Feature Extraction from Bounded Tree-Width Graphs Using Canonical String Representations

Ivan Danielov Ivanov

Matriculation number: 2690453

August 19, 2016

Master's Thesis

Computer Science

Supervisors:

Prof. Dr. rer. nat. Stefan Wrobel

Prof. Dr. Jens Lehmann

Dr. Tamás Horváth

Institut für Informatik, Abteilung III

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Declaration of Authorship

I, Ivan Danielov Ivanov, declare that this thesis titled, "Feature Extraction from Bounded Tree-Width Graphs Using Canonical String Representations" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Acknowledgements

First of all, I would like to express my sincere gratitude to Dr. Tamás Horváth, who proposed the topic of this work and guided me closely during the thesis duration. In addition, I would like to thank him for the dedicated time and the advice he gave me.

I would also like to thank Prof. Jens Lehmann for the productive discussions we had and his valuable suggestions.

Last but not least, I would like to thank my mother, sister, and grand mother, who supported me throughout my life and studies, and I dedicate this work to my father, who left this world too early.

Contents

1 Introduction **1**
- 1.1 Related Work . . . 4
- 1.2 Thesis Structure . . . 5

2 Preliminaries **7**
- 2.1 Graph Theory . . . 7
  - 2.1.1 Basics . . . 7
  - 2.1.2 Tree-Width . . . 8
- 2.2 Weisfeiler-Lehman Isomorphism Test . . . 10
- 2.3 w-shingling . . . 12

3 Algorithm **15**
- 3.1 Canonical String Representations of Partial 3-Trees . . . 16
  - 3.1.1 Definition . . . 16
  - 3.1.2 Example . . . 25
- 3.2 Auxiliary Methods . . . 25
  - 3.2.1 Function `weisfeiler_lehman` . . . 27
  - 3.2.2 Function `get_w_shingles` . . . 27
- 3.3 Feature Extraction . . . 27
  - 3.3.1 Definition . . . 28
  - 3.3.2 Example . . . 28
- 3.4 Theoretical Evaluation . . . 30

4 Implementation **33**
- 4.1 Arnborg-Proskurowski . . . 34
- 4.2 Weisfeiler-Lehman . . . 35
- 4.3 Data Processing . . . 36

5 Empirical Evaluation **39**
- 5.1 Prediction of Chemical Properties . . . 42
- 5.2 Type Inference of DBpedia Resources . . . 48

6 Summary **51**

List of Figures

3.1 Basic reduction rules for partial 3-trees. Figure taken from [1]. 18
3.2 Possible configurations when all vertices in I are of degree 1. Figure taken from [1]. 22
3.3 Cycles in I. Figure taken from [1]. 22
3.4 The prism pattern. Figure taken from [1]. 23
3.5 Conflicting buddy configurations. Figure taken from [1]. . . . 24
3.6 Conflicting cube configurations. Figure taken from [1]. 24
3.7 Example of computing the canonical string representation of a partial 3-tree. 26
3.8 Example of a directed graph of tree-width 2. 29
3.9 A second example of a directed graph of tree-width 2 yielding the same set of features as the first one. 32

5.1 Example of preprocessing an RDF graph describing a person authoring two books. 41
5.2 Time needed for extracting the canonical string representations of random graphs with different number of nodes. 42
5.3 Average time for extracting features from a chemical compound graph in the Carcinogenesis dataset for different number of Weisfeiler-Lehman iterations (h). 47

List of Tables

5.1 Tree-width of the graphs in the chemical datasets. 45
5.2 Number of nodes and edges of the graphs in the chemical datasets. 45
5.3 Classification results for chemical datasets in ordinary graph representations. 45
5.4 Classification results for chemical datasets in RDF format. . . 46
5.5 Ten most frequent RDF types in the sample I of DBpedia resources. 49
5.6 Number of nodes and edges of the neighborhoods of the extracted sample I of DBpedia resources. 50
5.7 Results for multi-label classification of DBpedia resources. . . 50

Abstract

This work aims to efficiently solve learning problems in large graph databases using a simple but fast feature extraction method for graphs of tree-width at most 3. The class of all such graphs is rich enough for a number of real-world application areas, such as molecular graphs and direct neighborhoods of resources in large RDF datasets. In this work, the proposed method is applied predominantly to solving learning problems on RDF data because of the increasing importance and popularity of the Semantic Web in recent years. The feature set of a graph is defined by the set of fixed-length substrings of its canonical string representation, which is computed in linear time using the algorithm of Arnborg and Proskurowski. The performance of the approach is evaluated empirically on benchmark chemical graph datasets, as well as on predicting type information of resources in DBpedia. The proposed method achieves good classification results on both ordinary graphs and RDF data. It is shown that preprocessing of the RDF data can strongly influence the performance of the method.

Keywords: feature extraction, bounded tree-width, canonical string representations

Chapter 1

Introduction

The problem of learning in graph structured data has been widely studied since *graph kernels* were introduced by [2]. Standard learning models assume that the input data can be fit into a single fixed-width table or a single row representation. However, graphs have no such natural representation. A graph kernel is a function $k(G_1, G_2)$, with G_1 and G_2 being graphs, which is equivalent to the dot product $\langle \Phi(G_1), \Phi(G_2) \rangle$ where Φ is a mapping from the class of all graphs to some Hilbert feature space. *Support Vector Machines* (SVM) [3], as well as other kernel methods [4], can solve graph learning problems using such graph kernels.

The ever-increasing amount of structured data on the World Wide Web demands the use of fast and efficient algorithms for solving machine learning and data mining problems. The commonly used representations for structured data are graphs. Defining a good general graph kernel is difficult because the high expressiveness of graphs leads to high time complexity of the graph kernels. For example, computing a complete graph kernel (i.e., one which is injective modulo isomorphism) is at least as hard as deciding graph isomorphism [5]. One way of reducing the time complexity of graph kernels is by restricting their domain.

An interesting application field of learning in graphs is the Semantic Web. *Resource Description Framework* (RDF) datasets, which build the web of Linked Data, can be viewed as graphs in which the RDF resources are vertices and the RDF triples are edges. Standard ways for data inference in RDF based on deductive reasoning do not always perform well on crowd sourced RDF datasets. The reason is that these approaches lead to the multiplication of errors in the inferred information [6]. Dealing with missing or inconsistent data is a usual problem in large RDF datasets which demands the use of statistical-based reasoning methods. More specifically, the inference of type information of RDF resources can be regarded as a classification problem

and solved with the help of graph kernels.

In this work we propose an efficient method for mapping graphs to binary feature vectors for solving machine learning and data mining problems on graph structured data. A *canonical string representation* of a graph is a string unique modulo isomorphism. Our method follows the intuitive idea that, although non-isomorphic graphs have different canonical string representations, similar graphs should share substrings of their canonical string representations. Therefore, we consider graphs which have large overlap of their canonical string representations more similar than graphs having smaller overlap. Following this idea, we apply the approach proposed by [7], which was originally used in the context of filtering near-duplicate documents on the Web, to identify similar canonical string representations.

The fast feature extraction method proposed in this work is applicable to graphs of tree-width at most 3. Given such a graph, the feature set is defined by the set of substrings of length w (also known as w-shingles [7]) extracted from the canonical string representation of the graph. The length parameter w is defined by the user. Two graphs are considered similar if they share a large proportion of their w-shingles.

We improve the quality of our mapping function by applying a method for refining the vertex labels of the input graph. This way we increase the diversity of the extracted w-shingles, which leads to the expansion of the distances between dissimilar graphs in the feature space. The label-refinement algorithm which we apply is based on the Weisfeiler-Lehman test of graph isomorphism [8].

The canonical string representations in our approach are computed in linear time for graphs of tree-width at most 3 using the algorithm of Arnborg and Proskurowski [1]. The algorithm performs a canonical sequence of reductions on the input graph using a number of safe reduction rules [9] until the graph is reduced to a single labeled vertex. The label of the remaining vertex is the canonical string representation of the graph. The input graph must have tree-width at most 3. The tree-width is restricted to 3 because there is no easy generalization of the reduction methods used by the algorithm for graphs of larger tree-width [9].

There are a number of beneficial algorithmic and application properties which apply to the class of graphs of tree-width at most 3. Isomorphism can be decided practically in linear time using canonical string representations, which, as mentioned above, are computed in linear time for such graphs. Moreover, the membership for this class is also decidable in linear time [9] and it includes interesting graph classes, such as, for example, the class of outerplanar graphs. In addition to the theoretical richness of the class, a number of real-world areas use graph structures which mostly fit into this

class. Particularly, the molecular graphs of most chemical compounds have tree-width at most 2 [10].

The proposed technique can be extended to a graph kernel using a kernel function (e.g., linear or Gaussian) defined in the usual way by means of the extracted feature sets. Most graph kernels are computed by extracting structural patterns from the graphs (e.g., walks [5], cyclic patterns [11]). The proposed method, however, extracts patterns which are not necessarily related to the structure of the graph. This makes it conceptually different from most other graph kernels.

A common goal when doing feature extraction is to maintain the proximity between similar graph structures while keeping dissimilar graphs apart. In particular, one might require two graphs to be associated with the same feature sets if and only if they are isomorphic. However, one can easily see that the feature extraction method proposed in this work is not necessarily injective modulo isomorphism. This can be considered a disadvantage. However, one can modify the strictness of the mapping (i.e., how difficult it is to find two non-isomorphic graphs which yield the same feature sets) by adjusting the parameters of the proposed method. This option increases the flexibility of the approach.

The predictive performance of the method is investigated on real-world chemical graph datasets, as well as on *DBpedia* [12]. First, the approach is applied to benchmark pharmacological datasets given in ordinary graph representations. In the second stage of our experiments we apply the method to chemical data in RDF format. The last experiments are on the task to predict the RDF types of resources in the DBpedia dataset. The results of our experiments show that the method proposed performs well on both ordinary graphs and RDF data. It is also shown that different preprocessing techniques applied to the raw RDF data influence strongly the classification performance. The results on RDF data show that the predictive power of the method is worse than that of techniques designed specifically for RDF, but the method is still useful for this domain.

In summary, we deal with the problem of applying machine learning and data mining techniques to graph structured data. In this work, we introduce a fast feature extraction method for graphs of tree-width at most 3. The method can be extended to a graph kernel. The advantages of our approach are that

- it performs in linear time,
- it is applicable to a class of graphs which is expressive enough for a number of practical applications,

- it provides the extracted feature vectors explicitly,
- it is conceptually different from most other graph kernels.

The disadvantage of the method is that it may map two non-isomorphic graphs to the same set of features. The method yields very good results on graph classification tasks and can be applied to solving learning problems in RDF data.

1.1 Related Work

Many graph kernels have been developed in recent years [13, 14]. They can be classified mostly into three types based on the patterns they are extracting [15]:

- walks and paths (e.g., [5, 13, 16, 17, 18, 19])
- limited-size subgraphs (e.g., [20, 11])
- subtrees (e.g., [21, 22, 23])

Graph kernels based on walks and paths count the number of shared random walks or paths between two graphs. Although the runtime complexity of the standard way to compute such a kernel is $O(n^6)$ [5], some optimizations have been proposed which reduce the computation time of the kernel [13]. Such kernels have shown good performance, for example, on applications in computer vision [16, 17, 18] or chemoinformatics [19].

Graph kernels based on limited-size subgraphs may consider patterns such as *graphlets* [20], which are subgraphs of small size. Another example of a graph kernel from the second type is the *Cyclic Pattern Kernel* (CPK) [11] which is computed by extracting first cyclic and tree patterns from each of the input graphs as features and counting then the number of shared features. The frequency of the patterns is ignored. On general graphs, computing the CPK is NP-hard [11]. However, it has been shown that CPK can be computed in time polynomial in the number of cyclic patterns for graphs of bounded tree-width [24]. The CPK shows good performance on classifying chemical compounds [11].

Graph kernels based on extracting subtree patterns compare all matching subtrees rooted at a particular vertex between two graphs G and G'. This is done for all pairs of vertices v from G and v' from G' where v and v' are regarded as the "roots". The original subtree kernel was proposed by [21]. The kernels proposed by [22, 23] give more specific definitions of the subtree kernel for tasks in chemoinformatics and hand-written digit recognition.

The *Weisfeiler-Lehman Graph Kernels* (WLGK) [15] are a class of graph kernels. The idea of WLGK is to extend the standard graph kernels by applying a given graph kernel iteratively, refining the labels of the graphs at each step using the Weisfeiler-Lehman graph isomorphism test [8], and summing the results of the kernel for all iterations. The Weisfeiler-Lehman subtree graph kernel ($\text{WLGK}_{\text{subtree}}$) [15], for example, computes the number of shared subtrees between two graphs using the previously defined procedure. WLGK is a state-of-the-art kernel which performs very well on graph classification problems. It is also very time efficient when applied to large graphs.

A number of approaches using graph kernels have been proposed in the context of learning in RDF data [14]. Two general RDF graph kernels, based on the intersection of graphs and the intersection of trees, have been proposed by [25]. The first method is a walks and paths based graph kernel, while the second is based on matching subtree patterns. A fast approximation of $\text{WLGK}_{\text{subtree}}$ optimized for RDF data has been proposed by [26]. In contrast to [25], the previous kernel is applied not exclusively on the subgraphs of interest but on the whole RDF graph. Compared to the original $\text{WLGK}_{\text{subtree}}$, this kernel is computed much faster and shows similar performance. In [27] have been introduced some modifications on existing RDF graph kernels and their performance and computation speed have been improved. In addition, these RDF graph kernels not only compute the pairwise similarities between graphs, but also allow the explicit computation of feature vectors. This allows arbitrary machine learning methods to be applied directly on the feature vectors for solving learning problems in RDF data.

