\varepsilon\}) \times Q'$. The restriction of A on a connected path Π is $A(\Pi) = A(Q_1 \cup \ldots \cup Q_j)$.

A decomposition of a weighted word $\boldsymbol{\omega}$ along a component path $(Q_1, \ldots, Q_j)$ is a sequence of integers $(i_0, \ldots, i_j)$ such that $0 = i_0 < \ldots < i_j = |\boldsymbol{\omega}|$. Notice that any decomposition depends on the length of $\boldsymbol{\omega}$ and the length of the component path (which we will sometimes leave implicit in the context). $\boldsymbol{\omega}$ is said ϵ-far from $(Q_1, \ldots, Q_j)$ if for all decompositions $0 = i_0 < \ldots < i_j = |\boldsymbol{\omega}|$ there exists some integer h such that:

$$d(\boldsymbol{\omega}]i_{h-1}, i_h], A(Q_h)) > \epsilon|\boldsymbol{\omega}|_*$$

We next claim that ϵ-far weighted words are also far from any component path of A :

Claim 6.6. *Let $A = (\Sigma, Q, \mathit{init}, \mathit{fin}, \Delta)$ be a productive automaton with k strongly connected components and let $\boldsymbol{\omega}$ be a weighted word of weight greater than $\frac{2(k+1)|Q|}{\epsilon}$. If $\boldsymbol{\omega}$ is ϵ-far from $L(A)$ then it is $\frac{\epsilon}{2k}$-far from any component path of A.*

Proof. The proof is by contrapositive. Let $\boldsymbol{\omega}$ be a weighted word $\frac{\epsilon}{2k}$-close to some component path $(Q_1, \ldots, Q_j)$ of A. We next show that $\boldsymbol{\omega}$ is ϵ-close to A. By assumption, there exists some decomposition $0 = i_0 < \ldots < i_j = |\boldsymbol{\omega}|$ such that for all natural numbers $1 \leqslant h \leqslant j$:

$$d(\boldsymbol{\omega}]i_{h-1}, i_h], A(Q_h)) < \frac{\epsilon|\boldsymbol{\omega}|_*}{2k}$$

Hence, we can correct each factor $\boldsymbol{\omega}]i_{h-1}, i_h]$ into a word $\boldsymbol{\omega}_h$ of $L(A(Q_h))$ with at most $\frac{\epsilon|\boldsymbol{\omega}|_*}{2k}$ edit operations. We then define $\boldsymbol{\omega}'_1 = \boldsymbol{\omega}_1$ and recursively for each $1 < h \leqslant j$, $\boldsymbol{\omega}'_h = \boldsymbol{\omega}'_{h-1} \cdot \zeta_h \cdot \boldsymbol{\omega}_h$, where ζ_h has weight at most $|Q|$ and is chosen such that w'_h is non blocking for A. Note that this is possible since every connected component Q_h is reachable from Q_{h-1}. To finally obtain a word of $L(A)$, we transform $\boldsymbol{\omega}'_j$ into $\boldsymbol{\omega}' = \zeta \cdot \boldsymbol{\omega}'_j \cdot \zeta'$, where ζ and ζ' have weight again at most $|Q|$, this last operation is possible since A is productive. The overall cost of these operations is $\frac{\epsilon|\boldsymbol{\omega}|_*}{2k} \cdot j + 2|Q| + (j-1)|Q| = \frac{|\boldsymbol{\omega}|_* j}{2k} + (j+1)|Q|$. Therefore

$$d(\boldsymbol{\omega}, A) \leqslant d(\boldsymbol{\omega}, \boldsymbol{\omega}') \leqslant \frac{|\boldsymbol{\omega}|_* j}{2k} + (j+1)|Q|$$

This inequality, $j \leqslant k$, and $\boldsymbol{\omega} \geqslant \frac{2(k+1)|Q|}{\epsilon}$ yield that $d(\boldsymbol{\omega}, A) \leqslant \epsilon|\boldsymbol{\omega}|_*$. □

So for weighted words, $\boldsymbol{\omega}$, ϵ-far from A, and for all component path $(Q_1, \ldots, Q_j)$ and decomposition $(i_0, \ldots, i_j)$, there exists some integer $1 \leqslant h \leqslant j$ such that:

$$d(\boldsymbol{\omega}]i_{h-1}, i_h], A(Q_h)) > \frac{\epsilon}{2k}|\boldsymbol{\omega}|_*$$

For such h one can apply Lemma 6.4 with the factor $\boldsymbol{\omega}]i_{h-1}, i_h]$ and $A(Q_h)$. Hence rephrasing Lemma 6.4, there is a set, with important weight, of positions that all start small intervals blocking for $A(Q_h)$. Such intervals are witnessing the fact that the component path doesn't fit the weighted word on the considered decomposition. This fact is formally expressed in the next claim.

Claim 6.7. *Let $A = (\Sigma, Q, init, fin, \Delta)$ be a productive automaton with k strongly connected components. let $\gamma' = \frac{16k|Q|}{\epsilon}$ and $\boldsymbol{\omega}$ be a weighted word of weight greater than $8\gamma'\lceil\log(\gamma')\rceil$. If $\boldsymbol{\omega}$ is ϵ-far from $L(A)$ then there exists a power of two $l \in [2, \gamma']$ such that for all component paths $(Q_1, \ldots, Q_j)$ of A and decompositions $(i_0, \ldots, i_j)$, there exists some integer $1 \leqslant h \leqslant j$ and a set $\mathfrak{B}$ of positions which all start subintervals of $]i_{h-1}, i_h]$ blocking for $A(Q_h)$ and of length less than $2l$. Furthermore the overall weight, $|\mathfrak{B}|_* = \sum_{i \in \mathfrak{B}} |\boldsymbol{\omega}[i]|_*$, of $\mathfrak{B}$ is greater than $|\boldsymbol{\omega}|_* \beta'_l$. Where $\beta'_l = l/(2\gamma'\lceil\log(\gamma')\rceil)$.*

Proof. Let $\boldsymbol{\omega}$ be ϵ-far from $L(A)$ with weight greater than $8\gamma'\lceil\log(\gamma')\rceil$. Then $|\boldsymbol{\omega}|_* \geqslant \frac{2(k+1)|Q|}{\epsilon}$ and we can apply Claim 6.6. Hence for any component path $(Q_1, \ldots, Q_j)$ of A and decomposition $(i_0, \ldots, i_j)$ there exists some integer h such that:

$$d(\boldsymbol{\omega}]i_{h-1}, i_h], L(A(Q_h)) > \frac{\epsilon}{2k}|\boldsymbol{\omega}|_*$$

Since $|\boldsymbol{\omega}|_* \geqslant 8\gamma'\lceil\log(\gamma')\rceil$, we can apply Lemma 6.4 for factor $\boldsymbol{\omega}]i_{h-1}, i_h]$, precision value $\epsilon' = \epsilon/2k$ and γ' (instead of $\boldsymbol{\omega}$, ϵ, γ). Then there exists a power of two $l' \in [2, \gamma']$ such that there is a set of positions with overall weight at least $\beta'_{l'}|\boldsymbol{\omega}|_*$, whose elements start subintervals of $]i_{h-1}, i_h]$, of length less than $2l'$, blocking for $A(Q_h)$. Let l be the least l' for all path $(Q_1, \ldots, Q_j)$ and decomposition $(i_0, \ldots, i_j)$. As $\beta'_{l'}$ grows monotonically with l', the claim follows. □

However the set of decompositions on $\boldsymbol{\omega}$ might have a size that is depending on the length of $\boldsymbol{\omega}$. So clearly in order to witness with constant query complexity, the ϵ-farness of $\boldsymbol{\omega}$ from A, one can not consider all component paths with all possible decompositions. Nevertheless we overcome this difficulty by characterizing special fragments that are blocking for A with a bounded number of positions. The set of fragments we consider are those we obtain by a union of sequence of small intervals. Such special fragments consisting of the fragments F^l_S for sequences of positions $S \in \mathcal{S}(\boldsymbol{\omega}, \alpha)$, for some integers l, α. We recall that $\mathcal{S}(\boldsymbol{\omega}, \alpha)$ is the set of sequences $S = (i_1, \cdots, i_\alpha)$ of α positions of $\boldsymbol{\omega}$. And $F^l_S = \cup_{1 \leqslant j \leqslant \alpha}[i_j, \min(i_j + l, |\boldsymbol{\omega}|)]$.

Now here is the proof Lemma 6.5.

Proof. The case of $k = 1$ where A is strongly connected follows directly from lemma 6.4. So we suppose $k \geqslant 2$ in the rest of this proof. Let $\boldsymbol{\omega}$ be a weighted word satisfying $|\boldsymbol{\omega}|_* \geqslant 8\gamma'\lceil\log(\gamma')\rceil$.

By Claim 6.7, there is a power of two $l \in [2, \gamma']$, such that for all component paths $(Q_1, \dots, Q_j)$ of A and decompositions $(i_0, \dots, i_j)$, there exists some integer $1 \leqslant h \leqslant j$ and a set $\mathfrak{B}$ of positions with weight at least $|\boldsymbol{\omega}|_* \beta'_l$ and whose elements start subintervals of $]i_{h-1}, i_h]$, of length less than $2l$, blocking for $A(Q_h)$. Let $\lambda = |\boldsymbol{\omega}|_* \beta'_l/5$, we recursively define $i_0 = 0$, $\Lambda_0 = \{0\}$ and for each $1 \leqslant o \leqslant 5/\beta'_l + 1$, $\Lambda_o = \Lambda_{o-1} \cup \{i_o\}$, where $i_o > i_{o-1}$ is the least position satisfying $|\boldsymbol{\omega}]i_{o-1}, i_o]|_* \geqslant \lambda$. Furthermore Λ is obtained by adding to Λ_{5/β'_l+1} all positions with weight greater than λ. It is straightforward that Λ contains at most $10/\beta'_l$ elements.

Now let $S = (i_1, \cdots, i_{\alpha_l})$ be a sequence of positions obtained with a drawing according to the weight distribution of $\boldsymbol{\omega}$. To estimate the probability that S is $2l$-blocking we will bound the probability of the opposite event. If $S \in \mathcal{S}(\boldsymbol{\omega}, \alpha_l)$ is $2l$-nonblocking for A, then clearly there exists some connected component $(Q_1, \dots, Q_j)$ of A and a decomposition $(i_0, \dots, i_j)$ such that no position of S is present in the corresponding set $\mathfrak{B}$. Moreover $\mathfrak{B}$ is a subset of $]i_{h-1}, i_h]$ with weight 5λ, therefore there are positions $i, i' \in \Lambda \cap]i_{h-1}, i_h]$ such that $i \leqslant i'$ and $\mathfrak{B} \cap [i, i']$ is of weight λ. Indeed if $]i_{h-1}, i_h]$ have a position of weight λ, then take i and i' being that position. Otherwise take $i, i' \in \Lambda \cap]i_{h-1}, i_h]$ such that $i' - i$ is maximal, then by definition both $]i_{h-1}, i[$ and $]i', i_h]$ contain at most a subset of $\mathfrak{B}$ of weight 2λ, so $\mathfrak{B} \cap [i, i']$ is of weight $5\lambda - 4\lambda = \lambda$.

Hence S is $2l$-nonblocking (i.e. F_S^{2l} is blocking for A) if only there exists some connected component Q_h of A and $i, i' \in \Lambda$ such that: there exits a set of positions $\mathfrak{B}' \subseteq [i, i']$ of weight $|\mathfrak{B}|'_* \geqslant \lambda$ satisfying that none of the positions of S are in $\mathfrak{B}'$, and all positions of $\mathfrak{B}'$ start subintervals of length less than $2l$, blocking for Q_h. For any fixed Q_h and $i, i' \in \Lambda$ the probability of such event to occur is at most: $((|w|_* - \lambda)/|w|_*)^{\alpha_l}$. we bound the probability of having a $2l$-nonblocking sequence S by using a union bound on all possible Q_h and $i, i' \in \Lambda$. Since the set of connected components is of size at most k and $|\Lambda| \leqslant 10/\beta'_l$. Thus we have the following bound :

$$
\begin{aligned}
\tfrac{100k}{\beta_l'^2} \cdot (\tfrac{|w|_* - \lambda}{|w|_*})^{\alpha_l} &= \tfrac{100k}{\beta_l'^2} \cdot (1 - \beta'_l/5)^{\alpha_l} \\
&\leqslant \tfrac{100k}{\beta_l'^2} \cdot e^{-\beta'_l \alpha_l / 5} \\
&\leqslant \tfrac{100k}{\beta_l'^2} \cdot e^{-3k \log(\gamma')} \\
&\text{since } \alpha_l \geqslant 15k \log(\gamma')/\beta'_l \\
&\leqslant \tfrac{100k}{\beta_l'^2} \cdot (\tfrac{1}{\gamma'})^{3k} \\
&\leqslant 100k\gamma'^2 \lceil \log(\gamma') \rceil^2 \cdot (\tfrac{1}{\gamma'})^{3k} \\
&\text{since } \beta'_l \geqslant 1/\gamma' \lceil \log(\gamma') \rceil
\end{aligned}
$$

$$\begin{aligned}
\tfrac{100k}{\beta_l'^2} \cdot (\tfrac{|w|_* - \lambda}{|w|_*})^{\alpha_l} &\leqslant 100k \cdot (\tfrac{1}{\gamma'})^{3k-3} \\
&\text{since } \lceil \log(\gamma') \rceil^2 \leqslant \gamma' \\
&\leqslant 100k \cdot (\tfrac{1}{\gamma'})^{3} \\
&\leqslant 100k \cdot (\tfrac{1}{\gamma'})^{3} \\
&\text{since } k \geqslant 2 \\
\tfrac{100k}{\beta_l'^2} \cdot (\tfrac{|w|_* - \lambda}{|w|_*})^{\alpha_l} &\leqslant \tfrac{1}{\gamma'} \leqslant \tfrac{1}{6} \\
&\text{since } \gamma' \geqslant 16k
\end{aligned}$$

□

Finally, Theorem 6.3 is a consequence of Lemma 6.5. Indeed, the algorithm in Figure 6.3 is a one sided membership tester for weighted words. To see this notice that whenever the algorithm in Figure 6.3 inputs a random object $rdm_{\boldsymbol{\omega}}$ of some weighted word $\boldsymbol{\omega} = w * p$, it first starts by verifying whether its length is greater than $l_{min} = 8\gamma'\lceil \log(\gamma') \rceil$. If it is a small weighted word then exact membership is checked by running the input automaton on w. Thus it provides a valid answer. So we can restrict the following discussion to weighted words of length greater than l_{min}. For A-valid such weighted words, the algorithm will never find a blocking fragment; since A-valid weighted words never contain a fragment blocking for A. So the algorithm always returns the correct CLOSE answer for input A-valid weighted words. Now for ϵ-far weighted words, one can apply Lemma 6.5 since they have also weight greater than l_{min}. Therefore whenever the algorithm draws a sequence $S \in \mathcal{S}(\boldsymbol{\omega}, \alpha_l)$ of positions with the weight distribution of $\boldsymbol{\omega}$, the algorithm finds a blocking fragment F_S^{2l} with probability at least $\frac{5}{6}$. Thus the algorithm answers correctly NO for input long enough ϵ-far weighted words. This proves that the algorithm in Figure 6.3 is a one sided membership tester for weighted words and that Theorem 6.3 is true.

6.6 Testing unranked trees

We reduce DTD approximate membership of trees to NFA approximate membership of weighted words. Let Σ be some alphabet. For trees $\mathrm{t} \in \mathcal{T}_\Sigma$, the reduction is based on the weighted words $\boldsymbol{\omega}_v = word(t_{|v}) * p_v$ for nodes v of t, where p_v is the sequence of sizes $|t_{|v'}|$ of the subtrees rooted at the children v' of v in document order. We already mentioned that our interest

```
fun member(rdm_ω, ϵ, A)
  // rdm_ω is the (σ_det, ∅, σ_rand, D, ⊥)−random object of some weighted word ω
  // ϵ is an error precision
  // A = (Σ, Q, Δ, init, fin) is an NFA
  let B ∪ S ∪ {R} = rdm_ω the set of queries of the random object.
  let k be the number of strongly connected components of A
  let γ' = 16k|Q| / ϵ
  if |ω| < 8γ'⌈log(γ')⌉ then
     //to test whether |ω| < 8γ'⌈log(γ')⌉, one starts with the query R(start) and
          moves on iterating 8γ'⌈log(γ')⌉ times with R(<,.) queries if ⊥ is
        returned then it the inequality holds.
     if ω ∈ L(A) //run A via start, succ, lab
     then return CLOSE else return NO
  else
     for i = 1 to ⌈log(γ')⌉ do
       let l = min(2^i, γ')
       let α_l = 30kγ'⌈log(γ')⌉^2/l
       let S be sequence of α_l positions of ω randomly drawn by using the
           query R(<, R(start))
       let F_S^2 l be the union of all intervals of ω of length 2l starting at
           positions in S. F_S is obtained using queries of the form R(<,.),
            B(lab_a,.); for all a ∈ Σ.
       if F_S^2 l is blocking wrt. A
          // run A via R(<,.) and B(lab_a,.) queries; for all a ∈ Σ.
       then return NO; exit else skip
     end
     return CLOSE
```

Figure 6.3: An approximate membership tester for weighted words.

in studying approximate membership of weighted words for NFAs, is to use weighted words as artefacts that help to solve the approximate membership for DTDs. In this section, with respect to the strong edit distance d_{strong}, we will relate approximate membership for DTDs to the approximate membership of weighted words $\boldsymbol{\omega}_v$ for NFAs, modulo the edit distance d_*. For now we detail the σ_{words}-structures we use as representations of words $word(t_{|v})$, and we show how to simulate random objects $rdm_{\boldsymbol{\omega}_v}$ of such structures with randomization weighted by p_v. See Section 6.3 for the definition of random objects of weighted words.

6.6.1 Simulating weighted words random objects with trees random objects

Let Σ be some alphabet. Let t be a tree over the alphabet Σ and let Γ be a σ_{trees}-structure representing t. For any node $v \in dom(\Gamma)$, we denote the weighted words associated to nodes $v \in dom(\Gamma)$ by $\boldsymbol{\omega}_v$. We recall that weighted words $\boldsymbol{\omega}_v$ are defined as $word(t_{|v}) * p_v$; where $word(t_{|v}) \in \Sigma^*$ is the word of the sequence of children of v and p_v is the sequence of sizes $|t_{|v'}|$ of subtrees rooted at children v' of v in document order. We then use the following σ_{words}-structures Γ_v as representations of words $word(t_{|v})$, and next we explain how to use the random object rdm_{t} of the tree t to simulated the random object $rdm_{\boldsymbol{\omega}_v}$ of $\boldsymbol{\omega}_v$ which accesses Γ_v. The domain and relations of

Γ_v are the followings for all $v \in dom(\Gamma)$.

$$\begin{aligned} dom(\Gamma_v) &= \{v' \mid child_t(v, v')\} \cup \{v\} \\ start^{\Gamma_v} &= \{v\} \\ succ^{\Gamma_v} &= \{(v, v') \mid fc_t(v, v')\} \cup \{(v', v'') \mid v' \neq v, ns_t(v', v'')\} \\ <^{\Gamma_v} &= \{(v', v'') \mid <_t (v', v'')\} \end{aligned}$$

Now we remind that to test approximate membership of the weighted words $\boldsymbol{\omega}_v$ for some NFA A (for some node $v \in dom(\Gamma)$), the tester of Figure 6.3 entirely reads the word part of input weighted words with small length, and for long enough weighted words, it draws enough positions of $\boldsymbol{\omega}_v$ according to the weight distribution of $\boldsymbol{\omega}_v$, and then it checks that the selected sequence of selected positions is locally non blocking for A.

Note that, for the tester of Figure 6.3, and except for the selection of positions of $\boldsymbol{\omega}_v$, it is straightforward that all other queries to the input random object $rdm_{\boldsymbol{\omega}_v}$ of $\boldsymbol{\omega}_v$ can be simulated on structure Γ_v using the random object rdm_{t} on the structure Γ. Thus, to simulate a tester on the weighted words $\boldsymbol{\omega}_v$, using the random objects rdm_{t} of t, we simply need to be able to simulate the drawing of positions of $\boldsymbol{\omega}_v$ with queries to rdm_{t}.

We will not need to compute the size part of $\boldsymbol{\omega}_v$ in order to be able to simulate the random drawing of positions of $\boldsymbol{\omega}_v$ according to the size part. We simply draw a child v' of some node v in the following way. First we select a descendent v'' of v, and then starting from v'', we iteratively uses the parent relation of rdm_{t} to reach a child v' of v. Therefore the probability of this process to select any node v' is exactly $\frac{|t_{|v'}|}{|t_{|v}|-1}$, and thus the probability is according to the weight distribution of $\boldsymbol{\omega}_v$.

Hence, using Theorem 6.3, the previous discussion shows that for all nodes v of t, we can test membership of $\boldsymbol{\omega}_v$ to an NFA A efficiently using only rdm_{t}. However, the previous process for drawing a position v' of $\boldsymbol{\omega}_v$ according to its weight distribution might require $d(v'')$ queries to the parent relation of rdm_{t}. Therefore this process requires at most $d(\mathrm{t})$ to rdm_{t} only for selecting a position. Thus, for testing approximate membership of $\boldsymbol{\omega}_v$ to an NFA A, the complexity measured in term in terms of accesses to rdm_{t} belongs only to $O(d(\mathrm{t}) \cdot \frac{k^2|\Sigma||A|}{\epsilon} \log^3(\frac{k|A|}{\epsilon}))$.

6.6.2 Reducing trees DTDs approximate membership to weighted words NFAs approximate membership

Now we can reduce approximate membership of trees t into DTDs, to the one of weighted words $\boldsymbol{\omega}_v$ into NFAs; knowing that the weighted words $\boldsymbol{\omega}_v$ can be tested using the random object rdm_{t}. In the following lemma, we start by linking the strong tree edit distance between a tree t and a DTD D, to the edit distance of weighted words $\boldsymbol{\omega}_v$ and NFAs defined by D. We define

below the notion of bad nodes which we use for the relation between the two distances in Lemma 6.8

We recall that m_D denotes the mintree size of any DTD D, see Section 6.2.3 for a precise definition of m_D. We will also use the following notations: for a tree t and a node v. The label and depth of v will denoted by $a^v = lab_{\mathrm{t}}(v)$ and $d^v = d(v)$ respectively. And the depth of the subtree $\mathrm{t}_{|v}$ rooted by v will be denoted by $d_v = d(\mathrm{t}_{|v})$. Thus the depth of t can be denoted $d_\varepsilon = d(\mathrm{t})$.