1.2 Thesis Structure

The thesis is organized as follows. We start with relevant background knowledge in Chapter 2. In that chapter, we introduce basic notions from graph theory which are used in the rest of the work. In addition, we discuss the Weisfeiler-Lehman test for graph isomorphism and the w-shingling process. In Chapter 3, we define the feature extraction algorithm and all inner procedures which are used. These include the algorithm for computing the canonical string representations proposed by Arnborg and Proskurowski [1], the Weisfeiler-Lehman subroutine, and the w-shingling of the canonical string representations. In Chapter 4, we discuss some important implementation details. In Chapter 5, we evaluate empirically the proposed feature extraction method. In the first part of the chapter, we conduct experiments on classifying chemical compounds of benchmark pharmacological datasets. In

the second part, we test the performance of the method on predicting the RDF types of resources on DBpedia. Finally in Chapter 6, we conclude with a summary of the proposed method and ideas for future work.

Chapter 2

Preliminaries

This chapter will introduce some basic notations of graph theory (see, e.g., [28]), the Weisfeiler-Lehman isomorphism test [8], and w-shingling [7], which are necessary to understand the discussion in the rest of the chapters.

2.1 Graph Theory

2.1.1 Basics

The topic of this master thesis is related to solving problems with graph structured data. A *graph* G will be regarded as an ordered triple (V, E, λ). V is a non-empty finite set of *vertices* (or *nodes*). E is a finite set of pairs $\{u, v\}$ called *edges* (or *arcs*) where $u, v \in V$ are called *endpoints* of the edge. The mapping $\lambda : V \cup E \to \mathbb{N}$ is a labeling function. If the elements of G are unlabeled, λ can be omitted. A *directed edge* is an ordered pair of vertices (u, v), which is said to be directed from u to v. A *directed graph* is a graph whose edges are directed. If G has any two edges with the same set of endpoints (such edges are called *parallel*), it is called a *multi-graph*. Graph $K = (V_K, E_K, \lambda_K)$ is a *subgraph* of G if $V_K \subseteq V$, $E_K \subseteq E$ and $\forall x \in V_K \cup E_K : \lambda_K(x) = \lambda(x)$. K is said to be *induced* if $\forall u, v \in V_K : \{u, v\} \in E_K \iff \{u, v\} \in E$.

Two vertices are *adjacent* (or *neighbors*) if they are connected by an edge. An edge connecting two nodes is *incident* to them. The *degree* of a vertex is the number of its neighbors. Let the *edge-degree* of a vertex denote the number of edges incident to that vertex (which may be different from the degree in multi-graphs). A *path* is a sequence of distinct vertices where each pair of consecutive nodes is adjacent. A *cycle* is a path such that the first and the last nodes are the same. G is called a *complete graph* if every $u, v \in V$

are adjacent. A *k-clique* is a complete graph with k vertices. A *regular* graph is a graph in which all vertices have the same degree r which is also called *order* of the graph.

A graph is *connected* if for every $u, v \in V$ there exists a path between u and v. Otherwise, the graph is *disconnected*. A maximal connected subgraph of a graph is called a *connected component*. For a set $U \subset V$, the (minimal) set of vertices $S \subset V$, $S \cap U = \emptyset$, the removal of which disconnects U from the rest of the graph is called a (minimal) *separator* of U.

A graph $B = (U \cup V, E, \lambda)$ is *bipartite* if its vertices can be divided into two disjoint sets U and V such that no two vertices belonging to the same set are adjacent.

A *forest* will be regarded as a graph which has no cycle (for the sake of simplicity edge directions and parallel edges are ignored). A *tree* is a connected forest.

A *hypergraph* H is an ordered triple (V_H, E_H, λ_H), where V_H and E_H are disjoint sets and $\lambda_H : V_H \cup E_H \to \mathbb{N}$ is a labeling function. The elements of E_H are non-empty subsets of V_H, called *hyperedges*. An *n-hyperedge* is a hyperedge of cardinality n. A 1-hyperedge is also called a *self-loop*. Graphs can be regarded as hypergraphs having only 2-hyperedges.

Two graphs $G_1 = (V_1, E_1, \lambda_1)$ and $G_2 = (V_2, E_2, \lambda_2)$ are *isomorphic* if there is a bijection ϕ between V_1 and V_2 such that:

- $\forall u, v \in V_1 : \{u, v\} \in E_1 \iff \{\phi(u), \phi(v)\} \in E_2$
- $\forall u \in V_1 : \lambda_1(u) = \lambda_2(\phi(u))$
- $\forall \{u, v\} \in E_1 : \lambda_1(\{u, v\}) = \lambda_2(\{\phi(u), \phi(v)\})$

Informally, two graphs are isomorphic if they are the same. G_1 is *subgraph isomorphic* to G_2 if G_2 has a subgraph isomorphic to G_1. It is an open problem if deciding isomorphism between two graphs can be solved in polynomial time [29]. The problem of deciding whether a graph is subgraph isomorphic to another graph is NP-Complete [30].

The *canonical string representation* of G is a unique string assigned to the graph. Two graphs have the same canonical string representations if and only if they are isomorphic. Thus, the graph isomorphism problem can be solved efficiently using such strings if they can be efficiently computed.

2.1.2 Tree-Width

A key measurement used in this work for the complexity of a graph is its *tree-width*, which will be defined shortly.

As defined in [31], a *tree decomposition* of a graph $G = (V, E, \lambda)$ is a pair $(T, \mathscr{X})$ where $T = (I, F)$ is a tree and $\mathscr{X} = \{X_i : i \in I\}$ is a family of subsets of V with the following properties:

- $\bigcup_{i \in I} X_i = V$
- for every $\{u, v\} \in E$ there exists an $i \in I$ with $\{u, v\} \subseteq X_i$
- for all $i, j, k \in I$, if j is on the path of T between i and k then $X_i \cap X_k \subseteq X_j$

The *width* of a tree decomposition is

$$\max_{i \in I}(|X_i| - 1).$$

The *tree-width* of a graph G is the minimum width over all tree decompositions of G. Following this definition, for example, the tree-width of paths and trees is 1, and the tree-width of cycles is 2.

A *k-tree* is a generalization of the concept of tree. It can be defined recursively as follows [32]. A k-clique is a k-tree. Given any k-tree $T_k(n)$ on n vertices, a k-tree on $n + 1$ vertices can be obtained by adding a new vertex adjacent to all vertices of a k-clique of $T_k(n)$. A vertex of a k-tree with degree k is called a *k-leaf*. Any subgraph of a k-tree is called a *partial k-tree*. It is shown by [33] that the class of partial k-trees and the class of graphs of tree-width at most k are equivalent. Therefore, in the rest of this work we will use the term partial k-tree as a synonym for a graph with tree-width at most k.

A number of computationally hard problems on arbitrary graphs become efficiently solvable for graphs of bounded tree-width (e.g., graph isomorphism [34]). Deciding if a graph belongs to the class of partial k-trees for a constant k can be done in linear time, as proposed by [35]. However, this method is not practical for large k. An efficient algorithm for deciding if a graph has tree-width at most k for $k \leq 3$ was proposed by [36]. Moreover, Arnborg and Proskurowski [1] introduced an algorithm for efficiently building canonical string representations of such graphs, which is essential to the feature extraction method discussed in Chapter 3. The class of partial 3-trees is rich enough for many practical applications. For example, many graph representations of chemical compounds have tree-width at most 2 [10]. Thus, it is interesting to investigate these structures and the problems that can be solved with them. The rest of this work focuses on graphs belonging to the class of partial 3-trees.

2.2 Weisfeiler-Lehman Isomorphism Test

As mentioned earlier, it is an open problem if isomorphism between two graphs can be decided in polynomial time. However, there exists a very simple but incorrect linear time algorithm for testing if two graphs are isomorphic. The method is based on the algorithm proposed by Weisfeiler and Lehman [8] for refining the colors of the set of vertices in a graph. This test for graph isomorphism errs with very small probability on arbitrary graphs [37, 38].

The Weisfeiler-Lehman (WL) method iteratively assigns colors (labels) of the vertices in the graph. In the initial step all vertices have their original labels. At each subsequent step the color of each vertex is updated to reflect its previous color augmented with the sorted multiset of colors of its neighbors. This continues until a stable coloring is reached. The steps of the method are listed in Algorithm 2.1.

Algorithm 2.1 $\mathcal{WL}(G)$

Input:
graph $G = (V, E, \lambda)$
Output:
a multiset of refined vertex labels

1: **for all** $v \in V$ **do**
2: $l_0(v) := \lambda(v)$
3: **end for**
4: $i := 0$
5: **repeat**
6: $i := i + 1$
7: **for all** $v \in V$ **do**
8: $N_i(v) := \{l_{i-1}(u) : \text{for all neighbors } u \text{ of } v \text{ in } G\}$ // multiset
9: Sort $N_i(v)$ lexicographically
10: $s_i(v) := \texttt{string}(l_{i-1}(v); N_i(v))$
11: $l_i(v) :=$ a unique identifier of the value of $s_i(v)$
12: **end for**
13: **until** $\forall u, v \in V : l_i(u) = l_i(v) \iff l_{i-1}(u) = l_{i-1}(v)$
14: **return** $\{l_i(v) : \forall v \in V\}$ // multiset

Let two vertices be equivalent if and only if they have the same label. Initially, for every pair of vertices in the input graph, two vertices belong to the same equivalence class if and only if they are equivalent. In each iteration, the algorithm produces a refinement of the equivalence classes of

the previous iteration. In at most $|V| - 1$ iterations, all vertices must belong to a singleton equivalence class. Therefore, the former algorithm finds a stable vertex coloring in at most $|V|$ iterations.

Let $G = (V, E, \lambda)$ and $G' = (V', E', \lambda')$ be arbitrary graphs. The augmentation $s_i(v)$ for some vertex $v \in V$ in iteration i is a string of concatenated labels. This string is given a global identifier (i.e., a number or a short string) which is specific to iteration i. Global means, if $s_i(v) = s_i(v')$ for $v \in V$ and $v' \in V'$, then $s_i(v)$ and $s_i(v')$ will have the same identifiers. The identifier represents the (possibly much larger) augmented string $s_i(v)$ and is regarded as the new label of vertex v in iteration i (i.e., $l_i(v)$). This way, the produced labels in the current iteration of the algorithm are kept short, which reduces the memory usage of the algorithm.

Following this method, it is true for all pairs G and G' that if G is isomorphic to G' then $\mathcal{WL}(G) = \mathcal{WL}(G')$. This is true because two isomorphic graphs will always have the same labels in each iteration of the algorithm. However, if $\mathcal{WL}(G) = \mathcal{WL}(G')$, G and G' are not necessarily isomorphic. The WL test of isomorphism fails for regular graphs, where it only provides information about their order [37]. An important advantage of this graph isomorphism test is that it can be performed in linear time [39].

Algorithm 2.1 is also called *1-dimensional* WL method. The k-dimensional WL method is a generalization of the discussed one and will not be covered here. More information about the k-dimensional WL method can be found for example in [40].

The WL method is also used as the basis of the Weisfeiler-Lehman graph kernel [15]. This kernel is defined as

$$\mathrm{WLGK}_k^{(h)}(G, G') = \sum_{i=0}^{h} k(G_i, G'_i)$$

where G and G' are graphs, k is a graph kernel and h is a number of iterations of the WL method to be performed. In this case exactly h iterations are applied on each of the graphs without performing a stability check on the vertex colors. G_i and G'_i denote the graphs with labels produced by the WL method at iteration i.

An approach based on the WL method is used in this work to make the features extracted from graphs more specific. This will be discussed in the next chapter.

2.3 w-shingling

Deduplication of documents on the Web can improve the diversity of results produced by search engines [7]. An efficient method for identifying and filtering near-duplicate documents was proposed by [7]. The algorithm is able to solve the task for very large collections of documents by building a sketch which is of much smaller size than the actual database.

The idea is to map each document A to a set of features called *shingles*. A document is a viewed as a sequence of tokens (e.g., characters, words). The set of w-shingles of A is the set of all subsequences of A with size w tokens. As an example, let A be the document "$xyzxyw$" and tokens be characters. Then the set of 2-shingles of A will be $\{xy, yz, zx, yw\}$. In order to reduce the necessary storage space, each set of w-shingles is converted to a set of *Rabin fingerprints* [41] of the shingles.

The *Jaccard similarity* between two sets X and Y is defined as:

$$\text{sim}_{\text{Jaccard}}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}.$$

Let A and B be two documents with S_A and S_B being their respective sets of fingerprints of their w-shingles. The similarity between documents A and B is defined as the Jaccard similarity between S_A and S_B, i.e.:

$$\text{sim}(A, B) = \text{sim}_{\text{Jaccard}}(S_A, S_B).$$

Intuitively, the documents are near-duplicates, if the similarity between their sets of w-shingles is high.

Given the database of sets of fingerprints representing the original documents, the sketch M can be computed in time linear in the size of the documents in the following way. For a given pair of adjustable parameters k and L, the sketch has m columns (m - number of documents) and kL rows. Let $\pi_1, ..., \pi_{kL}$ be random permutations of $\{0, ..., n-1\} = [n]$ where n is the size of the maximum fingerprint (in practice $n = 2^{64}$). Then for each cell in M let

$$M_{ij} = \min\{\pi_i(S_j)\}$$

where S_j is the set of w-shingle fingerprints of the j-th document. The described technique is called *min-hashing* [42].

Let S_x and S_y be two sets of fingerprints and π be a random permutation of $[n]$. It has been shown by [7] that the probability of S_x and S_y yielding the same minimum element with respect to π equals the Jaccard similarity of S_x and S_y, i.e.:

$$\Pr[\min\{\pi(S_x)\} = \min\{\pi(S_y)\}] = \text{sim}_{\text{Jaccard}}(S_x, S_y).$$

The rows of the sketch are split into L bands of k rows. Two documents with sets S_x and S_y are considered near-duplicates if their sketch columns agree on at least one band.