Definition 6.6. *Let $D = (\Sigma, init, (A_a)_{a\in\Sigma})$ be a* DTD*, t be a tree, and let $\epsilon > 0$ be a precision parameter. A node $v \in nod_t$ is called (ϵ, D)-bad if and only if $d(\boldsymbol{\omega}_v, A_{a^v}) > \frac{\epsilon}{m_D\cdot(d^v+1)^2}|\boldsymbol{\omega}_v|_*$, or if there exists some ancestor $v' \in nod_t$ of v such that $d(\boldsymbol{\omega}_{v'}, A_{a^{v'}}) > \frac{\epsilon}{m_D\cdot(d^v+1)^2}|\boldsymbol{\omega}'_v|_*$. Where $d^v = d(\mathrm{t})$ and $a^v = lab_{\mathrm{t}}(v)$ are the depth and label of v, and $a^{v'} = lab_{\mathrm{t}}(v')$ is the label of v'. A node which is not (ϵ, D)-bad is called (ϵ, D)-good.*

Note that a node v is (ϵ, D)-good if all of its ancestors v' have weighted words close to their appropriate language. Thus when v is (ϵ, D)-good, the root of t also satisfies $d(\boldsymbol{\omega}_\varepsilon, A_{a^\varepsilon}) < \frac{\epsilon}{m_D\cdot(d^v+1)^2}|\boldsymbol{\omega}_\varepsilon|_*$. Therefore if a tree t contains only (ϵ, D)-good nodes, then for all nodes $v \in nod_{\mathrm{t}}$, we have:

$$d(\boldsymbol{\omega}, A_{a^v}) < \frac{\epsilon}{m_D \cdot (\max(d_v, d^v) + 1)^2}|\boldsymbol{\omega}_v|_*$$

Where $d_v = d(t_{|v})$, $a^v = lab_{\mathrm{t}}(v)$, $d^v = d(v)$ and $d_v = d(\mathrm{t}_{|v})$.

Next we show that if a tree contains only (ϵ, D)-good nodes then it is ϵ-close to D.

Lemma 6.8. *Let $D = (\Sigma, init, (A_a)_{a\in\Sigma})$ be a* DTD*, $\epsilon > 0$ a precision, and t be a tree such that $lab_{\mathrm{t}}(\varepsilon) = init$. If t contains only (ϵ, D)-good nodes then $d_{strong}(t, D) \leqslant \epsilon|t|$.*

Proof. We recall that for a tree t, we use the following notations for all nodes v of t: $d_v = d(t_{|v})$, $a^v = lab_{\mathrm{t}}(v)$, $d^v = d(v)$ and $d_v = d(\mathrm{t}_{|v})$.

The proof works as it follows. Following weighted words edit operations, on weighted words associated to the nodes of t, we construct a repair strategy of the tree whose overall cost is at most $\epsilon|\mathrm{t}|$. The repair strategy is the following top-down recursive algorithm.

We start from the root of t and follow a transformation of $\boldsymbol{\omega}_\varepsilon$ into $\mathcal{L}_*(A_{init})$ with minimal cost:

1. Insertions of label $a \in \Sigma$ are replaced with insertions of trees of size at most m_D between the corresponding nodes. So every insertion in $\boldsymbol{\omega}_\varepsilon$ corresponds to m_D insertions in the tree t.

2. Any relabelling to a of the position corresponding to some node v is replaced by changing the whole tree $t_{|v}$ to a tree in $L(D_a)$. So every relabelling of positions of $\boldsymbol{\omega}_\varepsilon$ corresponds to a set of edit operations in t with cost less than $m_D\ |t_{|v}|$.

3. Deletion of a position corresponding to some node v, is replaced with deleting the whole tree $t_{|v}$. So every deletion corresponds to a set of tree edit operations with the same cost i.e. $|t_{|v}|$

4. For children v of the root which was not affected by the transformation, we apply the same algorithm on the subtree $t_{|v}$.

For a node $v \in nod_{\mathrm{t}}$, notice that the algorithm whenever it inputs a subtree $\mathrm{t}_{|v}$, it first follows the transformation of $\boldsymbol{\omega}_v$ into $\mathcal{L}_*(A_{a^v})$ with a minimal cost and then uses recursive calls on subtrees $\mathrm{t}_{|v'}$ rooted by children v' of v which aren't affected by the transformation of $\boldsymbol{\omega}_v$ into $\mathcal{L}_*(A_{init})$. Thus all recursive calls are made on subtrees rooted by unchanged nodes. And it is thus straightforward that the result of this algorithm is a member of $L(D)$, since the algorithm corrects every language below every node so that it belongs to the appropriate Nfa.

Now, for a fix tree t, we prove by induction on the depth of nodes that: for all nodes v, the cost of the edit operations made by the algorithm when ever it inputs $\mathrm{t}_{|v}$, is bounded by $\frac{d_v \cdot \epsilon}{d_v,+1}|\mathrm{t}_{|v}|$.

The starting point of the recursive argument is for nodes of depth $d(t)$. Note that such nodes are leaves and thus, for such nodes v, $\boldsymbol{\omega}_v$ is the empty weighted word, and since $d_*(\boldsymbol{\omega}_v, A_{a^v}) \leqslant \frac{d_v \cdot \epsilon}{(\max(d_v, d^v)+)^2}|\boldsymbol{\omega}_v|_* = 0$, the algorithm does not perform any tree edit operation on the subtree $\mathrm{t}_{|v}$. So the cost is 0 which is bounded by $\frac{d_v \cdot \epsilon}{d_v+1}|\mathrm{t}_{|v}|$.

Let us suppose for now on, that for all nodes v' of depth at least $i \in \mathbb{N}$, the cost of the algorithm on the subtrees $\mathrm{t}_{|v'}$ is bounded by $\frac{d_{v'} \cdot \epsilon}{d^{v'}+1}|\mathrm{t}_{|v'}|$.

Then, the cost of the algorithm on subtrees $\mathrm{t}_{|v}$ rooted by any node $v \in nod_{\mathrm{t}}$ of depth $i-1$, can be bound by: the cost of the edit operations which follow the transformation of $\boldsymbol{\omega}_v$, plus the cost of the recursive calls on subtrees rooted by the children v' of v.

To estimate this bound, we notice that any children v' of v satisfy $d_v > d_{v'}$, so $\frac{d_{v'}}{d_{v'}+1} \leqslant \frac{d_v-1}{d_v}$, since the function $x/x+1$ is non decreasing. And also: $d_*(\boldsymbol{\omega}_v, A_{a^v}) \leqslant \frac{\epsilon}{m_D \cdot (\max(d_v, d^v)+1)^2}|\boldsymbol{\omega}_v|_*$, since t contains only good nodes. Moreover the over all size of subtrees on which a recursive call is performed is bounded by $|\mathrm{t}_{|v}|$. Thus the overall cost of the algorithm on the subtree $\mathrm{t}_{|v}$ is bounded by:

$$
\begin{aligned}
R &= m_D \cdot \tfrac{\epsilon}{m_D(d_v+1)^2}|\boldsymbol{\omega}_v|_* + \tfrac{(d_v-1)\cdot\epsilon|\mathrm{t}_{|v}|}{d_v} \\
R &\leqslant \tfrac{\epsilon|\mathrm{t}_{|v}|}{(d_v+1)d_v} + \tfrac{(d_v-1)\cdot\epsilon|\mathrm{t}_{|v}|}{d_v} \qquad \text{since } |\boldsymbol{\omega}_v|_* \leqslant |\mathrm{t}_{|v}| \\
R &\leqslant \tfrac{d_v\epsilon|\mathrm{t}_{|v}|}{d_v+1}
\end{aligned}
$$

This ends the induction. And this also ends the proof of the lemma as we apply the above result to the root. □

Lemma 6.8 shows that trees ϵ-farness is witnessed by the existence of bad nodes. However, we need to be able to efficiently find those bad nodes in order to efficiently test membership of DTDs. In the overall size of subtrees rooted by by node is important with respect to the size of a far tree.

Lemma 6.9. *For* DTD D*, precision* $\epsilon > 0$*, and* t *a tree* ϵ*-far from* D*. Let* B_t *be the set of bad nodes whose ancestors aren't* $(\frac{\epsilon}{2}, D)$*-bad. we have* $|B_t| > \frac{\epsilon}{2}|t|$.

Proof. The proof is done by contrapositive. If $|B_t| \leqslant \frac{\epsilon}{2}|t|$, then first delete from t every subtree rooted by a node in B_t. All nodes v of the resulting tree t' are $(\frac{\epsilon}{2}, D)$-good. Hence by the previous lemma: $d(t', D) < \frac{\epsilon}{2}|t|$. Using the triangular inequality we conclude:

$$d(t, D) \leqslant d(t, t') + d(t', D) \leqslant \frac{\epsilon}{2}|t| + \frac{\epsilon}{2}|t| = \epsilon|t|$$

□

Note that D-valid trees do not contain bad nodes. We describe now how bad nodes are found with high probability for ϵ-far trees, using random objects of trees. As a consequence we obtain the membership tester in Figure 6.4. For a tree t precision far from a DTD D, note that from the last lemma, whenever we uniformly select a node of t using the query $\mathcal{R}(desc, \mathcal{R}(root))$ of some random object rdm_{t}, then with probability at least $\frac{\epsilon}{2}$, we obtain a node v such that: v is $(\frac{\epsilon}{2}, D)$-bad or v has a $(\frac{\epsilon}{2}, D)$-bad ancestor. Thus we simply move up iteratively to the ancestors of v and test their weighted words with precision $\epsilon' = \frac{\epsilon}{m_D \cdot (d^v+1)^2}$. As for all ancestor v' of v, $d^v > d^{v'}$, and according to Lemma 6.5, note that using the precision ϵ' we detect the bad ancestor with probability at least $\frac{5}{6}$. Therefore the overall probability of detecting ϵ-farness of t with this uniform drawing of nodes is at least $\frac{5\epsilon}{12}$. This shows the correctness of the tester of Figure 6.4. Now the query complexities of the tester of the tester *validate* is obtained by noticing that for each selected node v, we test at most $d^v = d(v)$, weighted words simulation the weighted words random objects by the tree random object. Thus the over all query and time complexities are bounded by $O(d^3\ |\Sigma|\ p(\mathbf{a}, d^2/\epsilon, m_D))$ and $O(\mathbf{a}\ d^3\ |\Sigma|\ p(\mathbf{a}, d^2/\epsilon, m_D) + |D|)$ respectively, as stated in Theorem 6.2. Where $D = (\Sigma, init, (A_a)_{a\in\Sigma})$, $d = d(t)$ and $\mathbf{a} = \max_{a\in\Sigma} |A_a|$.

Note also that, in the algorithm *validate* of Figure 6.4, we use the size query of the input random object to obtain the depth of the drawn node. However, as we query all ancestor of the uniformly drawn node, on do not need to use size queries for obtaining this depth and thus we can easily obtain a valid tester for DTDs approximate membership with oblivious random objects of trees.

Another remark is that the query complexity of the algorithm is obtained considering the worst case when the algorithm selects only nodes of maximal depth. However, this happens with high probability only when the tree has

```
fun validate(rdm_t, ε, D)
  // rdm_t is the (σ_det, σ_size, σ_rand, D, ⊥)−random object of some tree t
  // and ε an error precision
 let B ∪ S ∪ {R} = rdm_t the set of queries of the random object.
 let (Σ, init, (A_a)_{a∈Σ}) = D // match DTD
 for i = 1 to ⌈4/ε⌉ do
  let d = S(anc, v) //the depth of v
  let ε' = ε / (4m_D·(d+1)^2)
    fun pathTest(v)
      let a^v be the label of v // obtained using queries B(lab_a, v), a ∈ Σ.
      let r = rdm_{ω_v} //simulation of a random of ω_v using rdm_t
      //ω_v is the weighted word at v ∈ nod_t
      if member(r, ε', A_{a^v}) = NO then
        return NO and exit
      else if R(parent, v) ≠ ⊥ do
        return pathTest(R(parent, v))
      else if (a^v ≠ init) do //v is the root
        return  NO and exit
      else
        return YES
    let v = R(desc, R(root)) //generate a descendent of the root
    if (pathTest(v) = NO)
      return NO and exit
 return YES
```

Figure 6.4: A DTD validity tester for trees modulo the strong edit distance.

important number of leaves whose depth is exactly the depth of the tree. So in such case the algorithm is indeed sublinear. Thus, as future work, we may study the expected query complexity in more details to see how our algorithm performs.

6.7 Depth dependence

We will show in this section that the query complexity of any approximate DTD validity tester must depend at least linearly on the depth of the input tree. First of all we recall that a random algorithm may, at each execution step, base its decisions on the part of the input discovered so far but also on the result of some coin toss. Another fact is that algorithms we consider here input random data object of trees. Moreover a 'deterministic' algorithm inputs a random data object but is not allowed to use any coin for making its decisions. Therefore a deterministic algorithm may be seen as a decision tree with leaves labelled by its answer. And nodes of the decision tree corresponds to decisions made based only on the local structure discovered so far. The query complexity of such algorithm is then clearly the depth of its corresponding decision tree. And its error probability when receiving at random an input accounts both for the distribution of the random variable from which the input is drawn and for any randomization whatsoever in the input. Thus for random objects of trees, both distributions will be considered in estimating the probability error of our deterministic algorithms. We then use Yao's min-max theorem [Yao, 1977], which tell us that the query complexity

of any random algorithm cannot be smaller than the query complexity of the best deterministic algorithm on any randomly chosen inputs (here random object of some tree).

Theorem 6.10. *Testing approximate membership requires a query complexity linear on the tree depth of the input random object.*

Proof. As usual with proofs using the Yao's min-max principle, we give an example of DTD D, with two sets $\mathcal{P} \subseteq L(D)$ and $\mathcal{N}$ of trees of the same depth, such that all trees of $\mathcal{N}$ are ϵ-far from $L(D)$. We then provide a distribution on $\mathcal{P} \cup \mathcal{N}$ for which any deterministic algorithm with query complexity $o(d(t))$ fails with high probability (say $\frac{1}{2} - o(1)$) to distinguish elements of $\mathcal{P}$ from those of $\mathcal{N}$.

The DTD D is the following:

$$a \to a \mid b \mid \varepsilon, \quad b \to aa$$

Let $n = 5j$, $j > 3$, $\mathcal{P} = \{t_i \mid 0 < i < j - 3\}$ and $\mathcal{N} = \{t'_i \mid 0 < i < j - 3\}$, where t_i and t'_i are as in Figure 6.5. D-valid trees must have a a-node between every consecutive b-nodes, this node is missing in t'_i. Note that trees in both sets have height $3j$. It is straightforward that $\mathcal{P} \subseteq L(D)$ and all trees in $\mathcal{N}$ are $\frac{1}{5}$-far from D, using the strong edit distance.

Clearly any deterministic algorithm that distinguishes t_i from t'_i must read nodes $b \xrightarrow{fc} a \xrightarrow{fc} b$ in t_i or either $b \xrightarrow{fc} b$ or $a \xrightarrow{ns} b$ in t'_i. So it must find a b-node.

We recall that from a random object rdm_t of some tree t, we only can: access the root of t, deterministically test if some relations hold between elements of the underlying structure representing t, query the depth of the tree, uniformly generate a descendent of some node, or access the firstchild, nextsibling and parent of nodes. Therefore, note that we can not access an element in the path of two nodes without reading the whole path. This forbids any kind of dichotomy from a node to a descendent in order to find a b-node which is indispensable for distinguishing elements of $\mathcal{P}$ from those of $\mathcal{N}$.

So estimating the probability of any deterministic algorithm to find a b-node is enough to have our lower bound. Thus a fooling distribution is obtained by choosing with probability $\frac{1}{2}$ either an element of $\mathcal{P}$ or of $\mathcal{N}$. Elements in $\mathcal{P}$ and $\mathcal{N}$ are uniformly chosen. Under these settings it follows that any deterministic algorithm of query complexity C with input either rdm_{t_i} or $rdm_{t'_i}$ will find a b-node with probability less than $\frac{C}{j}$. Indeed it is not difficult to see that to guaranty a high probability of finding a b-node should use the random drawing of nodes as much as possible and each random generation of nodes fails with probability at least $\frac{1}{j}$.

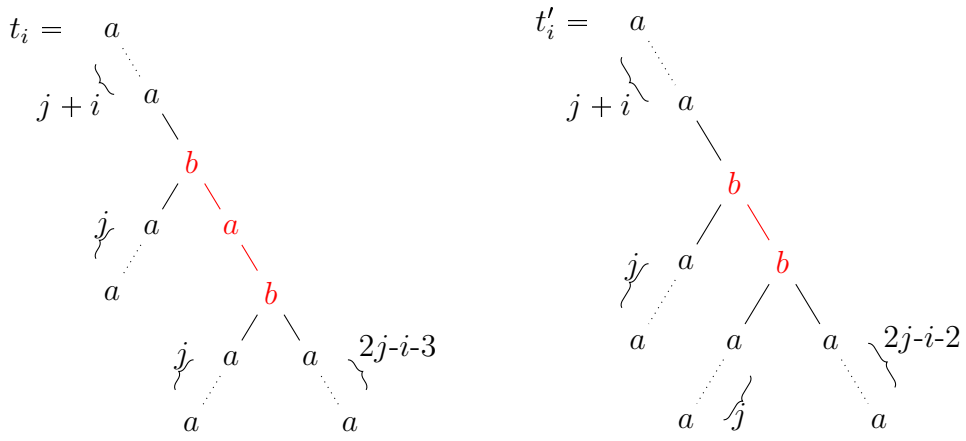


Figure 6.5: Positive and negative instances.

Then any deterministic algorithm also errs on our fooling distribution with probability at least $\frac{1}{2} \cdot (1 - \frac{C}{j})$. For $C = o(d(t_i)) = o(j)$ it follows that the error is at least $1/2 - o(1)$. □

Note that the set of trees used in our lower bound example also have fixed size linear in their height. So there is no hope to obtain membership testers sublinear for all trees. Therefore the fact that our algorithm is not sublinear in some cases (i.e. when it inputs a random object of some tree which height is almost its size) is in fact a limitation inherent to the strong edit distance.

The drawback of our lower bound is that it applies only to the strong edit distance, and not to the usual edit distance. However, we think we can obtain a lower bound for the usual edit distance, but in this case we construct trees with height logarithmic to their size (not developed so far). Such lower bound must be obtained by constructing trees ϵ-far for the usual edit distance and ϵ-close to a valid tree when the edit distance with moves is considered; and the move operations are difficult to see through random objects. However, while this would mean that no membership tester with constant query complexity exists for DTDs modulo the usual edit distance, it leaves open the question of finding a sublinear (at least logarithmic) tester for the usual edit distance. But as previously noticed neither the usual edit distance or the one with moves is appropriate for accounting trees structural errors in many applications (see introduction). Thus the lower bound here must be understood as a trade off we must make between good approximations and efficiency.

6.8 Conclusion and future work

We have presented the first approximate membership tester for DTDs modulo the strong tree edit distance. The most difficult part was to extend previ-

ous results for regular words languages to regular tree languages that are restricted to locality in vertical direction (but not horizontally). Some questions remain open. First of all, it might be possible that approximate membership modulo the edit distance can be tested efficiently for Xml Schemas by extending the methods presented here. In such a setting one would preserve top-down determinism but gives up vertical locality. A second more difficult question is whether approximate membership can be tested efficiently for bottom-up tree automata for ranked trees, while depending only on their depth. The third yet more difficult question is whether efficient algorithms exist for testing RelaxNG validity. Fourth, it might be interesting to study property testing for schemas with key constraints. Finally some recent works on property testing of distributions [Valiant, 2011; Canonne, Ron, and Servedio, 2012; Levi, Ron, and Rubinfeld, 2013], specially the introduction of new models by Canonne, Ron and Servedio using conditional samples, have similarities with our random data objects. Indeed a random data object might be seen as a collection of distributions on nodes relations. So we find it interesting to study the connection between their model and ours as future work.

7 Conclusion

Contents

7.1 Main results

In this manuscript we have studied property testing of word and tree relational structures. More specifically, we studied approximate membership of words in regular languages denoted by NFAs, and trees approximate DTD validity. The importance of property testing in the context of XML databases have been enlighten as a way of approximately and randomly checking property of databases. We have also discussed the way such properties are denoted using schema languages such as DTDs, XML schemas, Relax NG.

Our framework We introduced a new framework under which we study property testing of relational structures. The novelty of our framework is that it introduces a notion of random objects to specify the randomized accesses to relational structures. Hence testers for the approximate membership of relational structures input such random objects and we can therefore formally study the kind of accesses required for some specific property testing task. It is important for practical reasons that the random objects be efficiently implementable. Therefore we explained how we can implement random objects used in this thesis, with usual implementations of words as arrays and trees in databases.

Approximate membership for NFAs We studied approximate membership of word structures in regular languages denoted by non-deterministic finite automata. We considered the edit distance and the Hamming distance between words. For both distances, and some fixed NFA with some precision parameter ϵ, we showed that selecting uniformly some positions of the input word and running the NFA on the selected positions is sufficient to detect, with high probability, approximate membership of the word in the language denoted by the NFA. Thus we introduced the notions of blocking and infeasible fragments as witness of ϵ-farness of words from NFAs. Using the

aforementioned notions we provided testers for approximate membership of words in NFAs. As previous testers for the same problem, our tester is of constant query and time complexities, moreover its complexity improves all previous testers as its query and time complexities are polynomial in the size of the NFA, where as the previous tester of [Alon et al., 2000b] is exponential in the size of the NFA.

Approximate DTD validity We explained why property testing under the edit distance with move fails to detect some errors due mainly to order. Indeed the move edit operation mainly means that errors due to the order of appearance of tags in XML documents is not significance. However such errors may be important for practical concerns. Thus we have initiated property testing of trees modulo the strong edit distance [Selkow, 1977]. The strong edit distance is more restrictive than the standard edit distance between trees thus all positive results also apply to the standard edit distance. We then provided a tester for approximate DTD validity whose query and time complexities are polynomial in the depth of trees. And as a negative result we proved a linear lower bound in the depth of trees. We then proved that constant query complexity is impossible for DTD validity modulo the strong edit distance and thus the depth dependence is inherent to the problem. However our results provide testers with constant complexities for the class of trees with bounded depth.

7.2 Perspectives

Our further studies could be directed in mainly two directions. The first direction concerns our framework, while the second one concerns studying property testing of tree structures modulo the strong or standard edit distances; for more expressive schema languages and transformation languages.

Studying our framework Our framework is the result of our efforts to specify which kind of accesses to relational structures are necessary for a property to be testable. Thus for future work it will be interesting to characterize random objects that are required for some property to be testable. Indeed in this framework a random object can be seen as a set of queries that deterministically and randomly access the domain of relational structures, for some fixed vocabulary. Thus the question we would like to address is how to query relations of structures in order to design efficient testers of constant query complexities; and this independently of how to efficiently represent structures so that those queries are efficiently implemented.