Chapter 3

Algorithm

This chapter defines the method for extracting features from graphs of tree-width at most 3 (i.e., partial 3-trees). The former problem can be defined as follows. Let $\mathcal{G}_{\text{p3-tree}}$ be the class of partial 3-trees, $\mathcal{S} = \{0,1\}^n$ be a binary vector space (may also be referred to as *feature space*) of n dimensions (typically large n) and Φ be the mapping

$$\Phi : \mathcal{G}_{\text{p3-tree}} \mapsto \mathcal{S}.$$

Define Φ such that it ideally fulfills the following properties. Let for any $G_1, G_2 \in \mathcal{G}_{\text{p3-tree}}$ and some distance function d on $\mathcal{S}$:

1. $\Phi(G_i), i \in \{1,2\}$, can be computed in time polynomial in the size of the input graph,
2. G_1 is isomorphic to G_2 if and only if $\Phi(G_1) = \Phi(G_2)$,
3. when G_1 and G_2 are "similar" $d(\Phi(G_1), \Phi(G_2))$ is small, and it is larger when they are not.

These conditions are desired because they maximize the usefulness of the mapping for solving machine learning and data mining problems on graphs. Condition 1 makes sure that if we decide to use Φ for solving learning problems on graphs, the mapping will be computed fast, which is usually desired in practical situations. Condition 2 may be desirable for problems in which it is important that the mapping separates non-isomorphic graphs. Condition 3 is very important for learning problems because it makes sure that the mapping preserves the locality of the data. When condition 3 is fulfilled, two similar graphs will be mapped to vectors which lie close in the feature space. Therefore, if the properties of a graph G apply with high probability also to other similar graphs, we can assume that these properties apply with high

probability to the graphs which are mapped close to G in feature space. We will show in Section 3.4 that the proposed feature extraction method fulfills condition 1 and partially condition 2. The last condition will only be measured empirically in the context of the proposed feature extraction method because graph similarity is ill-defined.

The proposed method for mapping a graph to a set of features (i.e., w-shingles of the canonical string representation of the graph) covers partially the ideal properties of Φ. The w-shingles are a set of strings extracted from the canonical string representation of the input graph using a sliding window of size w.

This chapter is organized in the following way. Section 3.1 introduces the way to extract canonical string representations from partial 3-trees. After defining some additional functions in Section 3.2, the outer loop of the algorithm is presented in Section 3.3. The quality of the feature extraction is investigated in Section 3.4, with respect to the three properties defined at the beginning of this chapter.

3.1 Canonical String Representations of Partial 3-Trees

This section explains the way to construct canonical string representations of partial 2- and 3-trees proposed by Arnborg and Proskurowski [1]. The algorithm computes the canonical string representation of a graph with the help of safe reduction rules in log-linear time or linear time and quadratic space.

3.1.1 Definition

The method is based on a canonical reduction sequence using a number of safe reduction rules [9]. *Safe* means that the graph obtained after performing the reduction does not have larger tree-width than the original graph. The general schema of the algorithm for extracting a canonical string representation is listed in Algorithm 3.1. The function performing the algorithm is denoted as `get_canon_repr`(G). The input graph G is initially transformed to its hypergraph equivalent H_0. This step is necessary because some reduction rules result in the creation of 3-hyperedges. A hypergraph produced in any iteration may only have 1-, 2-, and 3-hyperedges. The functions `get_reducible_features`(H_i) and `reduce`(H_i, R_i) respectively list and perform all the possible reductions of the graph in iteration i. Further follows a detailed explanation of the two operations.

Algorithm 3.1 `get_canon_repr`(G)

Input:
graph $G = (V, E, \lambda)$
Output:
the canonical string representation of G

```
1: i := 0
2: H_i := the hypergraph equivalent of G
3: while |V_i| > 1 do
4:     R_i := get_reducible_features(H_i)
5:     if R_i = ∅ then
6:         return NULL // The graph is not a partial 3-tree
7:     end if
8:     H_{i+1} := reduce(H_i, R_i)
9:     i := i + 1
10: end while
11: v := the only vertex left in V_i
12: return λ_i(v)
```

Let $H_i = (V_i, E_i, \lambda_i)$ be the hypergraph produced by the algorithm from the input graph for iteration i. A reduction rule has as a left-hand side the pattern that needs to match some subgraph of H_i. The right-hand side of the rule is the resulting pattern after reducing the matching subgraph. The vertices affected by a reduction rule are split into two sets. Let A be the set of nodes which will not be present in H_{i+1} after the reduction step and let B be the minimal separator of A in H_i. Theorem 3.1 states the basic reduction rules applicable to graphs of tree-width at most 3. The same rules are visualized on Figure 3.1 where the black vertices belong to A and the white ones to B (the same visual notation is used for all following figures of reduction rules).

Theorem 3.1. *[9] The following reduction rules are confluent under a congruence relation under which all partial 3-trees are equivalent to the empty graph: isolated vertex removal, reduction of a vertex of degree 1, reduction of a vertex of degree 2, and a star-triangle substitution when (i) two of the neighbors of a vertex of degree 3 are adjacent (triangle rule), (ii) all the neighbors of a vertex of degree 3 are also neighbors of one other vertex of degree 3 (buddy rule), or (iii) the neighbors of a vertex of degree 3 are shared with those of two other vertices of degree 3 that also share a forth vertex (cube rule).*

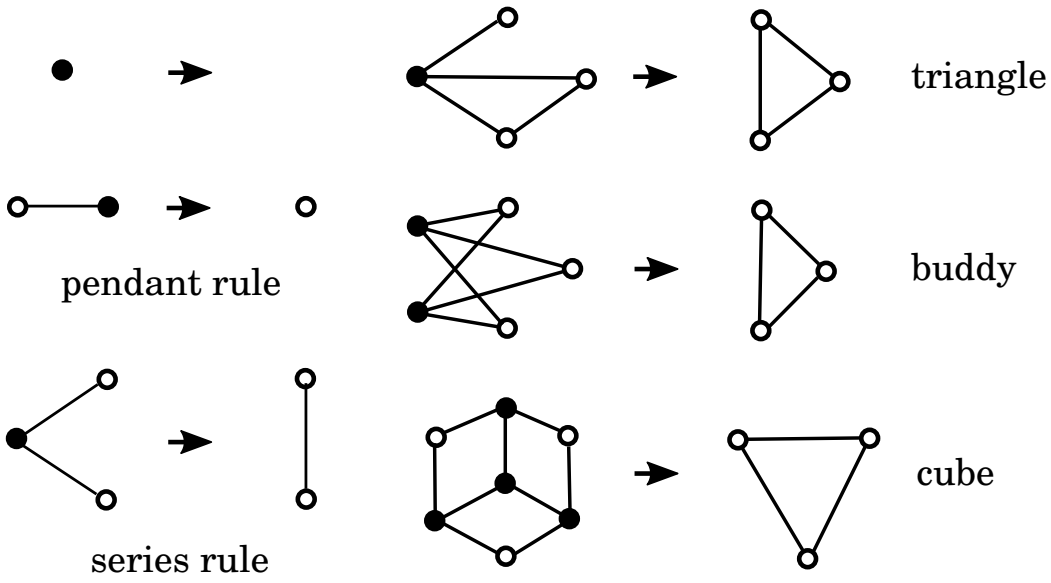


Figure 3.1: Basic reduction rules for partial 3-trees. Figure taken from [1].

The construction of the canonical string representation of a graph is done by applying sequentially the reduction rules. The list of specific rules, to be presented shortly, defines the order in which reductions are performed. The function `get_reducible_features`(H_i) returns all possible reductions on H_i which are instances of the first applicable rule starting from the beginning of the list. Function `reduce`(H_i, R_i) performs on H_i all reductions of the instances R_i returned by `get_reducible_features`(H_i). For vertices of degree 1 and 2, the reductions can be applied directly. The situation gets more complex when reducing nodes of degree 3. The problem comes from the fact that two or more nodes of degree 3 which are subject to reduction by the same rule may be adjacent. In this case, there is a *conflict*. All possible conflicting patterns are given a separate rule.

Some basic definitions are needed before listing the specific reduction rules. In the context of the Arnborg-Proskurowski algorithm, we will use the following definition of hyperedge direction. The *direction* of a hyperedge h will be regarded as a set of permutations $\{\pi_1, ..., \pi_k\}$ of its endpoints, where $\pi_i(v)$ is the index of $v \in h$ in the i-th permuted sequence of vertices. For example, if a 2-hyperedge is directed from vertex v_1 to vertex v_2, then its direction will be $\{\pi_0\}$, such that $\pi_0(v_1) = 0$ and $\pi_0(v_2) = 1$. The direction of an undirected hyperedge is implicitly the set of all permutations of its endpoints.

Let h be an n-hyperedge and $\pi_i^{-1}(q)$ be the vertex $v \in h$ for which it is true that $\pi_i(v) = q$, where π_i is a permutation in the direction of h. Let a

direction mask of an n-hyperedge $h = \{v_1, ..., v_n\}$ with direction $\{\pi_1, ..., \pi_k\}$ denote the lexicographically sorted k-tuple of n-tuples

$$((\rho(\pi_i^{-1}(0)), ..., \rho(\pi_i^{-1}(n-1))), ..., (\rho(\pi_j^{-1}(0)), ..., \rho(\pi_j^{-1}(n-1))))$$

where ρ is some permutation of a superset of h. In this case, π_i and π_j denote respectively the first and the last elements of the direction of h sorted with respect to the lexicographical order of the n-tuples. For example, if there is an edge pointing from v_1 to v_2 and a permutation ρ is given such that $\rho(v_1) = 1$ and $\rho(v_2) = 0$, then the direction mask of that edge will be $((1, 0))$. Let the *code* of an edge with respect to a permutation ρ of its endpoints be the tuple (l, d_ρ) where l is the label and d_ρ is the direction mask of the edge with respect to ρ. The code of an n-hyperedge with endpoints $\{v_1, ..., v_n\}$ may be denoted as $c(v_i, ..., v_j)$ where the sequence given in the signature of c defines ρ when the permutation covers exactly the set of endpoints. In this case, v_i and v_j are respectively the first and the last endpoints of the hyperedge ordered with respect to ρ.

A *sufficient set of permutations* denotes permutations of vertices of a subgraph that are able to represent all symmetries of the subgraph. Let a *basic degree 3 reduction* be the action of replacing a vertex of degree 3 with a 3-hyperedge between its neighbors. A set of degree 3 vertices R separated from the rest of the graph by vertices S can be reduced similarly to the basic degree 3 reduction. R are removed and a new hyperedge is produced with endpoints S and label the minimum string l_π among all permutations π in the sufficient set of permutations of $R \cup S$. In this case, l_π is a sorted concatenation of the labels of the removed nodes and the codes of all removed edges with respect to π. In the rule definitions below, all comparisons of labels and direction masks are done lexicographically. Each label generated by a certain rule starts with the number of the rule from the list.

In the context of the Arnborg-Proskurowski algorithm, vertices can have multiple labels (i.e., for any vertex v, $\lambda(v)$ returns a set of labels). Hyperedges, however, may only have a single label.

All safe reduction rules and subrules necessary for reducing a partial 3-tree to a single labeled vertex are listed below. In the following list, i always denotes the current iteration of the Arnborg-Proskurowski algorithm. The list follows closely the one given in [1].

0 Multiple vertices and edges. In this rule no vertices are reduced. It consists of the following subrules.

0.1 Vertex label merge. If a vertex v has more than one labels (i.e.,

$|\lambda_i(v)| > 1$), they are merged into one single label which is a concatenation of the sorted list of labels.

0.2 Parallel rule. All parallel edges between two nodes, if there are any, are replaced by a single edge. The label and the direction of the new edge are defined by the permutation of the endpoints yielding the smallest sorted list of codes of the parallel edges.

1 Pendant rule. The following subrules apply to nodes of degree 1.

1.1 Dipole. Pattern: Two adjacent nodes v_1 and v_2 of degree 1 forming a connected component. Reduction: The connected component is reduced to a single node with label the smallest of the strings $(\lambda_i(v_1), c(v_1, v_2), \lambda_i(v_2))$ and $(\lambda_i(v_2), c(v_2, v_1), \lambda_i(v_1))$.

1.2 Pendant vertices. Pattern: A vertex v with degree 1 adjacent to another vertex s of larger degree. Reduction: v is removed and the string $(c(s, v), \lambda_i(v))$ is added to the labels of s.

1.3 Self-loops. All self-loops of any vertex v are reduced and their codes are added to the labels of v.

2 Series rule. The subrules of this rule apply to nodes of degree 2.

2.1 Chain rule. Pattern: A maximal set of degree 2 vertices inducing a connected graph and ending in two vertices s_1 and s_2. Reduction: The degree 2 vertices are reduced to an edge between s_1 and s_2. The new edge has direction defined by one of the permutations $\{(s_1, s_2), (s_2, s_1)\}$ which yields the smallest concatenation of vertex labels and edge codes following the path of degree 2 vertices, and the label of the new edge is this concatenation.

2.2 Ring rule. Pattern: A connected component consisting of m degree 2 vertices forming a cycle. Reduction: It is reduced to a single vertex with label the smallest among the $2m$ strings built by collecting the vertex labels and edge codes following the ring in any direction starting at any node.

3 Parallel triangles. This rule is the same as rule 0.2 but regarding parallel 3-hyperedges.

4 Isolated degree 3 reduction instances. The following subrules apply to isolated instances of the degree 3 reductions.

4.1 Triangle. Pattern: A vertex v of degree 3 with at least two of its neighbors being adjacent and none the neighbors is triangle reducible. Reduction: v is reduced by a basic degree 3 reduction.

4.2 Buddy. Pattern: Two or more vertices of degree 3 having the same set of neighbors. Reduction: All of them are reduced by a basic degree 3 reduction.

4.3 Cube. Pattern: Three degree 3 vertices where each shares a neighbor with each of the other two and all of them have a forth common neighbor. Reduction: Each of the three vertices is reduced by a basic degree 3 reduction. The fourth vertex (the *hub*) becomes triangle reducible, if it has degree 3.

5 Conflicting triangles. This rule concerns possible configurations of two or more instances of triangle reducible vertices adjacent to each other.

5.1 Diamonds. This subrule applies to adjacent triangle reducible vertices which have two common neighbors.

5.1.1 K_4. Pattern: A 4-clique (all the vertices are triangle reducible). Reduction: Reduced to a single vertex. The sufficient set consists of all 24 permutations of the 4 vertices.

5.1.2 K_4^-. Pattern: A 4-clique without one edge (vertices v_1 and v_2 are of degree 3 and are reducible, the rest s_1 and s_2 are the separator). Reduction: Reduced to an edge between s_1 and s_2. The sufficient set of permutations consists of each of the two permutations of $\{s_1, s_2\}$ followed by each permutation of $\{v_1, v_2\}$.

5.2 Subgraph I. This subrule applies to adjacent triangle reducible vertices which have at most one common neighbor. Let I be a maximal connected subgraph of H_i consisting of triangle reducible vertices subject to reductions according to the triangle rule.

5.2.1 Degree 1 only. Pattern: Two adjacent vertices of degree 1 in I as visualized on Figure 3.2. Reduction: In case of a three vertex separator the reducible vertices are replaced by a 3-hyperedge connecting the separator vertices. In case of a four vertex separator, there must be some other applicable rule on H_i or else it is not a partial 3-tree.

5.2.2 Degree 1 and 2 or 3. Pattern: There is a vertex of degree 1 in

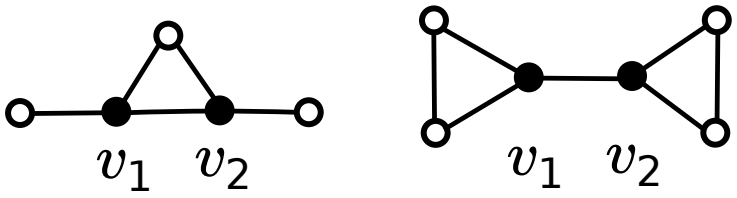
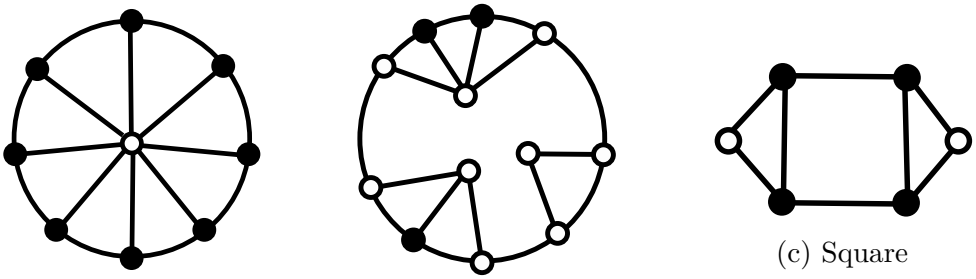


Figure 3.2: Possible configurations when all vertices in I are of degree 1. Figure taken from [1].

I. If there are more, they must be nonadjacent. Reduction: All vertices of degree 1 in I are reduced by basic degree 3 reductions.

5.2.3 Degree 2 only. All vertices of I have degree 2. There are three possible patterns for this case which are shown on Figure 3.3. Let any edge which is a side of a triangle in H_i be called t and let all other edges be called f. Let M be the set of all vertices in I incident to both a t and an f edge.



(a) Wheel (b) A general case (c) Square

Figure 3.3: Cycles in I. Figure taken from [1].

5.2.3.1 Wheel. Pattern: The set M is empty, thus all edges of I are t edges. Reduction: The wheel is reduced to a single vertex label of the center node.

5.2.3.2 Collection of paths. Pattern: M does not contain all vertices of I and its vertices partition the set of the remaining vertices of I into connected components. Reduction: Those connected components can be dealt with in a manner similar to that in rules 4.1, 5.2.1, and 5.2.2.

5.2.3.3 Square. Pattern: If I consists of alternating t and f edges, the only safe configuration is the one on Figure 3.3c. If the configuration is different, there must be some other applicable rule on H_i or else it is not

a partial 3-tree. Reduction: The configuration is reduced by a separate reduction rule.

5.2.4 Degrees 2 and 3. Pattern: When I consists of degree 2 and 3 vertices, then the vertices of degree 2 form in I paths that end in degree 3 vertices. Reduction: When such a path has exactly two degree 2 vertices, these can be reduced according to rule 5.2.1. Otherwise, there are unique and nonadjacent vertices of degree 2 in I and the corresponding vertices in H_i can be reduced with a basic degree 3 reduction, similarly to the situation in 5.2.2.

5.2.5 Degree 3 only (Prism). Pattern: All vertices of I have degree 3. The only safe configuration in this case is the prism, Figure 3.4. Reduction: Reduced to a single labeled vertex by a separate reduction rule.

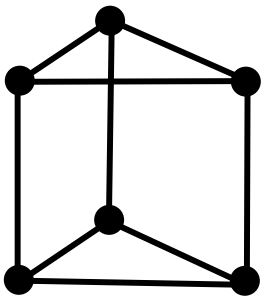


Figure 3.4: The prism pattern. Figure taken from [1].

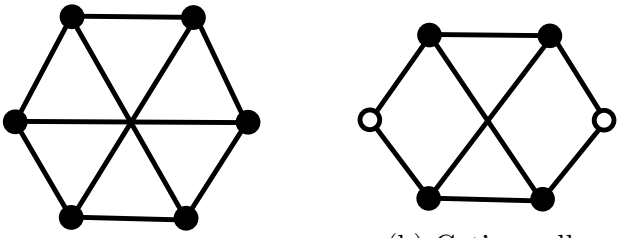
6 Conflicting buddies. This rule concerns the two possible configurations of two or more instances of buddy reducible vertices adjacent to each other.

6.1 $K_{3,3}$. Pattern: The graph on Figure 3.5a. Reduction: Reduced to a single labeled vertex by a separate reduction rule.

6.2 Cat's cradle. Pattern: The graph on Figure 3.5b. Reduction: The reducible vertices are replaced by an edge between the separator vertices.

7 Conflicting cubes. There are two possible cases of conflicting cubes.

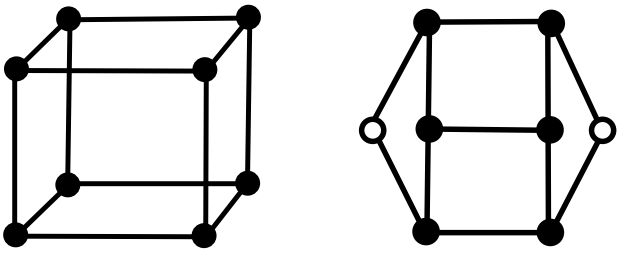
7.1 Cube. Pattern: A graph with 8 vertices and 12 edges forming a three dimensional cube, Figure 3.6a. Reduction: Reduced to a single labeled vertex by a separate reduction rule.



(a) $K_{3,3}$ (b) Cat's cradle

Figure 3.5: Conflicting buddy configurations. Figure taken from [1].

7.2 Hammock. Pattern: The graph on Figure 3.6b. Reduction: The reducible vertices are replaced by an edge between the separator vertices.



(a) Cube (b) Hammock

Figure 3.6: Conflicting cube configurations. Figure taken from [1].

The algorithm for building the canonical string representation of a partial 3-tree can be used also to compute the tree-width of a graph in the following way. If the graph has only one vertex, it has tree-width 0. Otherwise, if the graph is reduced to a single vertex using only rules 0 and 1, it has tree-width 1. If rules up to 2 are applied, the graph has tree-width 2. If some rule among 3, 4, 5, 6, and 7 is needed to reduce the graph, it has tree-width 3. If the algorithm is not able to reduce the graph to a single vertex, then it has tree-width greater than 3.

The canonical string representation can be computed in log-linear time or linear time and quadratic space [1]. The second case is if the algorithm is implemented using *ready lists* of candidate reducible vertices. A vertex may become a candidate for reduction in the initial step of the algorithm or in a subsequent iteration if its degree has changed. The degree of a vertex may change only after a reduction in its neighborhood. Therefore, keeping track of the neighborhood of already reduced nodes using ready lists decreases the search time for reducible vertices.

3.1.2 Example

Consider the following example, for the purpose of demonstrating a practical case of computing the canonical string representation of a partial 3-tree using the algorithm defined in this section. Let the input graph G be the one on Figure 3.7a. Each vertex of G has label a, b, c, or d and the label of each edge is x, y, or z. All reduction steps are visualized on Figure 3.7. Let

- $l_1 =$"`(1.2;(x,((1,0))),a)`",
- $l_2 =$"`(1.2;(x,((0,1))),a)`",
- $l_3 =$"`(y,((1,0)))`",
- $l_4 =$"`(y,((0,1)))`",
- $l_5 =$"`(0.1;`l_1`,b)`",
- $l_6 =$"`(0.1;`l_1`,c)`",
- $l_7 =$"`(0.1;`l_2`,c)`",
- $l_8 =$"`(2.1;`l_3`,`l_5`,`l_4`)`"

be strings. In the first iteration of the algorithm all vertices of degree 1 are reduced by rule 1.2. In the second iteration the multi-labels of the vertices are merged into single labels by rule 0.1. In the third iteration the algorithm reduces the only vertex of degree 2 to an edge between its neighbors by rule 2.1, and the graph turns into a 4-clique. In the last iteration the connected component is reduced to a single vertex by rule 5.1.1 with label

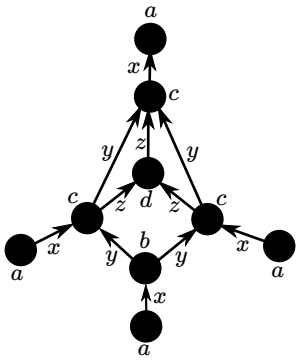
$$C = \text{"}\texttt{(5.1.1;(}l_8\texttt{,((0,1),(1,0))),}l_7\texttt{,}l_6\texttt{,}l_6\texttt{,(y,((0,2))),(y,((1,2))),}$$

$$\texttt{(z,((0,3))),(z,((1,3))),(z,((3,2))),d)}\text{"}$$

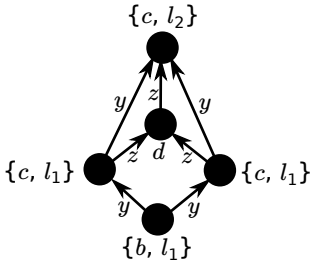
which is also the canonical string representation of G.

3.2 Auxiliary Methods

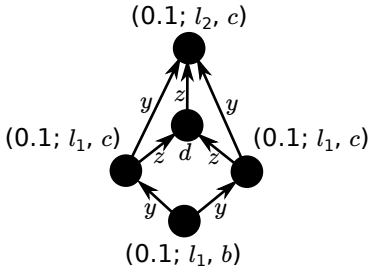
Before proceeding to the general scheme of the proposed algorithm for extraction of features from graphs of bounded tree-width, two inner functions need to be defined, `weisfeiler_lehman` and `get_w_shingles`.



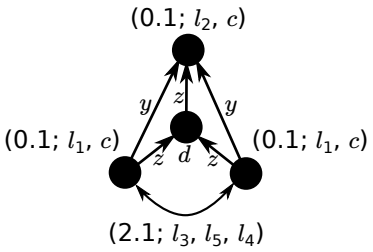
(a) $i = 0$ (initial graph)



(b) $i = 1$



(c) $i = 2$



(d) $i = 3$

(e) $i = 4$

Figure 3.7: Example of computing the canonical string representation of a partial 3-tree.

3.2.1 Function `weisfeiler_lehman`

The first function is the Weisfeiler-Lehman label refinement function defined as `weisfeiler_lehman`(G, i) where G is a graph and i is number of iterations of the Weisfeiler-Lehman method [8] discussed in Section 2.2. This function returns the graph G_i which is structurally the same as G but with vertex and edge labels constructed by the WL method until iteration i. Since the original WL method refines only the vertex labels, the work-around used here is to represent internally all edges of G as vertices of degree 2 connected by unlabeled edges to their endpoints. Let $B = (U_V \cup U_E, F, \lambda)$ be the bipartite graph obtained from G by turning all of its edges into vertices. U_V is the same as the set of vertices of G and U_E is the set of edges of G turned into vertices of B. Let $s_j(v)$ denote the string obtained at step 10 of Algorithm 2.1 for some vertex v of B at iteration $j \leq i$. Then $s_j(v)$ has two parts: prefix and augmentation. In order for the direction of edges in the graph to be encoded using the WL method, the following modification is applied. The augmentation part of $s_j(v)$ is split into three sections: *any*, *in* and *out*. The labels of all neighbors of v in B are distributed in each section in the following way:

- **any** - vertices connected to v by an undirected edge,
- **in** - vertices connected to v by an arrow pointing to v,
- **out** - vertices connected to v by an arrow going out of v.

The rest of the WL method operation performed by `weisfeiler_lehman`(G, i) remains as described in the preliminaries. Similarly to the WL graph kernel, exactly i iterations are executed on G without checking if the colors are stable. If $i = 0$, the initial graph is returned.

3.2.2 Function `get_w_shingles`

The shingling function `get_w_shingles`(C_i, w) maps the canonical string representation C_i of a graph G_i to a set of w-shingles [7]. The way of extracting the shingles is the same as described in Section 2.3. C_i is viewed as a sequence of character tokens.