Property testing for tree structures We have initiated property testing of tree structures and provided an efficient tester for DTD validity with query

complexity that only depends on trees depth. Thus our tester is sublinear in many cases. As future work, we want first to provide a tester that matches our linear lower bound on the depth of trees. Secondly we could take property testing of tree structure, modulo the strong edit distance, further to properties denoted by more expressible language such as XML schemas, Relax NG; so to properties denoted by tree automata. As we already know that property testing of trees modulo the strong edit distance is hard, we may consider the standard edit distance and we believe more efficient testers could exist for this weaker distance. Thus investing tree property testing for regular tree languages, under the edit distance, is an interesting open question. Thirdly, we could consider property testing under the strong or standard edit distance for XML transformations as initiated in [de Rougemont and Vieilleribière, 2007] for XSLT. Finally, as all studies of tree property testing consider only structural constraints, the next more difficult challenge is to consider schemas with key constraints.

8 Résumé

Les ordinateurs servent de nos jours à stocker et à traiter des informations diverses. Avec les ordinateurs, on communique à travers les réseaux sociaux sur le Web, on stocke d'importantes masses de données dans des bases de données ou sur le cloud, et en informatique décisionnelle ces données sont ensuite utilisées pour extraire des informations pour l'aide à la décision. Ainsi, ces dernières décennies, on assiste à une explosion de la quantité des données dans les bases de données. Donc récemment, une problématique importante en informatique est comment stocker de grosses masses de données, et comment les exploiter afin d'en extraire efficacement les informations requises. Dans ce contexte, la notion d'efficacité qui traditionnellement se traduisait en la recherche d'algorithmes linéaires en la taille des données, s'est un peu plus raffinée en laissant cours à la recherche d'algorithmes sous-linéaires en la taille des grosses masses de données [Rubinfeld and Shapira, 2011].

Algorithmes sous-linéaires

Pour obtenir des algorithmes sous-linéaires, les méthodes les plus courantes se basent sur des indexes [voir Garcia-Molina, Ullman, and Widom, 2008, chap 14] et le parallélisme. Google, par exemple, utilise des indexes pour la recherche par mots clefs [Ghemawat, Gobioff, and Leung, 2003; Dean and Ghemawat, 2004; Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, and Gruber, 2006], et IBM [IBM, 2013] comme Amazon [Amazon, 2013] ont aussi récemment développé des outils basés sur le patron d'architecture MapReduce. MapReduce permettant de faire des calculs en parallèle.

Une autre manière d'obtenir des algorithmes sous-linéaires est de considérer des algorithmes randomisés. Dans les bases de données en streaming, des algorithmes randomisées permettent d'obtenir des algorithmes efficaces, en se basant sur des statistiques comme résumés des informations pertinentes des données [Cormode and Muthukrishnan, 2007]. Il est clair que dans la plupart des cas, les algorithmes randomisés ne donnent que des réponses approchées aux problèmes. Cependant un algorithme très efficace, au prix d'une approximation aussi précise que souhaitée, est souvent préférable comparé à une autre algorithme exacte mais inefficace.

Dans cette thèse notre approche pour obtenir des algorithmes sous-linéaires est basée sur le *property testing* [Blum, Codenotti, Gemmell, and Shahoumian,

1995]. En property testing, l'objectif est de concevoir des algorithmes randomisés qui inspectent une petite partie des données afin de répondre de façon approchée à des problèmes de décision. Plus précisément, on s'intéresse aux bases de données XML et aux problèmes de validation d'un document XML par rapport à diverses schémas.

XML et le problème de validation

XML est un format de données standardisé par le W3C [Bray et al., 2008b] et XML s'est imposé comme un standard pour les échanges de données sur Web. XML est utilisé par exemple pour représenter des données dans les web services et beaucoup de bases de données XML on aussi vu le jour. Avec XML, beaucoup de langages standards existent: des langages de schéma comme les DTD et XML Schema, des langages pour définir des requêtes sur les documents XML comme XPath, et des langages de transformations comme XSLT et XQuery.

Le problème de validation pour un document XML consiste à vérifier si celui ci est conforme à un schéma. Pour étudier le problème de validation, nous modélisons un document XML par un arbre et les schémas sont modélisés par des automates d'arbres. Un example de document XML est donné ci dessous. l'arbre correspondant à ce document est donné à la Figure 8.2 et une DTD validée par le document est aussi donnée à la Figure 8.3.

```
<collection>
  <book>
    <title>Principia Mathematica</title>
    <author>Russell</author>
    <author>Whitehead</author>
    <year>1913</year>
  </book>
  <book>
    <title>M.W.M.W.W.S</title>
    <author>Cavell</author>
    <year>1969</year>
  </book>
</collection>
```

Figure 8.1: Exemple de document XML d'une librairie

Dans cette thèse on s'intéresse alors au problème de validation comme un problème d'appartenance d'un arbre dans un langage défini par un automate d'arbre. Ainsi en utilisant l'approche property testing, on s'intéresse au problème approché d'appartenance d'un arbre dans un langage défini par un automate d'arbre. On explique cette approche plus en détails dans la section suivante.

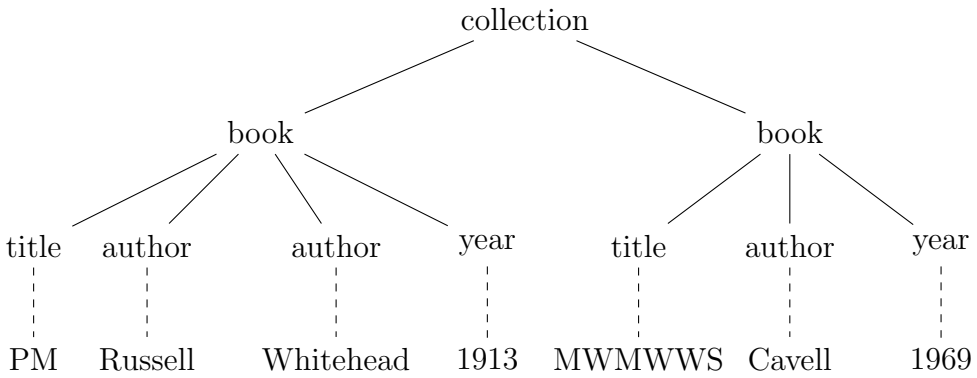


Figure 8.2: Arbre correspondant au document XML de la Figure 8.1

```
<!DOCTYPE collection[
  <!ELEMENT collection     (book*)>
  <!ELEMENT book           (title, author+, year)>
  <!ELEMENT title          (#PCDATA)>
  <!ELEMENT author         (#PCDATA)>
  <!ELEMENT year           (#PCDATA)>
]>
```

Figure 8.3: DTD validée par le document XML de la Figure 8.1

Property testing : validation approchée

Afin de définir une notion approchée d'appartenance d'un arbre t dans le langage $L(A)$ des arbres reconnus par un automate A, une distance d entre les arbres est utilisée. Pour une précision $0 < \epsilon < 1/2$, nous dirons que t est ϵ-proche d'appartenir au langage $L(A)$, ou de A, si et seulement si il existe un arbre $t' \in L(A)$ tel que $d(t, t') < \epsilon|t|$. Où $|t|$ est la taille de l'arbre t. En d'autres termes, pour la distance normalisée par la taille de t, il existe un arbre du langage dont la distance entre t est au plus de ϵ. Et dans le cas contraire nous disons alors que t est ϵ-loin de $L(A)$, ou A.

Le problème de validation approchée qui nous occupe dans cette thèse est alors le suivant : étant donné une précision ϵ, un arbre t et un automate A, décider avec grande probabilité (example 2/3) si t appartient a $L(A)$ ou si t est ϵ-loin de A. Pour les arbre non reconnu par A et ϵ-proche de A, les algorithmes randomisés (tester) solution de cette validation approchée n'ont aucune restriction sur la qualité de leurs réponses. Il est important de noter que plus la distance utilisée sépare les arbres, plus on obtient une meilleure approximation de la tâche de validation.

Contributions

Nous avons conduit nos études de la manière suivante. Premièrement nous avons considéré le cas des mots et des automates de mots comme un version simplifier du problème d'appartenance approchée des arbres. Pour le cas des mots nous avons considéré la distance d'édition et la distance de Hamming pour étudier l'appartenance approchée d'un mot à un langage régulier défini par un automate non-déterministe.

Ensuite nous nous sommes intéressés plus précisément au cas des arbres avec la distance d'édition standard ("standard edit distance") et pour une distance d'édition plus forte restreignant les insertions et deletions simplement aux feuilles des arbres ("strong edit distance").

Appartenance approchée à un langage régulier de mots

Afin de résoudre l'appartenance approchée d'un mot à un langage régulier de mots définit par un automate non-déterministe, nous avons introduit la notion de fragment bloquant pour un automate. Un fragment est défini comme un ensemble de positions du mot, et le fragment est dit bloquant lorsqu'il n'existe aucun chemin de l'automate correspondant à ce fragment. C'est à dire aucun calcul de l'automate ne peut être considéré pour ce fragment. Ensuite on prouve que pour la distance d'édition aussi bien que pour la distance de Hamming, les mots qui sont loin de l'automate contiennent beaucoup de fragments bloquants. Alors que pour les mots reconnus par un automate aucun fragment n'est bloquant pour cet automate. Ainsi nous obtenons un algorithme dont la complexité en temps est polynomiale en la taille de l'automate considéré, et en la précision. Nous améliorions ainsi le précédent algorithme [Alon et al., 2000b] pour la validation approchée des mots avec la distance de Hamming.

Appartenance approchée à un langage régulier d'arbre

Pour la distance d'édition, nous avons étudié le cas général d'appartenance d'un arbre à un langage régulier d'arbres définit par un automate. Nous avons réduit ce problème à celui des mots, mais cependant l'algorithme obtenu est exponentielle en la profondeur de l'arbre et aussi exponentielle en la taille de l'automate. Cette réduction ne marchant pas pour la "strong edit distance", laquelle distance est plus intéressante pour l'approximation de la validation, et notre algorithme ayant une complexité en temps exponentielle; pour la "strong edit distance" nous avons alors considéré le cas plus restrictive de la validation par rapport à une DTD.

Validation approchée pour une DTD Pour la "strong edit distance", la validation approchée d'un arbre par rapport à une DTD est faite par un

algorithme polynomial en la profondeur de l'arbre. Nous obtenons ainsi de meilleures complexités en utilisant les propriétés de localité des DTD. Nous avons aussi prouvé qu'il était impossible d'avoir un algorithme sous-linéaire en la profondeur de l'arbre.

Perspectives

Nous souhaiterions étendre les travaux présentés dans cette thèse principalement dans trois directions. Premièrement il serait intéressant d'étudier l'appartenance approchée des arbres à un langage régulier d'arbre pour la "strong edit distance". Ensuite on aimerait trouver des algorithmes optimaux en termes de complexité, car en effet notre borne inférieur pour la validation approchée des DTD ne correspond pas actuellement à la complexité de notre tester. En dernier nous aimerions étendre nos résultats aux langages de transformations (XSLT et XQuery); comme c'est déjà le cas avec la distance d'édition avec déplacements [de Rougemont and Vieilleribière, 2007] pour XSLT. La distance d'édition avec déplacements sépare moins les arbres et donc correspond à une approximation plus lâche de la tâche de validation. Dans nos études on a cependant considéré que les aspects structurels des documents XML, et donc un challenge plus ardu serait de considérer les contraintes de clefs des schémas XML.