3.3 Feature Extraction

The outer loop of the feature extraction algorithm is presented in this section. After defining it in Subsection 3.3.1, a practical example of its usage is presented in Subsection 3.3.2.

3.3.1 Definition

The steps of the method are listed in Algorithm 3.2. The labels in the partial 3-tree G are refined h times by the Weisfeiler-Lehman method [8]. For each step $i \leq h$ the canonical string representation C_i of G_i is mapped to a set of w-shingles S_i. All such sets are accumulated in set S which is the final result of the algorithm.

Algorithm 3.2 `feature_extraction`(G, w, h)

Input:
partial 3-tree G
$w > 0$, size of the w-shingles in number of characters
$h \geq 0$, number of Weisfeiler-Lehman iterations
Output:
set of w-shingles S

1: $S := \emptyset$
2: $i := 0$
3: **while** $i \leq h$ **do**
4: $G_i :=$ `weisfeiler_lehman`(G, i)
5: $C_i :=$ `get_canon_repr`(G_i)
6: $S_i :=$ `get_w_shingles`(C_i, w)
7: $S := S \cup S_i$
8: $i := i + 1$
9: **end while**
10: **return** S

3.3.2 Example

Let G be the graph on Figure 3.8. Then for $w = 5$ and $h = 1$, `feature_extraction`(G, w, h) will be executed in the following manner.

In iteration $i = 0$ the canonical string representation of $G_0 \equiv G$ computed in step 5 will be

$$C_0 = \text{“}\texttt{(2.2;a,(x,((0,1))),a,(x,((1,0))),b,(x,((0,1))),}$$
$$\texttt{b,(x,((1,0)))}\text{”}.$$

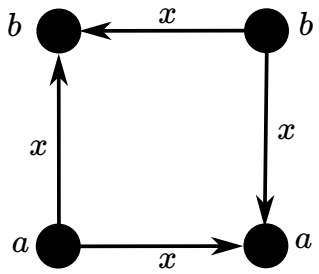


Figure 3.8: Example of a directed graph of tree-width 2.

The 32 5-shingles extracted from C_0 at step 6 will be

$$\begin{aligned} S_0 = \{&\text{“((0,1”}, \text{“((1,0”}, \text{“(0,1)”}, \text{“(1,0)”}, \\ &\text{“(2.2;”}, \text{“(x,((”}, \text{“))),a”}, \text{“))),b”}, \\ &\text{“)),a,”}, \text{“)),b,”}, \text{“),a,(”}, \text{“),b,(”}, \\ &\text{“,((0,”}, \text{“,((1,”}, \text{“,(x,(”}, \text{“,0)))”}, \\ &\text{“,1)))”}, \text{“,a,(x”}, \text{“,b,(x”}, \text{“.2;a,”}, \\ &\text{“0))))”}, \text{“0))),”}, \text{“0,1))”}, \text{“1))),”}, \\ &\text{“1,0))”}, \text{“2.2;a”}, \text{“2;a,(”}, \text{“;a,(x”}, \\ &\text{“a,(x,”}, \text{“b,(x,”}, \text{“x,((0”}, \text{“x,((1”}\}. \end{aligned}$$

In iteration $i = 1$ the canonical string representation of G_1 computed in step 5 will be

$$C_1 = \text{“(2.2;wl\_1.4,(wl\_1.0,((1,0))),wl\_1.5,(wl\_1.3,((0,1))),}$$

$$\text{wl\_1.6,(wl\_1.1,((1,0))),wl\_1.7,(wl\_1.2,((0,1))))”}.$$

where $\texttt{wl\_}i\texttt{.}x$ is the identifier of the augmented label of a vertex or an edge in iteration i of the WL method discussed in Section 2.2. The 65 5-shingles extracted from C_1 at step 6 will be

$S_1 = \{$“((0,1”,“((1,0”,“(0,1)”,“(1,0)”,“(2.2;”,“(wl_1”,
“))),w”,“)),wl”,“),wl_”,“,((0,”,“,((1,”,“,(wl_”,
“,0)))”,“,1)))”,“,wl_1”,“.0,((”,“.1,((”,“.2,((”,
“.2;wl”,“.3,((”,“.4,(w”,“.5,(w”,“.6,(w”,“.7,(w”,
“0))),”,“0,((1”,“0,1))”,“1))))”,“1))),”,“1,((1”,
“1,0))”,“1.0,(”,“1.1,(”,“1.2,(”,“1.3,(”,“1.4,(”,
“1.5,(”,“1.6,(”,“1.7,(”,“2,((0”,“2.2;w”,“2;wl_”,
“3,((0”,“4,(wl”,“5,(wl”,“6,(wl”,“7,(wl”,“;wl_1”,
“_1.0,”,“_1.1,”,“_1.2,”,“_1.3,”,“_1.4,”,“_1.5,”,
“_1.6,”,“_1.7,”,“l_1.0”,“l_1.1”,“l_1.2”,“l_1.3”,
“l_1.4”,“l_1.5”,“l_1.6”,“l_1.7”,“wl_1.”}.

The result of `feature_extraction`(G, w, h) will be the set $S = S_0 \cup S_1$ with cardinality 84.

3.4 Theoretical Evaluation

The quality of the algorithm can be evaluated according to conditions 1, 2 and 3 defined at the beginning of this chapter. The focus of this section is to investigate conditions 1 and 2, whereas the focus of Chapter 5 is condition 3.

The following claims address the time complexity and correctness of the mapping according to conditions 1 and 2.

Claim 3.1. *For given w, h, and a partial 3-tree G,* `feature_extraction`(G, w, h) *can be computed in* linear *in the size of G time.*

Proof. As mentioned earlier, line 4 of Algorithm 3.2 can be computed in linear in the size of the input graph time [39]. The canonical string representation in line 5 can also be computed in linear time using ready lists [1]. The extraction of w-shingles from the canonical string representation of the graph (line 6) is done in linear of the length of the string time using a sliding window of size w. If the labels in the graph are represented as short identifiers of their actual values, then the canonical string representation of the graph will have length which is linear in the size of the graph. This is because the label of each element of the graph (vertex or edge) appears only once in the canonical string representation. Therefore, shingling of the canonical string

representation is also done in linear of the size of the graph time. Altogether, feature extraction using the proposed algorithm can be done in *linear* of the size of the input graph time. □

Claim 3.2. *For given w, h, and any two partial 3-trees G and G', if G is isomorphic to G', then*

$$\texttt{feature\_extraction}(G, w, h) = \texttt{feature\_extraction}(G', w, h).$$

Proof. Let γ be isomorphism between $G = (V, E, \lambda)$ and $G' = (V', E', \lambda')$. For $\texttt{feature\_extraction}(G, w, h)$ let G_i, C_i and S_i denote respectively the results of lines 4, 5 and 6 in iteration i of Algorithm 3.2. Let G'_i, C'_i and S'_i denote the same for $\texttt{feature\_extraction}(G', w, h)$.

Assume $\texttt{feature\_extraction}(G, w, h) \neq \texttt{feature\_extraction}(G', w, h)$. There must exist some iteration i for which $S_i \neq S'_i$, which means $C_i \neq C'_i$, and therefore G_i is not isomorphic to G'_i (by the definition of canonical string representation).

If G_i is not isomorphic to G'_i, there must exist some $x \in V \cup E$ with $\gamma(x) = x' \in V' \cup E'$ for which $\lambda_i(x) \neq \lambda'_i(x')$. Therefore, by Weisfeiler-Lehman method [8], G_{i-1} is not isomorphic to G'_{i-1}. By induction, $G_0 \equiv G$ is not isomorphic to $G'_0 \equiv G'$, but this is a contradiction. Therefore, $\texttt{feature\_extraction}(G, w, h)$ and $\texttt{feature\_extraction}(G', w, h)$ must be the same. □

Claim 3.3. *For given w, h, and any two partial 3-trees G and G', if $\texttt{feature\_extraction}(G, w, h) = \texttt{feature\_extraction}(G', w, h)$, then G and G' are not always isomorphic to each other.*

Proof. The claim can be trivially proven for $w = 1$ (i.e., G and G' will be mapped to the same feature vectors if they use the same alphabet in their labels). Let us show that the claim is true also for larger w by using the following example. Let G be the graph on Figure 3.8, G' be the graph on Figure 3.9, $w = 5$ and $h = 0$. G is not isomorphic to G'. The canonical string representation of G is

$$C_0 = \text{“}\texttt{(2.2;a,(x,((0,1))),a,(x,((1,0))),b,(x,((0,1))),}$$

$$\texttt{b,(x,((1,0))))}\text{”},$$

and the canonical string representation of G' is

$$C'_0 = \text{“}\texttt{(2.2;a,(x,((0,1))),b,(x,((0,1))),a,(x,((1,0))),}$$

$$\texttt{b,(x,((1,0))))}\text{”}.$$

C_0 and C'_0 are not the same, however the shingles produced by `get_w_shingles`(C_0, w) and `get_w_shingles`(C'_0, w) are both equal to S_0 from Subsection 3.3.2. Therefore, `feature_extraction`(G, w, h) and `feature_extraction`(G', w, h) are identical. □

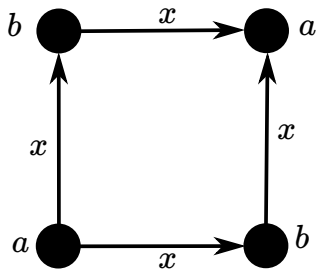


Figure 3.9: A second example of a directed graph of tree-width 2 yielding the same set of features as the first one.

In summary, the proposed mapping is fast to compute and always maps isomorphic graphs to the same set of features. However, two graphs may be mapped to the same features even if they are not isomorphic. For larger w and h it is much less probable to find a pair of non-isomorphic graphs which produce identical sets of features.

Chapter 4

Implementation

In this chapter will be discussed an overview of the techniques used for implementing the described feature extraction algorithm. The implementation[1] is used for performing the experiments presented in Chapter 5. The programming language in which the project is written is Python (version 2.7). The graph library NetworkX[2] (version 1.10) is used for performing all graph related operations. The RDF data is processed using the RDFLib[3] library (version 4.2.1).

The developed project consists of two conceptual parts: algorithmic and data processing. The algorithmic part consists of all modules which participate in the feature extraction process. The implementation of some of the modules (e.g., the shingling of canonical string representations) is trivial and that is why we will not cover them in this chapter. We will focus on the implementations of the Arnborg-Proskurowski algorithm [1] for computing the canonical string representations of graphs and the Weisfeiler-Lehman algorithm [8]. The data processing part consists of all functions used to process the input datasets and the output results. This chapter is structured in the following way. After discussing some general implementation details, Section 4.1 describes the implementation of the Arnborg-Proskurowski algorithm, Section 4.2 the implementation of the Weisfeiler-Lehman algorithm, and Section 4.3 the data processing part of the project.

The NetworkX library provides the class Graph which defines the concept of graphs. For the specific requirements of the implementation of the feature extraction method, an encapsulation of this class is created which extends its functionalities. The new class is called Hypergraph and sup-

[1]The source code of the project is available at `https://github.com/idanivanov/master_thesis`.

[2]`https://networkx.github.io/`

[3]`https://github.com/RDFLib/rdflib`

ports also edges of degree different from 2 (i.e., it supports the instantiation of hypergraphs). Some of its added functionalities, however, are restricted to hyperedges with degree at most 3. A hypergraph $H = (V_H, E_H, \lambda_H)$ is internally represented by the class as the bipartite graph $B = (V_H \cup E_H, E, \lambda_H)$ where an edge $e = \{v, h\}$ belongs to E if and only if $u \in V_H$ is a vertex in H and $h \in E_H$ is a hyperedge in H. Informally, all vertices and hyperedges of H are represented as vertices in B. If a hyperedge h is incident to a vertex v in H, then h is connected via an edge to v in B. The HYPERGRAPH class is specifically designed to be used with the implementation of the Arnborg-Proskurowski algorithm. When a hypergraph is instantiated, there are the following helper sets which are maintained in the object:

- **Nodes with more labels** - the set contains all vertices which have more than one labels;
- **Self loops** - the set contains all hyperedges of degree 1;
- **Groups of parallel n-hyperedges** - there is a separate set for each $n \in \{2, 3\}$ containing all n-hyperedges with exactly the same endpoints;
- **Nodes with n neighbors** - there is a separate set for each $n \in \{1, 2, 3\}$ containing all vertices with degree n.

These sets are maintained in order to reduce the search time for these particular elements in the hypergraph. They are initialized in the constructor of the class and are updated every time an element in the hypergraph is changed (i.e., the changed element is checked if it should be added to or removed from some of the sets and the action is performed). This leads to slow instantiation of hypergraphs, slower `add` and `remove` operations, but more efficient search of elements during the execution of Arnborg-Proskurowski.

The project covers the process from reading the input dataset to outputting the binary feature vectors. The machine learning algorithms used for obtaining the results presented in Chapter 5 are not included in the implementation because they are applied separately from external sources (e.g., the *scikit-learn*[4] library) after the feature extraction is performed. Therefore, they will not be discussed in this chapter.

4.1 Arnborg-Proskurowski

The algorithm proposed by Arnborg and Proskurowski [1] discussed in Chapter 3 is implemented with the capability for solving three tasks:

[4] `http://scikit-learn.org`

- Build the canonical string representation of a graph;
- Compute the tree-width of a graph, if it is up to 3 or else return -1;
- Return all rule instances reduced by the algorithm.

The input graph which is given to the algorithm is first converted to a hypergraph (i.e., instance of class HYPERGRAPH). This is because, if the graph has tree-width larger than 2, there will be 3-hyperedges generated from the degree 3 reduction rules. As stated at the beginning of this chapter, the HYPERGRAPH class implements the *ready lists* which were discussed at the end of Subsection 3.1.1. This makes the implementation of the algorithm to execute in linear time.

As described in Section 3.1, the Arnborg-Proskurowski algorithm is based on the canonical reduction of safe reduction rules. In the implemented version of the algorithm, the instances of safe reduction rules are objects of class REDUCIBLEFEATURE. Such an object contains two sets of vertices: *reducible nodes* (i.e., vertices which will be removed from the graph after the reduction step) and *peripheral nodes* (i.e., the separator of the reducible nodes in the graph). The static methods of this class are dealing with the search and extraction of applicable instances of safe reduction rules in a given graph. These functions identify patterns in the graph, using the ready lists provided by the HYPERGRAPH class, and instantiate REDUCIBLEFEATURE objects. The patterns are identified according to the rules listed in Subsection 3.1.1. The objects of type REDUCIBLEFEATURE provide the method `reduce`() which reduces the respective rule instances in the graph.

The reduction instances can be returned by the algorithm. This may be helpful in cases that the user needs to know how the reduction process was performed. These instances can also be used as features of the graph[5].

4.2 Weisfeiler-Lehman

The implementation of the Weisfeiler-Lehman algorithm [8] follows the principle defined in Subsection 3.2.1. The HYPERGRAPH class provides the bipartite representation of the graph, such that the hyperedges can also be taken into account when refining the labels according to the WL method.

[5]A feature extraction method based on such features was also considered in the process of development. However, the currently presented feature extraction method outperforms in the classification tasks the one where features are the instances of reduction rules and is much simpler to implement.

The current version of the WL method implementation does not support hyperedges with more than 2 endpoints. This is not a problem for the feature extraction method because the WL algorithm is applied only to the initial version of the input graph which does not have any hyperedges with more than 2 endpoints. At each iteration of the algorithm, the new labels of the graph have the format "`wl_`i`.`n", where i is the iteration and n is a number that is incremented every time a new label is generated for iteration i starting from $n = 0$. This format is used for the global identifiers of the augmented labels in the graph.

The implemented function performing the WL algorithm returns not only the produced graph but also the state of the refinements. This state can be reused in other invocations of the function. It consists of the following elements:

- **labels** - a dictionary in which the keys are the augmented labels produced by the algorithm and the values are the identifiers of the augmented labels using the previously mentioned format;
- **next identifiers** - a dictionary in which the keys are the iterations of the algorithm and the values are the the next values for n in the format of the identifiers for each iteration.

When this state information is provided in a new run of the WL algorithm, the augmented labels which are produced are first checked if they are present in the **labels** of the state. If this is the case, then the corresponding identifiers are reused. If a newly produced augmented label is not present in the state so far, then a new identifier is generated with the corresponding value of n from the **next identifiers**. The new augmented label and its identifier are added to the state and the **next identifiers** value for the current iteration is incremented. Using this state information for all runs of the algorithm, it is guaranteed that the generated identifiers will be global for all graphs in the database.

4.3 Data Processing

The input datasets can be in two formats which will be discussed in more details in Chapter 5. When the data is in RDF format, it is loaded using the RDFLib library. In all cases, when dealing with RDF data in the current project, it is necessary to extract the neighborhoods of some RDF resources. This is done by a function which extracts the r-ball around a particular resource (i.e., a subgraph containing all neighboring vertices reachable with

at most r hops starting from the vertex representing the resource). This can be done either by loading the whole dataset as a graph and extracting the r-balls from this graph or by querying the RDF endpoint directly to get each r-ball individually (e.g., when working with DBpedia it is not possible to load the whole dataset in memory, so the neighborhoods are extracted using SPARQL queries to the RDF endpoint).

The output of the feature extraction algorithm is a text file in which each line corresponds to a record in the input database. A record in the input database consists of a graph and its target labels (e.g., the types of an RDF resource of interest). A line in the output file has the following format:

$$l_1,l_2,...,l_k \ f_1{:}b_1 \ f_2{:}b_2 \ ... \ f_d{:}b_d$$

where $l_1, l_2, ..., l_k$ are the target labels of the record, followed by a sequence of $f_i{:}b_i$ where f_i is the identifier[6] of a feature and b_i is 1 if this feature was extracted from the graph of the record or 0 otherwise. Usually, a sparse version of this format is used, such that if b_i is 0, the feature is omitted from the line. When dealing with concept learning problems, each line has exactly one target label l which can be 1 if the example is positive, or -1 if it is negative. If the classes are binary, the output format is compatible with the SVM library *LIBSVM* and can be directly used as an input for it.

[6]The identifier of a feature is the Rabin fingerprint of the extracted shingle as mentioned in Section 2.3.

Chapter 5

Empirical Evaluation

A number of application areas deal with graphs of simple structure. The algorithm discussed in Chapter 3 is not bound to a specific area and can be applied on general graph structures which have tree-width smaller than 4. The performance of the algorithm was empirically tested in two application areas. The results from these experiments will be discussed in this chapter. The first field is the prediction of chemical activity of molecules, presented in Section 5.1. The second is the prediction of resource types in the DBpedia RDF dataset, discussed in Section 5.2.

Both application examples in this chapter are dealing with classification problems. Let D be a *graph database*, such that each record $r \in D$ is a tuple (G, T), where G is a graph and T is a set of target labels of r. In case of binary classes, T is a singleton set containing a boolean value. Using feature extraction, the graph database is mapped to a database B in which each record $r_B \in B$ is a tuple (b, T), where b is a binary vector of features. The classification problem can be solved on B using a standard method for classification. The classification algorithm used for the experiments in this chapter is a *Support Vector Machine* (*SVM*). The SVM implementation used here is the C-SVC provided by *LIBSVM*[1].

The predictive performance of the models is scored from a 10-fold cross-validation. In each run, B is split randomly into two disjoint subsets B_{train} and B_{test}, such that B_{test} contains one tenth of the examples in B and B_{train} contains the rest of B. The SVM is trained on B_{train} and its predictive power is evaluated on the unseen examples in B_{test}. This process is repeated 10 times. The final scores are the mean and standard deviation calculated among all 10 runs.

The general setup for all experiments consists of the following steps:

[1] `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`

1. Load the dataset
2. Preprocess the data and build a graph database D
3. Extract features from the graphs in D and build B
4. Grid search for best SVM parameters using 10-fold cross-validation

The first step is to load the data in main memory. A dataset can be loaded from different formats. For the current experiments there are two possible formats. In the first one, the data are given as *ordinary graphs*. More precisely, they are a set of tuples (G, T) where G is a graph and T are target labels. Each tuple stands for a record in the dataset and its graph is defined by its vertices and edges. All elements of the graph are labeled. In this case the preprocessing step is omitted because the graph database is clearly defined and feature extraction can be applied directly.

The other format is the *Resource Description Framework* (*RDF*) in which the data are given as a set of RDF triples that can be viewed as a network of resources inter-connected by predicates. RDF data contain a lot of meta-information which may not be useful when applying feature extraction. Therefore, a good preprocessing heuristic should be used to clean the graph from the unnecessary information. The preprocessing used for the experiments in this chapter is the following: Let G_{RDF} be the initial RDF graph. G_{RDF} consists of an A-box (i.e., definitions of instances and links between them) and a T-box (i.e., schema or ontology). From G_{RDF} we want to obtain the new preprocessed graph $R = (V, E, \lambda)$. Let initially R be the same as the A-box of G_{RDF}, such that each RDF resource is represented by a vertex in R and each triple is represented by an edge in R. The direction of the edge representing triple (s, p, o) is from the vertex representing s to the vertex representing o. For each vertex $v \in V$, let $\lambda(v) := C_v$, where C_v is the set of all RDF types[2] of the resource in G_{RDF} corresponding to v. For each edge $e \in E$, let $\lambda(e) := \{p\}$, where p is the predicate of the triple (s, p, o) in G_{RDF} corresponding to e. An example of the described process is depicted on Figure 5.1. Using this preprocessing the elements of graph R have more generalized labels which will result in larger number of shared features when applying feature extraction for our experiments. On the other hand, if the labels of the elements were the specific URIs of the resources, the shared features would have been much less. In addition, this preprocessing technique may reduce the tree-width of the graph, which can be seen on Figure 5.1.

[2]If v represents a boolean literal, the value of the literal (true or false) is encoded in the label of v.

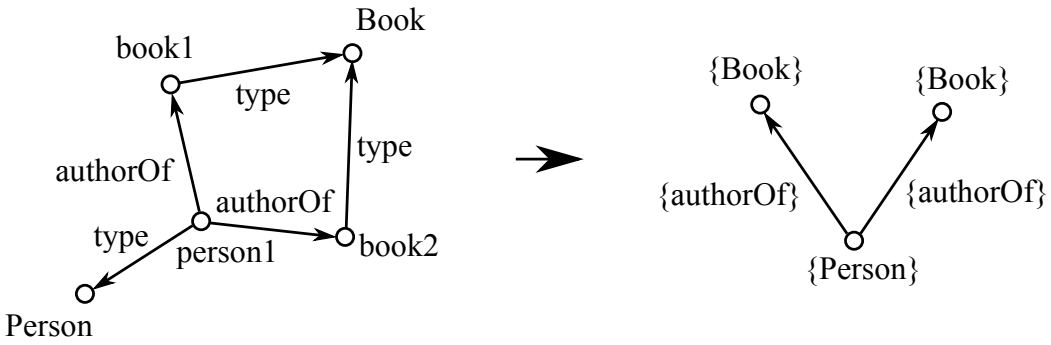


Figure 5.1: Example of preprocessing an RDF graph describing a person authoring two books.

Once R is obtained, the question remains how to define the graph database D. This is trivial for the ordinary graph representation, however, when dealing with RDF data a second level of preprocessing is needed. Building D can be done differently for the specific problem that needs to be solved. Therefore, the second preprocessing layer will be discussed additionally later in the context of a given problem.

The third step in the experimental process is to perform feature extraction on the graph database. This is done separately for each record in the database. However, a global state of the algorithm is maintained, so that whenever a feature is extracted from two graphs it triggers the same dimension of the extracted binary vectors. In this step there are two parameters that need to be taken into account: h (i.e., the number of Weisfeiler-Lehman iterations) and w (i.e., the size of the shingling window). Different values of these parameters lead to different classification performances.

Once the binary vector database B is obtained, it is necessary to find the best parameter setting of the SVM using cross-validation. When the kernel of the SVM is the *Radial Basis Function* (*RBF*), it is necessary to find the best values of γ (the only kernel parameter) and C (i.e., the trade-off between training error and margin). This has been done using grid search over a number of chosen values for both parameters.

Besides the classification scores that can be achieved using the proposed feature extraction method, it is also important to evaluate the runtime of the algorithm. The most time consuming step of the feature extraction process is the building of the canonical string representation of the input graph. As stated in Chapter 3, the extraction of canonical string representations of graphs is done in linear in the size of the input graph time. To show that this is practically the case, the following experiment was performed. For all n in $\{10, 100, 1000, 5000, 10000\}$ do: Generate 50 random graphs with n

vertices using the Erdős-Rényi model for $p = 1/n$ and calculate the mean time for computing their canonical string representations. The results of this experiment are depicted on Figure 5.2 with the range at each point denoting the standard deviation of the computation time. One can see that indeed the time increases linearly with the number of nodes. The average tree-width of the generated graphs is 2.088 ($\pm$0.790).

The runtime evaluations presented in this chapter were performed on a computer with Intel® Core™ i5-2450M CPU at 2.50GHz and 6GB of main memory.

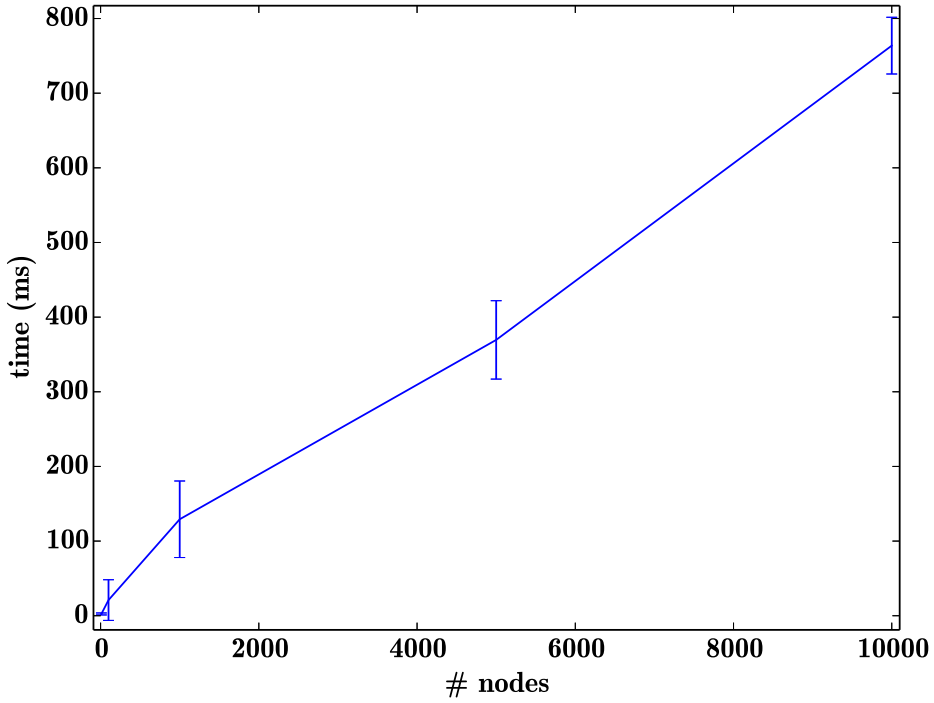


Figure 5.2: Time needed for extracting the canonical string representations of random graphs with different number of nodes.