Notation	Description	Def. on page

Words

w : `word`, i : `natural`, a : `letter`

$\|w\|$	size of the word w	15
$w[i]$	i^{th} letter in w	15
$pos(w)$	set of positions of w	15
$dom(w)$	domain of w	15
I	interval of w	15
F	fragment of w	15
d_h	Hamming distance between words	30
d_l	Levenshtein distance between words	30
d_m	Edit distance distance with move	30
lab_a	the label predicate for a letter a	41
σ_{words}	signature for word structures	41
S_w	relational σ_{words}-structures representating w	41
rdm_w	random object to some structure isomorphic to S_w	75

Trees

a : `letter`, t : `tree`

$\mathcal{T}_\Sigma$	set of all trees over Σ	18
$\|\mathrm{t}\|$	size of t	18
nod_t	set of nodes	18
$child_t$	child predicate	18
$root_t$ *or* ε	root of t	18
fc_t	first child predicate	18
ns_t	next sibling predicate	18
ps_t	previous sibling predicate	18
anc_t	ancestor predicate	18
$desc_t$	ancestor predicate	18
$parent_t$	parent predicate	18
$<_t$	preorder relation	18
fcns(t)	first child next sibling encoding	21
lab_a	the label predicate for a letter a	41
σ_{trees}	signature for tree structures	41
d_{strong}	Strong distance between trees	33
d_{stand}	Standard edit distance	33
d_{move}	Edit distance distance with move	33
S_t	relational σ_{trees}-structures representating w	41

Notation	Description	Def. on page
rdm_{t}	random object to some structure isomorphic to S_{t}	75

Random objects

σ : a `vocabulary`, Γ : a `relational` σ-`structure`

σ_{rand}	relations for which a random generation of elements is allowed	70
σ_{det}	relations for which boolean queries are allowed	70
σ_{size}	relations for which size queries are allowed	70
$\mathcal{B}$	boolean queries for random objects	70
$\mathcal{R}$	random functions to generate elements of the domain of Γ	70
$\mathcal{S}$	size queries for random objects	70
rdm_{Γ}	random object for Γ	70

Weighted words

$\boldsymbol{\omega}$: a `weighted word`, A : a `NFA`

$\lvert\boldsymbol{\omega}\rvert$	size of $\boldsymbol{\omega}$	120
$\lvert\boldsymbol{\omega}\rvert_*$	weight of $\boldsymbol{\omega}$	120
$\mathcal{L}_*(A)$	set of weighted words denoted by A	120
d_*	edit distance for weighted words	121

List of Figures

Bibliography

Stanford encyclopedia of philosopy. http://plato.stanford.edu/entries/properties/, 2011. (Cited page 44)

Rudolf Ahlswede, Ning Cai, Shuo-Yen R. Li, and Raymond W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4): 1204–1216, 2000. (Cited page 51)

Tatsuya Akutsu. A relation between edit distance for ordered trees and edit distance for Euler strings. *Inf. Process. Lett.*, 100(3):105–109, 2006. (Cited pages 9, 61, 110, 116, and 118)

Noga Alon and Asaf Shapira. Testing satisfiability. In *SODA*, pages 645–654, 2002. doi: 10.1145/545381.545467. URL http://dx.doi.org/10.1145/545381.545467. (Cited pages 47 and 82)

Noga Alon and Asaf Shapira. Every monotone graph property is testable. *SIAM J. Comput.*, 38(2):505–522, 2008. (Cited page 47)

Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. (Cited page 67)

Noga Alon, Eldar Fischer, Michael Krivelevich, and Mario Szegedy. Efficient testing of large graphs. *Combinatorica*, 20(4):451–476, 2000a. (Cited page 53)

Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular Languages are Testable with a Constant Number of Queries. *SIAM J. Comput.*, 30(6):1842–1862, 2000b. (Cited pages iii, 6, 9, 60, 82, 83, 84, 89, 101, 102, 103, 108, 110, 119, 126, 142, and 148)

Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.*, 64(3): 719–747, 2002. (Cited page 67)

Noga Alon, Seannie Dar, Michal Parnas, and Dana Ron. Testing of Clustering. *SIAM J. Discrete Math.*, 16(3):393–417, 2003. (Cited page 51)

Noga Alon, Tali Kaufman, Michael Krivelevich, Simon Litsyn, and Dana Ron. Testing Reed-Muller codes. *IEEE Transactions on Information Theory*, 51(11):4032–4039, 2005. (Cited page 51)

Noga Alon, Eldar Fischer, Ilan Newman, and Asaf Shapira. A Combinatorial Characterization of the Testable Graph Properties: It's All About Regularity. *SIAM J. Comput.*, 39(1):143–167, 2009. (Cited pages 5, 47, 53, and 54)

Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009. URL http://doi.acm.org/10.1145/1516512.1516518. (Cited pages 108 and 118)

Amazon. Big data on aws. https://aws.amazon.com/big-data/, 2013. (Cited pages 1 and 145)

Michael R. Anderberg. *Cluster analysis for applications*. Academic Press, 1973. (Cited page 51)

Alexandr Andoni and Robert Krauthgamer. The Computational Hardness of Estimating Edit Distance [Extended Abstract]. In *FOCS*, pages 724–734, 2007. (Cited page 37)

Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured Materialized Views for XML Queries. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Y. Chan, Venkatesh Ganti, Carl C. Kanne, Wolfgang Klas, Erich J. Neuhold, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Y. Chan, Venkatesh Ganti, Carl C. Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 87–98. ACM, 2007. ISBN 978-1-59593-649-3. URL http://dblp.uni-trier.de/rec/bibtex/conf/vldb/ArionBMP07. (Cited page 108)

Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. These, UC Berkeley, 1994. URL http://www.cs.princeton.edu/~arora/pubs/thesis.pdf. (Cited page 51)

Tim Austin and Terence Tao. On the testability and repair of hereditary hypergraph properties, May 2009. URL http://arxiv.org/abs/0801.2179. (Cited pages 53 and 54)

Maria-Florina Balcan, Eric Blais, Avrim Blum, and Liu Yang. Active Property Testing. In *FOCS*, pages 21–30, 2012. (Cited page 47)

Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002. (Cited page 67)

Tugkan Batu, Lance Fortnow, Ronitt Rubinfeld, Warren D. Smith, and Patrick White. Testing closeness of discrete distributions. *J. ACM*, 60 (1):4, 2013. (Cited page 5)

Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Transactions on Database Systems*, 34(4), 2009. URL http://doi.acm.org/10.1145/1620585.1620592. (Cited page 62)

Michael Benedikt, Leonid Libkin, and Frank Neven. Logical definability and query languages over ranked and unranked trees. *ACM Transactions on Computational Logics*, 8(2):1–62, April 2007. (Cited page 62)

Michael Benedikt, Gabriele Puppis, and Cristian Riveros. The Cost of Traveling between Languages. In Luca Aceto, Monika Henzinger, Jiri Sgall, Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 234–245. Springer, 2011. ISBN 978-3-642-22011-1. doi: 10.1007/978-3-642-22012-8_18. URL http://dx.doi.org/10.1007/978-3-642-22012-8_18. (Cited page 105)

Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XQuery 1.0 and XPath 2.0 data model (XDM). W3C Recommendation, 2010. URL http://www.w3.org/TR/xpath-datamodel/. (Cited page 2)

Marc Bernard, Laurent Boyer, Amaury Habrard, and Marc Sebban. Learning probabilistic models of tree edit distance. *Pattern Recognition*, 41(8):2611–2629, 2008. (Cited page 109)

Geert J. Bex, Frank Neven, and Jan Van den Bussche. DTDs textitversus XML Schema: A practical study. In *7-th International Workshop on the Web and Databases*, 2004. doi: 10.1145/1017074.1017095. URL http://dx.doi.org/10.1145/1017074.1017095. (Cited pages 63 and 64)

Arnab Bhattacharyya, Swastik Kopparty, Grant Schoenebeck, Madhu Sudan, and David Zuckerman. Optimal Testing of Reed-Muller Codes. In *FOCS*, pages 488–497, 2010. (Cited page 51)

Lakshminath Bhuvanagiri, Sumit Ganguly, Deepanjan Kesh, and Chandan Saha. Simpler algorithm for estimating frequency moments of data streams. In *SODA*, pages 708–713, 2006. (Cited page 67)

Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, June 2005. ISSN 03043975. doi: 10.1016/j.tcs.2004.12.030. URL http://dx.doi.org/10.1016/j.tcs.2004.12.030. (Cited pages 8, 30, 109, and 114)

Manuel Blum, Bruno Codenotti, Peter Gemmell, and Troy Shahoumian. Self-Correcting for Function Fields Transcendental Degree. In *ICALP*, pages 547–557, 1995. (Cited pages 5, 47, 51, and 145)

Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. URL http://www.w3.org/TR/2007/REC-xquery-20070123/. (Cited page 62)

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergea. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, November 2008a. URL http://www.w3.org/TR/REC-xml/#sec-prolog-dtd. (Cited page 2)

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergea. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, November 2008b. URL http://www.w3.org/TR/2008/REC-xml-20081126/. (Cited pages 2, 61, 63, and 146)

Anne Brüggemann-Klein. Regular Expressions to Finite Automata. *Theoretical Computer Science*, 120(2):197–213, November 1993. (Cited page 113)

J. R. Büchi. On a Decision Method in a Restricted Second Order Arithmetic. In Press, editor, *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, pages 1–11, 1960. (Cited page 44)

Richard J. Büchi. Weak Second-Order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6 (1-6):66–92, 1960. doi: 10.1002/malq.19600060105. URL http://dx.doi.org/10.1002/malq.19600060105. (Cited page 54)

Michael J. Cafarella, Doug Downey, Stephen Soderland, and Oren Etzioni. Knowitnow: Fast, scalable information extraction from the web. In *IN PROCEEDINGS OF THE HUMAN LANGUAGE TECHNOLOGY CONFERENCE (HLT-EMNLP-05*, pages 563–570, 2005. (Cited page 2)

Clément Canonne, Dana Ron, and Rocco A. Servedio. Testing probability distributions using conditional samples. *CoRR*, abs/1211.2664, 2012. (Cited page 139)

Jérôme Champavère, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Efficient inclusion checking for deterministic tree automata and XML schemas. *Information and Computation*, 207(11):1181–1208, 2009. doi: 10.1016/j.ic.2009.03.003. URL http://dx.doi.org/10.1016/j.ic.2009.03.003. (Cited page 65)

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, pages 205–218, Seattle, WA, USA, November 2006. URL http:

//www.usenix.org/events/osdi06/tech/chang.html. (Cited pages 1 and 145)

Boris Chidlovskii. Using Regular Tree Automata as XML Schemas. In *Proceedings of IEEE Advances in Digital Libraries*, pages 89–98, 2000. (Cited pages 61 and 63)

Hana Chockler and Orna Kupferman. w-Regular languages are testable with a constant number of queries. *Theor. Comput. Sci.*, 329(1-3):71–92, 2004. (Cited pages 60, 61, and 110)

Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. URL http://www.jstor.org/stable/2269326. (Cited page 45)

James Clark. XSL transformations (XSLT) version 1.0. W3C Recommendation, November 1999. URL http://www.w3.org/TR/1999/REC-xslt-19991116. (Cited page 62)

James Clark and Makoto Murata. Relax NG specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001. (Cited page 66)

Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. October 2007a. URL http://tata.gforge.inria.fr/. (Cited pages 16 and 58)

Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available online since 1997: http://tata.gforge.inria.fr, October 2007b. (Cited page 63)

Graham Cormode and Minos N. Garofalakis. Sketching probabilistic data streams. In *SIGMOD Conference*, pages 281–292, 2007. (Cited page 67)

Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1), February 2007. doi: 10.1145/1186810.1186812. URL http://doi.acm.org/10.1145/1186810.1186812. (Cited pages 2, 30, and 145)

Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*, pages 263–272, 2006. (Cited page 67)

Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches.