5.1 Prediction of Chemical Properties

Testing the properties of a chemical compound in a laboratory is an expensive process which demands the physical presence of the substance. The knowledge if a molecule is active against some disease is very important for

pharmacological companies and the minimization of resources to acquire this knowledge is of big interest for them. The activity of chemical compounds is related to their molecule structure. This structure can be seen as an undirected labeled graph in which the atoms are vertices and the bonds between the atoms are edges. Therefore, computational models which can do classification on graph structured data can be applied for predicting the chemical activity of compounds.

There are a number of widely used benchmark pharmacological datasets. The performance of the proposed algorithm is tested on the following tree such datasets.

The *NCI-HIV* dataset[3] contains 42 687 compounds described by their structure and labeled by their capability to inhibit the HIV virus. In the dataset, 422 compounds are labeled as active (CA), 1081 are moderately active (CM), and 41 181 are inactive (CI). The dataset is used in the same way as in [11]. We investigate three concept learning problems for this dataset:

- **CA vs CM** - classify active against moderately active molecules,
- **CA + CM vs CI** - classify active and moderately active against inactive molecules,
- **CA vs CI** - classify active against inactive molecules.

The *Mutagenesis* dataset [43] consists of 230 aromatic and heteroaromatic nitro compounds. The goal is to predict if a compound is mutagenic or not. The set is divided into two subsets according to the original paper: the first contains 188 compounds, and the second – 42. Each subset is regarded as a separate concept learning problem.

The *Carcinogenesis* dataset[4] is distributed as an example dataset for the DL-Learner [44]. It contains 340 molecules. We investigate three concept learning problems for this dataset. The CAR-298 problem is the prediction for carcinogenicity of 298 molecules using the given labels from the DL-Learner website. The CAR-337 is similar to the previous problem but includes also the 39 PTE-1 challenge molecules provided in the same package. The MUTAG problem is the prediction of the *isMutagenic* property which is present in the RDF definition of all 340 compounds.

For our experiments we use the chemical datasets in the following formats. The NCI-HIV dataset is used only in ordinary graph format. The full Mutagenesis dataset is used in RDF format[5] and the 188 subset is also

[3] `http://cactus.nci.nih.gov/`

[4] `http://dl-learner.org/community/carcinogenesis/`

[5] `https://github.com/AKSW/SML-Bench`

investigated in ordinary graph representation. The Carcinogenesis dataset is used only in RDF format.

For the RDF representation of chemical data, a second layer of preprocessing is applied in order to generate the graph database D from the preprocessed graph $R = (V, E, \lambda)$, discussed in the beginning of the chapter. In all of the used RDF chemical datasets, for each chemical compound there is an RDF resource that represents it as an entity. Let the vertex in V representing such a resource be called the *entity vertex* of a compound. The entity vertex of a compound is connected to other vertices in the graph that define the molecule structure of the compound or some literal values. There are two ways for building the graph database D used in the experiments in this section.

- *Pre-1* - Let $D := \emptyset$ be an empty graph database. For each compound c in the dataset do: Let $v_c \in V$ be the entity vertex of c. Let G_c be the subgraph of R containing v_c and the neighborhood of v_c with radius 2 hops[6]. Then remove all literals from G_c and also remove v_c (i.e., only the molecule structure of c will remain in G_c). Create a new record $r_c = (G_c, \{t_c\})$, such that t_c is the target label of c. Add r_c to D.

- *Pre-2* - This method is similar to Pre-1 but it differs from it in the following way. G_c initially also contains v_c and the neighborhood of v_c with radius 2 hops. Afterwards, all literals except the ones representing boolean values are removed from the graph. This way, G_c remains with v_c, the molecule structure of c, and the boolean literals (which may bring important information about the activity of the compound).

When applying the Pre-2 preprocessing in the case of the MUTAG problem of the Carcinogenesis dataset, the *isMutagenic* property is removed from the graphs.

The feature extraction method discussed in Chapter 3 can be applied to the current problems because most of the graphs representing molecule structures in the three datasets have small tree-width. The tree-width distribution among the compounds in the datasets is shown on Table 5.1. One can see that when the Pre-1 preprocessing method was used on RDF data the graphs had smaller tree-width, compared to the ones when using the Pre-2 method. This is because the entity node v_c, which is connected to all other nodes in the extracted neighborhood, is missing in Pre-1 but is present in Pre-2.

[6]The radius 2 is chosen because in the first hop are extracted all direct neighbors of v_c and in the second hop are extracted all edges between the neighbors of v_c.

Dataset	Format	Preproc.	Tw = 1	Tw = 2	Tw = 3	Tw ≥ 4	Total
NCI-HIV	Ordinary	-	1655 (4 %)	39 098 (92 %)	1911 (4 %)	23 (0 %)	42 687
Mutagenesis	Ordinary	-	0 (0 %)	125 (66 %)	62 (33 %)	1 (1 %)	188
	RDF	Pre-1	0 (0 %)	161 (70 %)	67 (29 %)	2 (1 %)	230
	RDF	Pre-2	0 (0 %)	0 (0 %)	161 (70 %)	69 (30 %)	230
Carcinogenesis	RDF	Pre-1	84 (25 %)	253 (74 %)	3 (1 %)	0 (0 %)	340
		Pre-2	0 (0 %)	84 (25 %)	253 (74 %)	3 (1 %)	340

Table 5.1: Tree-width of the graphs in the chemical datasets.

Dataset	Format	Preproc.	Average number of nodes	Average number of edges
NCI-HIV	Ordinary	-	45.708 (±23.685)	47.713 (±24.574)
Mutagenesis	Ordinary	-	26.027 (±6.301)	27.888 (±7.440)
	RDF	Pre-1	60.500 (±17.934)	54.861 (±15.842)
	RDF	Pre-2	63.000 (±18.277)	117.361 (±33.740)
Carcinogenesis	RDF	Pre-1	64.800 (±49.460)	54.806 (±44.309)
	RDF	Pre-2	67.656 (±49.482)	125.476 (±93.449)

Table 5.2: Number of nodes and edges of the graphs in the chemical datasets.

The average size of the graphs in each dataset is shown Table 5.2. The classification results for the datasets in ordinary graph representation are shown on Table 5.3 and the results for the RDF datasets are shown on Table 5.4. The presented scores are only using the parameters that have maximized the performance of the method on the investigated problem.

For the NCI-HIV dataset it can be seen that the results for problems CA vs CM and CA + CM vs CI are worse, compared to CA vs CI. This leads to the conclusion that the separation between active and inactive compounds is high in the feature space. Therefore, the feature extraction method has successfully preserved the locality of the data (i.e., keeping similar graphs close and dissimilar graphs apart). The moderately active compounds seem to be mapped mostly in the space between the active and inactive ones.

The Mutagenesis dataset is investigated in two different formats. It can be seen that the results on the RDF representation of the dataset are slightly

Problem	w	h	AUC	F1	Precision	Recall	Accuracy
NCI-HIV (CA vs CM)	10	5	.842(.032)	-	-	-	-
NCI-HIV (CA + CM vs CI)	5	5	.841(.010)	-	-	-	-
NCI-HIV (CA vs CI)	5	4	.949(.028)	-	-	-	-
Mutagenesis (188)	9	1	-	.955(.028)	.971(.047)	.944(.050)	.942(.036)

Table 5.3: Classification results for chemical datasets in ordinary graph representations.

Problem	Preproc.	w	h	F1	Precision	Recall	Accuracy
Mutagenesis (188)	Pre-1	6	3	.941(.041)	.936(.064)	.951(.055)	.921(.057)
Mutagenesis (42)	Pre-1	5	1	.767(.396)	.800(.400)	.750(.403)	.915(.138)
Carcinogenesis (CAR-298)	Pre-1	6	0	.710(.063)	.676(.082)	.759(.094)	.663(.081)
	Pre-2	9	5	.865(.127)	.841(.146)	.910(.135)	.840(.151)
Carcinogenesis (CAR-337)	Pre-1	13	1	.692(.107)	.648(.091)	.754(.156)	.644(.111)
	Pre-2	17	5	.833(.136)	.830(.139)	.849(.155)	.815(.157)
Carcinogenesis (MUTAG)	Pre-1	15	3	.748(.149)	.828(.102)	.713(.203)	.833(.076)
	Pre-2	5	3	.675(.147)	.755(.158)	.651(.174)	.773(.086)

Table 5.4: Classification results for chemical datasets in RDF format.

worse than the ordinary graph representation. This may be due to the fact that in the RDF dataset each bond is present as a vertex connected to two other vertices (atoms) via edges, whereas the same bond in the ordinary graph representation is an edge between the two atoms. This makes the RDF graphs twice bigger compared to the ordinary graphs, which can also be seen on Table 5.2. The additional size of the RDF compounds does not bring any useful information for classification and actually seems to be leading to worse results.

The problems for the Carcinogenesis dataset are investigated using both of the preprocessing techniques Pre-1 and Pre-2. Interestingly, while Pre-2 yields much better results for CAR-298 and CAR-337, compared to Pre-1, it gives worse performance on MUTAG, compared to the performance given by Pre-1. This means that each of the two preprocessing techniques may be better or worse depending on the problem that needs to be solved.

Figure 5.3 shows how the average time needed for extracting features from a compound in the Carcinogenesis dataset depends on the number of Weisfeiler-Lehman iterations (h). For this experiment w was set to 10 and the preprocessing was Pre-2. One can see that the time increases linearly in h.

The results presented in this section show that the proposed feature extraction method performs well on predicting the properties of chemical compounds. The achieved scores for NCI-HIV can be compared to the ones in [11, 45]. The scores for Mutagenesis can be compared to [46]. The scores for Carcinogenesis can be compared to DL-Learner[7] and [27].

[7] `http://dl-learner.org/community/carcinogenesis/#h672-4`

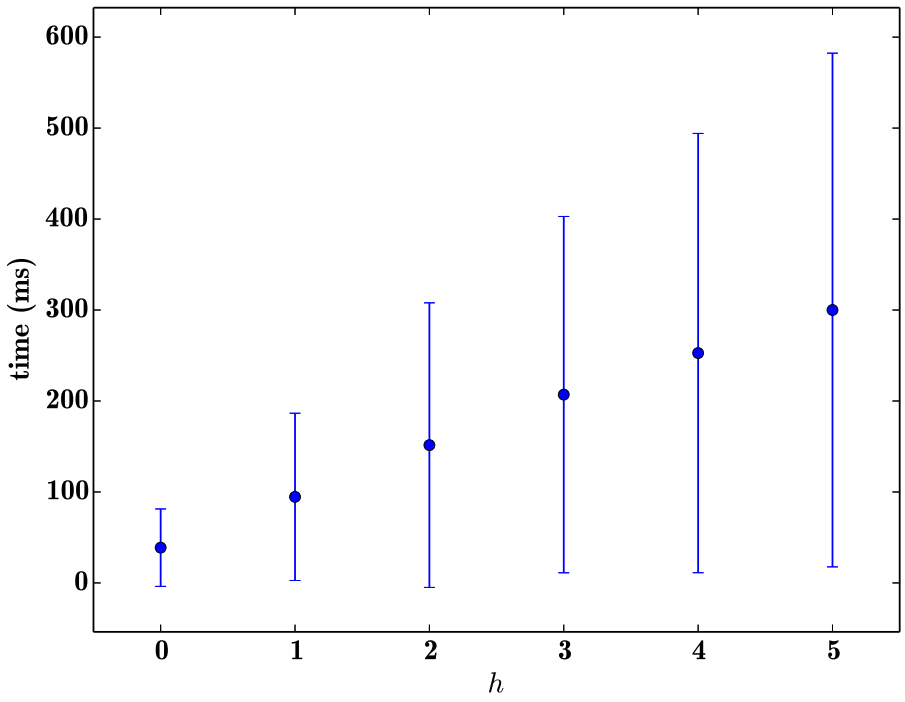


Figure 5.3: Average time for extracting features from a chemical compound graph in the Carcinogenesis dataset for different number of Weisfeiler-Lehman iterations (h).

5.2 Type Inference of DBpedia Resources

Information about the types of entities in knowledge bases is very important. It allows users to express practical queries and understand the data better. However, type information in knowledge bases is often incomplete. Therefore, automatic type inference methods can be beneficial in such cases. In RDF, a resource may have multiple types (e.g., the resource representing Leonardo da Vinci may have types Scientist and Artist). The standard way of inferring type information in RDF is by using logic-based reasoning. However, this approach may lead to the multiplication of errors in the database due to the fact that some resources may have wrong properties in a crowd-source dataset [6]. Therefore, the problem of inferring the types of RDF resources in a way that avoids the propagation of errors is important for such datasets. The problem can be defined in the following way: for a given RDF dataset and an untyped resource in this dataset, give the most probable types of this resource with respect to the distribution of types among the resources in the dataset.

DBpedia [12] is a Semantic Web project for translating data from Wikipedia infoboxes into structured data using RDF format. The structured information is openly available for querying on the World Wide Web. The DBpedia dataset is available for download[8] and has multiple versions. This dataset is a very good example of a real-world RDF graph with large coverage. It contains errors because it was build automatically from the user generated data on Wikipedia.

The setup of the DBpedia experiments discussed in this section is similar to the one used in [6]. The dataset used for the following experiments is DBpedia (version 3.8) when considering only the raw infobox data and the given type information of the resources. A sample of 10 000 resources with incoming degree of at least 25 is selected uniformly at random. Let this set of resources be denoted as I. A database D is built, such that each record in D represents a resource in I and its target labels. The target labels of a resource are its RDF types contained in the dataset. Therefore, each record in the database may have multiple target labels. The type inference process is done by performing multi-label classification on D. The performance is evaluated from a 10-fold cross-validation.

The RDF dataset is initially loaded and preprocessed using the method discussed in the beginning of this chapter. Let R be the graph obtained by preprocessing the RDF dataset. Each resource in I is also a vertex in R. Building the graph database D is done in the following way. Let initially

[8] `http://wiki.dbpedia.org/datasets`

rdf:type	Number of resources
owl:Thing	10 000
dbo:Agent	3319
schema:Place	2895
dbo:Place	2895
dbo:PopulatedPlace	2508
dbo:Settlement	1900
dbo:Organisation	1763
schema:Organization	1763
schema:Person	1556
foaf:Person	1556

Table 5.5: Ten most frequent RDF types in the sample I of DBpedia resources.

$D := \emptyset$. For each resource $s \in I$ do: Extract the subgraph G_s of R which contains s and all direct neighbors of s. Depending on the direction of the edges, there are three ways to extract the direct neighbors of s:

- **in** - consider only neighbors which are reachable by an incoming edge;
- **out** - consider only neighbors which are reachable by an outgoing edge;
- **all** - consider all neighbors of s.

Then create a record $r_s = (G_s, T_s)$ where T_s is the set of target labels (i.e., RDF types) of s. Add r_s to D.

The most frequent RDF types[9] appearing in the sample I which is used for the experiments in this section are shown on Table 5.5. In total the RDF types of the resources in I are 213. The average sizes and standard deviation of the sizes of the graphs in the graph database D built with respect to I using the previously described method are shown on Table 5.6. It can be seen by the standard deviation that some graphs are much larger than others. The tree-width of all graphs is equal to 1 because all of them are star graphs.

The multi-label classification of the resources is performed using a one-versus-rest learning strategy. This means that the multi-label classification problem is divided into multiple concept learning problems and a separate SVM classifier is trained for each of the classes that appear in the database. For all F1, precision, and recall scores listed in this section a *micro* averaging among all classes is used.

[9]The used URI prefixes are defined in: `http://dbpedia.org/sparql?nsdecl`

Neighborhood type	Average number of nodes	Average number of edges
in	143.845 (±756.677)	159.743 (±843.983)
out	34.274 (±43.977)	37.573 (±52.099)
all	174.661 (±758.983)	197.075 (±847.705)

Table 5.6: Number of nodes and edges of the neighborhoods of the extracted sample I of DBpedia resources.

Neighborhood type	w	h	SVM kernel	F1	Precision	Recall	Accuracy
in	5	1	RBF	.895(.005)	.916(.010)	.875(.007)	.682(.012)
in	5	0	linear	.887(.006)	.917(.009)	.859(.008)	.669(.013)
out	5	0	linear	.970(.003)	.988(.002)	.952(.005)	.890(.009)
all	5	0	linear	.967(.003)	.989(.002)	.946(.006)	.874(.012)

Table 5.7: Results for multi-label classification of DBpedia resources.

The results from the DBpedia experiments are shown on Table 5.7. Having in mind that the extracted sample of resources has in total over 200 RDF types, it takes long time to train the models multiple times when doing grid search for the best parameters. That is why for neighborhood types **out** and **all** only an SVM with linear kernel is used, which is trained much faster than with RBF kernel and has one parameter less to adjust. Even though the RBF kernel is more expressive, the results using a linear kernel still show that the feature extraction method performs well on this task. It should be mentioned that the types of resources in the DBpedia dataset are generated in the same step as the outgoing properties [6]. This makes the inference of RDF types from outgoing properties much easier than from incoming properties. However, it is still interesting to see that the feature extraction method preserves the relevant information in the mapping to binary vectors in all cases, which shows, once again, that the mapping is not arbitrary. The results achieved by the RDF specific method used by [6], called *SDType*, are still much better than the results listed in this section. However, keeping in mind that the method presented here is applicable to general graphs, it is important to note that type inference of RDF resources is one of the application fields in which the method performs well and can be used as an alternative to RDF specific methods.

Chapter 6

Summary

In this work we proposed a fast feature extraction method for graphs of tree-width at most 3. The method is based on the algorithm of Arnborg and Proskurowski [1] for computing the canonical string representations of partial 3-trees. After computing the canonical string representation, we extract the features using w-shingling [7]. We refine the vertex labels of the input graph using the Weisfeiler-Lehman method [8], in order to improve the quality of the mapping. The extracted feature vectors can be computed in linear time and used to solve machine learning and data mining problems on graph structured data. The proposed method can be extended to graph kernels using, for example, linear or Gaussian kernels. The mapping is not necessarily injective modulo graph isomorphism.

We discussed how to implement the algorithm of Arnborg and Proskurowski and the Weisfeiler-Lehman method in order to perform the feature extraction in linear time. In addition, we discussed how to turn a large RDF graph into a graph database suitable for the proposed method.

The usefulness of the feature extraction approach was evaluated by performing experiments on chemical datasets and DBpedia. The results achieved on the NCI-HIV dataset showed that the feature extraction method indeed gives a meaningful mapping with respect to graph similarity. From the experiments on the Mutagenesis dataset it can be seen that the approach performs similarly well on both ordinary graphs and RDF data. The results on the Carcinogenesis RDF dataset show that the preprocessing of the RDF data has strong influence on the classification performance for different learning problems. Combining the results from the three datasets, we can conclude that the proposed method performs well on classifying chemical compounds.

We also performed experiments for type inference of DBpedia resources using the feature extraction method. The results of these experiments are worse than the ones achieved by the SDType method [6], however, they show

that our feature extraction approach is still applicable for this task.

For future work, we can propose the following ideas. It was shown that preprocessing can significantly impact the performance of the method. The used preprocessing techniques Pre-1 and Pre-2, defined in Section 5.1, were designed without a comprehensive analysis of their specific advantages. The use of a well designed preprocessing method may bring large improvement on the results of the feature extraction method. It is important to develop such a preprocessing technique in future.

The restriction to work with partial 3-trees is limiting the applicability of the method. We are currently unaware of any correct linear time algorithm for computing the canonical string representations of graphs of tree-width larger than 3. However, it may be appropriate to use a linear time algorithm which computes a semi-canonical string representations of the graphs of larger tree-width, if such an algorithm exists. Even if the string representation of a graph is not completely canonical, it may still bring useful information about the structure of the graph. Therefore, the extracted features from such a string representation may be useful for solving machine learning and data mining problems on graphs.

In addition, conducting more experiments on different learning problems can bring better insight to how useful is the mapping in general. It can be also interesting to compare the performance of the method to more graph kernels on specific problems for learning in chemical data and RDF.

Bibliography

[1] Arnborg, S., Proskurowski, A.: Canonical representations of partial 2- and 3-trees. BIT Numerical Mathematics **32**(2) (June 1992) 197–214

[2] Kondor, R.I., Lafferty, J.D.: Diffusion kernels on graphs and other discrete input spaces. In: Proceedings of the Nineteenth International Conference on Machine Learning. ICML '02, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2002) 315–322

[3] Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory. COLT '92, New York, NY, USA, ACM (1992) 144–152

[4] Hofmann, T., Schölkopf, B., Smola, A.J.: Kernel methods in machine learning. Annals of Statistics **36** (2008) 1171–1220

[5] Gärtner, T., Flach, P., Wrobel, S.: On graph kernels: Hardness results and efficient alternatives. In: Learning Theory and Kernel Machines. Springer Berlin Heidelberg (2003) 129–143

[6] Paulheim, H., Bizer, C. In: Type Inference on Noisy RDF Data. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 510–525

[7] Broder, A.Z.: Identifying and filtering near-duplicate documents. In: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching. COM '00, London, UK, UK, Springer-Verlag (2000) 1–10

[8] Weisfeiler, B., Lehman, A.A.: A reduction of a graph to a canonical form and an algebra arising during this reduction (in Russian). Nauchno-Technicheskaya Informatsia (1968)

[9] Arnborg, S., Proskurowski, A.: Characterization and recognition of partial 3-trees. SIAM J. Algebraic Discrete Methods **7**(2) (April 1986) 305–314

[10] Horváth, T., Ramon, J., Wrobel, S.: Frequent subgraph mining in outerplanar graphs. Data Mining and Knowledge Discovery **21**(3) (2010) 472–508

[11] Horváth, T., Gärtner, T., Wrobel, S.: Cyclic pattern kernels for predictive graph mining. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '04, New York, NY, USA, ACM (2004) 158–167

[12] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - a crystallization point for the web of data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(3) (2009) 154 – 165 The Web of Data.

[13] Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M.: Graph kernels. J. Mach. Learn. Res. **11** (August 2010) 1201–1242

[14] Ristoski, P., Paulheim, H.: Semantic web in data mining and knowledge discovery: A comprehensive survey. Web Semantics: Science, Services and Agents on the World Wide Web **36** (2016) 1 – 22

[15] Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. J. Mach. Learn. Res. **12** (November 2011) 2539–2561

[16] Harchaoui, Z., Bach, F.R.: Image classification with segmentation graph kernels. In: 2007 IEEE Conference on Computer Vision and Pattern Recognition. (June 2007) 1–8

[17] Suard, F., Guigue, V., Rakotomamonjy, A., Benshrair, A.: Pedestrian detection using stereo-vision and graph kernels. In: IEEE Proceedings. Intelligent Vehicles Symposium, 2005. (June 2005) 267–272

[18] Vert, J.P., Matsui, T., Satoh, S., Uchiyama, Y.: High-level feature extraction using SVM with walk-based graph kernel. In: 2009 IEEE International Conference on Acoustics, Speech and Signal Processing. (April 2009) 1121–1124

[19] Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P.: Extensions of marginalized graph kernels. In: Proceedings of the Twenty-first International Conference on Machine Learning. ICML '04, New York, NY, USA, ACM (2004) 70–

[20] Shervashidze, N., Vishwanathan, S.V.N., Petri, T., Mehlhorn, K., Borgwardt, K.: Efficient graphlet kernels for large graph comparison. In: JMLR Workshop and Conference Proceedings Volume 5: AISTATS 2009, Cambridge, MA, USA, Max-Planck-Gesellschaft, MIT Press (April 2009) 488–495

[21] Ramon, J., Gärtner, T.: Expressivity versus efficiency of graph kernels. In: Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences. (2003) 65–74

[22] Mahé, P., Vert, J.P.: Graph kernels based on tree patterns for molecules. Machine Learning **75**(1) (2009) 3–35

[23] Bach, F.R.: Graph kernels between point clouds. CoRR **abs/0712.3402** (2007)

[24] Horváth, T. In: Cyclic Pattern Kernels Revisited. Springer Berlin Heidelberg, Berlin, Heidelberg (2005) 791–801

[25] Lösch, U., Bloehdorn, S., Rettinger, A. In: Graph Kernels for RDF Data. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 134–148

[26] de Vries, G.K.D.: A fast approximation of the Weisfeiler-Lehman graph kernel for RDF data. In: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 8188. ECML PKDD 2013, New York, NY, USA, Springer-Verlag New York, Inc. (2013) 606–621

[27] de Vries, G.K.D., de Rooij, S.: Substructure counting graph kernels for machine learning from RDF data. Web Semantics: Science, Services and Agents on the World Wide Web **35, Part 2** (2015) 71 – 84 Machine Learning and Data Mining for the Semantic Web (MLDMSW).

[28] Diestel, R.: Graph Theory, 4th Edition. Volume 173 of Graduate texts in mathematics. Springer (2012)

[29] Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)

[30] Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71, New York, NY, USA, ACM (1971) 151–158

[31] Robertson, N., Seymour, P.D.: Graph minors. III. planar tree-width. J. Comb. Theory, Ser. B **36**(1) (1984) 49–64

[32] Rose, D.J.: On simple characterizations of k-trees. Discrete Mathematics **7**(3) (1974) 317 – 322

[33] Scheffler, P.: Linear-time algorithms for NP-complete problems restricted to partial k-trees. Report (Karl-Weierstrass-Institut für Mathematik). Akademie der Wissenschaften der DDR, Karl-Weierstrass-Institut für Mathematik (1987)

[34] Bodlaender, H.L.: Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. In: SWAT 88: 1st Scandinavian Workshop on Algorithm Theory Halmstad, Sweden, July 5–8, 1988 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (1988) 223–232

[35] Bodlaender, H.L.: A linear time algorithm for finding tree-decompositions of small treewidth. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing. STOC '93, New York, NY, USA, ACM (1993) 226–234

[36] Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods **8**(2) (April 1987) 277–284

[37] Cai, J.Y., Fürer, M., Immerman, N.: An optimal lower bound on the number of variables for graph identification. Combinatorica **12**(4) (1992) 389–410

[38] Evdokimov, S.A., Ponomarenko, I.N.: On high closed cellular algebras and high closed isomorphisms. The Electronic Journal of Combinatorics **6** (1999)

[39] Babai, L., Kucera, L.: Canonical labelling of graphs in linear average time. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science. SFCS '79, Washington, DC, USA, IEEE Computer Society (1979) 39–46

[40] Douglas, B.L.: The Weisfeiler-Lehman Method and Graph Isomorphism Testing. ArXiv e-prints (January 2011)

[41] Rabin, M.O.: Fingerprinting by Random Polynomials. Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. (1981)

[42] Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences 1997. SEQUENCES '97, Washington, DC, USA, IEEE Computer Society (1997) 21–

[43] Debnath, A.K., de Compadre, R.L.L., Debnath, G., Shusterman, A.J., Hansch, C.: Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. Journal of Medicinal Chemistry **34**(2) (1991) 786–797

[44] Lehmann, J.: DL-Learner: Learning concepts in description logics. J. Mach. Learn. Res. **10** (December 2009) 2639–2642

[45] Costa, F., Grave, K.D.: Fast neighborhood subgraph pairwise distance kernel. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel. (2010) 255–262

[46] Lodhi, H., Muggleton, S.: Is mutagenesis still challenging? In: Proceedings of the 15th International Conference on Inductive Logic Programming, ILP 2005, Late-Breaking Papers. (2005) 35–40. (2005) 35–40