Found. Trends databases, 4(1–3):1–294, January 2012. ISSN 1931-7883. doi: 10.1561/1900000004. URL http://dx.doi.org/10.1561/1900000004. (Cited page 67)

B. Courcelle and M. Mosbah. Monadic second-Order Evaluations on Tree-Decomposable Graphs. Technical Report 90–110, LABRI, Université de Bordeaux I, November 1990. (Cited page 46)

John N. Crossley. What is mathematical logic? a survey. In *Proof, Computation and Agency*, pages 3–17. 2011. (Cited page 42)

Artur Czumaj and Christian Sohler. Sublinear-time Algorithms. In *Property Testing*, pages 41–64, 2010. (Cited pages 47 and 108)

Michel de Rougemont and Adrien Vieilleribière. Approximate Data Exchange. In *Database Theory - ICDT 2007, 11th International Conference*, volume 4353 of *Lecture Notes in Computer Science*, pages 44–58. Springer Verlag, 2007. doi: 10.1007/11965893_4. URL http://dx.doi.org/10.1007/11965893_4. (Cited pages 2, 105, 108, 143, and 149)

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. (Cited pages 1 and 145)

Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. Early Nested Word Automata for XPath Query Answering on XML Streams. Technical report, INRIA Lille, March 2013. URL http://hal.inria.fr/hal-00676178. (Cited page 62)

F.M. Dekking, C. Kraaikamp, H.P. Lopuhaä, and L.E. Meester. *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer Texts in Statistics. Springer, 2010. ISBN 9781849969529. URL http://books.google.fr/books?id=8suJcgAACAAJ. (Cited page 34)

Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012. ISBN 978-3-642-14278-9. (Cited page 23)

J. E. Doner. Decidability of the Weak Second-Order Theory of Two Successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965. (Cited page 56)

J. E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Science*, 4:406–451, 1970. (Cited pages 44 and 56)

Rod G. Downey, Michael R. Fellows, and Udayan Taylor. The parameterized complexity of relational database queries and an improved characterization of w [1. In *Combinatorics, Complexity, and Logic – Proceedings of DMTCS '96*, pages 194–213. Springer-Verlag, 1996. (Cited page 45)

Calvin C. Elgot. Decision Problems of Finite Automata Design and Related Arithmetics. *Transactions of The American Mathematical Society*, 98: 21, 1961. doi: 10.2307/1993511. URL http://dx.doi.org/10.2307/1993511. (Cited page 54)

Shimon Even, Alan L. Selman, and Yacov Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Information and Control*, 61(2):159–173, 1984. (Cited page 47)

David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition. W3C Recommendation, October 2004. URL http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. (Cited pages 2, 61, 63, and 65)

Eldar Fischer. The Art of Uninformed Decisions. *Bulletin of the EATCS*, 75:97, 2001. URL http://dblp.uni-trier.de. (Cited pages 47, 51, 82, and 108)

Eldar Fischer. Testing graphs for colorability properties. *Random Struct. Algorithms*, 26(3):289–309, 2005. (Cited page 53)

Eldar Fischer, Frederic Magniez, and Michel de Rougemont. Approximate Satisfiability and Equivalence. In *21th IEEE Symposium on Logic in Computer Science*, pages 421–430, 2006. URL http://doi.ieeecomputersociety.org/10.1109/LICS.2006.12. (Cited pages 6, 60, 82, 83, 84, 105, 108, 110, and 119)

Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal Packing and Covering in the Plane are NP-Complete. *Inf. Process. Lett.*, 12(3):133–137, 1981. URL http://dblp.uni-trier.de/db/journals/ipl/ipl12.html#FowlerPT81. (Cited page 51)

Ronald Fraïsé. Sur quelques classification des systemes de relations. *Université d'Alger, Publications Scientifiques, Série A*, 1:35–182, 1984. (Cited page 43)

Katalin Friedl and Madhu Sudan. Some Improvements to Total Degree Tests, 1995. (Cited page 51)

Sylvia Friese. *On Normalization and Type Checking for Tree Transducers*. PhD thesis, Technische Universität München, München, March 2011. (Cited page 62)

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254. (Cited pages 1 and 145)

Olivier Gauwin and Joachim Niehren. Streamable Fragments of Forward XPath. In Béatrice B. Markhoff, Pascal Caron, Jean M. Champarnaud, and Denis Maurel, editors, *International Conference on Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2011. ISBN 978-3-642-22255-9. doi: 10.1007/978-3-642-22256-6_2. URL http://dx.doi.org/10.1007/978-3-642-22256-6_2. (Cited page 62)

Pawel Gawrychowski. Chrobak Normal Form Revisited, with Applications. In Béatrice B. Markhoff, Pascal Caron, Jean M. Champarnaud, Denis Maurel, Béatrice B. Markhoff, Pascal Caron, Jean M. Champarnaud, and Denis Maurel, editors, *CIAA*, volume 6807 of *Lecture Notes in Computer Science*, pages 142–153. Springer, 2011. ISBN 978-3-642-22255-9. doi: 10.1007/978-3-642-22256-6_14. URL http://dx.doi.org/10.1007/978-3-642-22256-6_14. (Cited pages 83 and 92)

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003. (Cited pages 1 and 145)

Lior Gishboliner and Asaf Shapira. Deterministic vs non-deterministic graph property testing. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:59, 2013. (Cited page 54)

Oded Goldreich. Combinatorial Property Testing (a survey). In *In: Randomization Methods in Algorithm Design*, pages 45–60, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.6933. (Cited pages 47, 82, and 108)

Oded Goldreich and Dana Ron. Algorithmic aspects of property testing in the dense graphs model. *SIAM J. Comput.*, 40(2):376–445, 2011. (Cited pages 5 and 47)

Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property Testing and its Connection to Learning and Approximation. *J. ACM*, 45(4):653–750, 1998. (Cited pages 2, 5, 47, 79, and 82)

Georg Gottlob and Christoph Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In *21rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 17–28. ACM-Press, 2002. (Cited page 91)

Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and Its Applications*. Texts in Theoretical Computer Science. An EATCS Series. Springer, June 2007. ISBN 3540004289. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540004289. (Cited page 42)

Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, December 2004. ISSN 0362-5915. doi: 10.1145/1042046.1042051. URL http://dx.doi.org/10.1145/1042046.1042051. (Cited page 108)

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. (Cited page 37)

Christian Hagenah and Anca Muscholl. Computing epsilon-free nfa from regular expressions in $O(nlog^2(n))$ time. *ITA*, 34(4):257–278, 2000. doi: 10.1051/ita:2000116. URL http://dx.doi.org/10.1051/ita:2000116. (Cited page 113)

Shirley Halevy and Eyal Kushilevitz. Distribution-Free Property-Testing. *SIAM J. Comput.*, 37(4):1107–1138, November 2007. doi: 10.1137/050645804. URL http://dx.doi.org/10.1137/050645804. (Cited page 47)

R Hamming. Error Detecting and Error Correcting Codes. *Bell System Techincal Journal*, 29:147–160, 1950. (Cited page 29)

Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for xml. In *JOURNAL OF FUNCTIONAL PROGRAMMING*, pages 67–80. ACM Press, 2001. (Cited page 2)

IBM. Ibm infosphere biginsights, 2013. URL http://pic.dhe.ibm.com/infocenter/bigins/v1r1/index.jsp. (Cited pages 1 and 145)

M. A. Iwen. A deterministic sub-linear time sparse fourier algorithm via non-adaptive compressed sensing methods. In *in Proceedings of the 19th Symposium on Discrete Algorithms (SODA*, 2008. (Cited page 2)

Wim Janssen, Alexandr Korlyukov, and Jan Van den Bussche. On the tree-transformation power of XSLT. *Acta Inf.*, 43(6):371–393, January 2007. ISSN 0001-5903. doi: 10.1007/s00236-006-0026-8. URL http://dx.doi.org/10.1007/s00236-006-0026-8. (Cited page 62)

T. S. Jayram, Satyen Kale, and Erik Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, pages 346–355, 2007. (Cited page 67)

Jozef Jirásek, Galina Jirásková, and Alexander Szabari. Deterministic blow-ups of minimal nondeterministic finite automata over a fixed alphabet. In *Proceedings of the 11th International Conference on Developments in Language Theory*, DLT'07, pages 254–265, Berlin, Heidelberg, 2007. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=1770310.1770337. (Cited page 6)

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The dsd schema language. In *In Proceedings of the 3th ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP*, pages 285–319, 2000. (Cited page 3)

Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Transactions on Database Systems*, 31(4):1215–1256, 2006. (Cited page 62)

Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, 11(5):782–795, 2003. (Cited page 51)

Christian Konrad and Frédéric Magniez. Validating XML documents in the streaming model with external memory. In *ICDT*, pages 34–45, 2012. (Cited pages 23 and 108)

Kenneth Kunen. *Set theory : an introduction to independence proofs*. Studies in logic and the foundations of mathematics. Elsevier science, Amsterdam, Lausanne, New York, 1980. ISBN 0-444-86839-9. URL http://opac.inria.fr/record=b1117475. (Cited page 13)

Gad M. Landau and Uzi Vishkin. Efficient String Matching with k Mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986. (Cited page 37)

Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3):76–87, September 2000. doi: 10.1145/362084.362140. URL http://dx.doi.org/10.1145/362084.362140. (Cited page 63)

Dongwon Lee, Murali Mani, and Makoto Murata. Reasoning about xml schema languages using formal language theory. Technical report, Technical Report, IBM Almaden Research Center, RJ# 10197, Log# 95071, 2000. (Cited page 3)

Reut Levi, Dana Ron, and Ronitt Rubinfeld. Testing Properties of Collections of Distributions. *Theory of Computing*, 9:295–347, 2013. (Cited page 139)

Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1 edition, August 2004. ISBN 3540212027. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540212027. (Cited pages 42 and 43)

László Lovász and Katalin Vesztergombi. Non-deterministic graph property testing. *Combinatorics, Probability & Computing*, 22(5):749–762, 2013. (Cited page 54)

Frédéric Magniez and Michel de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007. (Cited pages 56, 60, 61, 67, 105, 110, and 116)

Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact XML Type Checking in Polynomial Time. In *International Conference on Database Technology*, volume 4353 of *Lecture Notes in Computer Science*, pages 254–268. Springer Verlag, 2007. (Cited page 62)

Wim Martens, Frank Neven, and Thomas Schwentick. Which XML schemas admit 1-pass preorder typing? In *10-th International Conference on Database Theory*, 2005. (Cited pages 64 and 66)

Wim Martens, Frank Neven, Thomas Schwentick, and Geert J. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, September 2006a. doi: 10.1145/1166074.1166076. URL http://dx.doi.org/10.1145/1166074.1166076. (Cited pages 61, 63, 64, and 66)

Wim Martens, Frank Neven, Thomas Schwentick, and Geert J. Bex. Expressiveness and complexity of XML schema. *ACM Transactions of Database Systems*, 31(3):770–813, 2006b. (Cited pages 62, 63, and 108)

William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. (Cited page 37)

Nimrod Megiddo and Eitan Zemel. An O(n log n) Randomizing Algorithm for the Weighted Euclidean 1-Center Problem. *J. Algorithms*, 7(3):358–368, 1986. (Cited page 51)

Philippe Michiels. *Optimizing XPath in the Context of an XQuery Implementation*. PhD thesis, Universiteit Antwerpen, 2007. (Cited page 62)

Tova Milo, Dan Suciu, and Victor Vianu. Type checking XML transformers. *Journal of Computer and System Science*, 1(66):66–97, 2003. (Cited page 62)

Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. 1997. (Cited page 37)

M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001. URL citeseer.ist.psu.edu/murata00taxonomy.html. (Cited page 63)

Makoto Murata. Data model for document transformation and assembly (extended abstract). In *In Proceedings of the workshop on Principles of*

Digital Document Processing, pages 140–152. Springer-Verlag, 1998. (Cited page 3)

Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, November 2005a. doi: 10.1145/1111627.1111631. URL http://dx.doi.org/10.1145/1111627.1111631. (Cited page 64)

Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005b. (Cited page 63)

S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005. (Cited page 67)

S. Muthukrishnan. Theory of data stream computing: where to go. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 317–319, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0660-7. doi: 10.1145/1989284.1989314. URL http://dx.doi.org/10.1145/1989284.1989314. (Cited page 67)

Antoine Ndione, Joachim Niehren, and Aurélien Lemay. Approximate Membership for Regular Languages modulo the Edit Distance. Technical report, INRIA Lille, February 2012. URL http://hal.inria.fr/hal-00666288. (Cited page 110)

Antoine Ndione, Aurélien Lemay, and Joachim Niehren. Approximate membership for regular languages modulo the edit distance. *Theor. Comput. Sci.*, 487:37–49, 2013. (Cited pages 11, 84, 118, 119, and 126)

Frank Neven. Automata theory for XML researchers. *SIGMOD Rec.*, 31(3): 39–46, 2002a. URL gaga. (Cited page 55)

Frank Neven. Automata, logic, and XML. In *Computer Science Logic*, Lecture Notes in Computer Science, pages 2–26. Springer Verlag, 2002b. (Cited page 55)

Ilan Newman. Testing Membership in Languages that Have Small Width Branching Programs. *SIAM J. Comput.*, 31(5):1557–1570, 2002. doi: 10.1137/s009753970038211x. URL http://dx.doi.org/10.1137/s009753970038211x. (Cited page 82)

Ilan Newman and Christian Sohler. Every property of hyperfinite graphs is testable. In Lance Fortnow, Salil P. Vadhan, Lance Fortnow, and Salil P.

Vadhan, editors, *STOC*, pages 675–684. ACM, 2011. ISBN 978-1-4503-0691-1. doi: 10.1145/1993636.1993726. URL http://dx.doi.org/10.1145/1993636.1993726. (Cited pages 47, 82, and 110)

Krzysztof Onak, Dana Ron, Michal Rosen, and Ronitt Rubinfeld. A near-optimal sublinear-time algorithm for approximating the minimum vertex cover size. In *SODA*, pages 1123–1131, 2012. (Cited page 47)

Ruhsan Onder and Zeki Bayram. XSLT Version 2.0 Is Turing-Complete: A Purely Transformation Based Proof. In Oscar H. Ibarra, Hsu C. Yen, Oscar H. Ibarra, and Hsu C. Yen, editors, *CIAA*, volume 4094 of *Lecture Notes in Computer Science*, pages 275–276. Springer, 2006. ISBN 3-540-37213-X. doi: 10.1007/11812128_26. URL http://dx.doi.org/10.1007/11812128_26. (Cited page 62)

Makoto Onizuka. Processing XPath queries with forward and downward axes over XML streams. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 27–38, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-945-9. doi: 10.1145/1739041.1739048. URL http://dx.doi.org/10.1145/1739041.1739048. (Cited page 62)

Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7. URL http://dblp.uni-trier.de/rec/bibtex/books/daglib/0072413. (Cited pages 37 and 46)

Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *19-th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000. doi: 10.1145/335168.335173. URL http://dx.doi.org/10.1145/335168.335173. (Cited page 64)

Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing parenthesis languages. In *Proceedings of the 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and 5th International Workshop on Randomization and Approximation Techniques in Computer Science: Approximation, Randomization and Combinatorial Optimization*, APPROX '01/RANDOM '01, pages 261–272, London, UK, UK, 2001. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=646977.711676. (Cited page 5)

Mateusz Pawlik and Nikolaus Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *PVLDB*, 5(4):334–345, 2011. URL http://dblp.uni-trier.de/rec/bibtex/journals/pvldb/PawlikA11. (Cited pages 109 and 114)

Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate XML Query Answers. In *SIGMOD Conference*, pages 263–274, 2004. (Cited page 109)

Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *BULLETIN of the American Mathematical Society*, 74:1025–1029, July 1968. (Cited page 45)

Prabhakar Raghavan. Information retrieval algorithms: a survey. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 11–18, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics. URL http://dl.acm.org/citation.cfm?id=314161.314170. (Cited page 51)

Dana Ron. Property Testing. In *Handbook of Randomized Computing, Vol. II*, pages 597–649. Kluwer Academic Publishers, 2000. (Cited pages 47, 51, and 80)

Dana Ron. Property Testing: A Learning Theory Perspective. *Foundations and Trends in Machine Learning*, 1(3):307–402, 2008. doi: 10.1561/2200000004. URL http://dx.doi.org/10.1561/2200000004. (Cited pages 47, 51, 82, and 110)

Dana Ron, Ronitt Rubinfeld, Muli Safra, and Omri Weinstein. Approximating the influence of monotone boolean functions in $O(\sqrt{n})$ query complexity. In *APPROX-RANDOM*, pages 664–675, 2011. (Cited pages 47 and 82)

Ronitt Rubinfeld and Asaf Shapira. Sublinear time algorithms. *SIAM J. Discrete Math.*, 25(4):1562–1588, 2011. (Cited pages 1, 47, and 145)

Ronitt Rubinfeld and Madhu Sudan. Robust Characterizations of Polynomials with Applications to Program Testing. *SIAM J. Comput.*, 25(2):252–271, 1996. (Cited pages 5, 47, 51, 82, and 108)

Peter Sanders, Sebastian Egner, and Ludo Tolhuizen. Polynomial Time Algorithms for Network Information Flow. In *in 15th ACM Symposium on Parallel Algorithms and Architectures*, pages 286–294, 2003. (Cited page 51)

Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, Shubha U. Nabar, and Jennifer Widom. Representing uncertain data: models, properties, and algorithms. *VLDB J.*, 18(5):989–1019, 2009. (Cited page 34)

Michael Schmidt, Stefanie Scherzinger, and Christoph Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *23rd IEEE International Conference on Data Engineering*, pages 236–245, 2007. (Cited page 62)

Georg Schnitger. Regular expressions and nfas without *epsilon*-transitions. In Bruno Durand and Wolfgang Thomas, editors, *STACS*, volume 3884 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2006. ISBN 3-540-32301-5. doi: 10.1007/11672142_35. URL http://dx.doi.org/10.1007/11672142_35. (Cited page 113)

Thomas Schwentick. Automata for XML—a survey. *Journal of Computer and System Science*, 73(3):289–315, 2007. (Cited pages 55 and 63)

Stanley M. Selkow. The Tree-to-Tree Editing Problem. *Inf. Process. Lett.*, 6 (6):184–186, 1977. (Cited pages 8, 31, 109, 114, and 142)

Asaf Shapira. *Graph Property Testing and Related Problems*. These, Tel Aviv University, 2006. URL http://people.math.gatech.edu/~{}asafico/phd.pdf. (Cited page 51)

Dana Shapira and James A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007. (Cited pages 30 and 37)

Shirinivas and Elango. Applications of graph theory in computer science an overview. *International Journal of Engineering Science and Technology*, 2 (9):4610–4621, 2010. (Cited page 23)

Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, June 2010. ISSN 1367-4811. doi: 10.1093/bioinformatics/btq217. URL http://dx.doi.org/10.1093/bioinformatics/btq217. (Cited page 2)

Larry J. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, 1974. URL http://opac.inria.fr/record=b1000295. PHD. (Cited page 45)

T.A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Pearson international edition. Pearson Addison-Wesley, 2006. ISBN 9780321315342. URL http://books.google.fr/books?id=UMcKJwAACAAJ. (Cited page 56)

Endre Szemerédi. Regular partitions of graphs. In *Colloques Internationaux C.N.R.S 260 - Problème Combinatoire et Thèorie des Graphes, Orsay*, pages 399–401, 1976. (Cited page 54)

Endre Szemerédi. Various regularity lemmas in graphs and hypergraphs. In *CiE*, page 403, 2013. (Cited page 54)

Min Tan, Raymond W. Yeung, Siu-Ting Ho, and Ning Cai. A Unified Framework for Linear Network Coding. *IEEE Transactions on Information Theory*, 57(1):416–423, 2011. (Cited page 51)

Alfred Tarski. *Introduction to Logic*. Dover Publications, March 1995. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/048628462X. (Cited page 43)

Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 204–215, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564715. URL http://dx.doi.org/10.1145/564691.564715. (Cited page 62)

J. W. Thatcher and J. B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 820, 1965. (Cited pages 54 and 56)

J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968a. (Cited pages 54 and 56)

James W. Thatcher and Jesse B. Wright. Generalized Finite Automata Theory With an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory*, 2(1):57–81, March 1968b. doi: 10.1007/BF01691346. URL http://dx.doi.org/10.1007/BF01691346. (Cited page 44)

B. Trahktenbrot. Impossibility of an algorithm for the decision problem in finite classes. 1963. (Cited page 45)

Thanh T. L. Tran, Liping Peng, Boduo Li, Yanlei Diao, and Anna Liu. Pods: a new model and processing algorithms for uncertain data streams. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD Conference*, pages 159–170. ACM, 2010. ISBN 978-1-4503-0032-2. URL http://dblp.uni-trier.de/db/conf/sigmod/sigmod2010.html#TranPLDL10. (Cited page 34)

Alan M. Turing. Computability and lambda-definability. *J. Symb. Log.*, 2 (4):153–163, 1937. (Cited page 45)

Paul Valiant. Testing Symmetric Properties of Distributions. *SIAM J. Comput.*, 40(6):1927–1968, 2011. (Cited pages 6 and 139)

Eric van der Vlist. *Relax˜NG*. O'Reilly & Associates, 2003. URL http://books.xmlschemata.org/relaxng/. (Cited pages 2, 61, and 63)

Stijn Vansummeren. Deciding Well-Definedness of XQuery Fragments. In *24rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2005. (Cited page 62)

Moshe Y. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–146, 1982. (Cited page 45)

Mingzhu Wei, Ming Li, Elke A. Rundensteiner, and Murali Mani. Processing Recursive XQuery over XML Streams: The Raindrop Approach. In *22nd International Conference on Data Engineering Workshops (ICDEW)*, pages 85–94, 2006. (Cited page 62)

XPath. XPath Recommendation. urlhttp://www.w3.org/TR/xpath20/, 2010. (Cited pages 61 and 62)

Andrew C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.24. URL http://dl.acm.org/citation.cfm?id=1398506.1382556. (Cited pages 38, 111, and 136)

Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6): 1245–1262, December 1989. ISSN 0097-5397. doi: 10.1137/0218082. URL http://dx.doi.org/10.1137/0218082. (Cited pages 109 and 114)