KU LEUVEN
ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

UNIVERSITÀ DEGLI STUDI FIRENZE

Learning from Structured Data with Kernels, Neural Networks and Logic

Francesco Orsini

Supervisors:
Prof. dr. Luc De Raedt
Prof. dr. ing. Paolo Frasconi

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2017

Learning from Structured Data with Kernels, Neural Networks and Logic

Francesco ORSINI

Examination committee:
Prof. dr. Patrick Wollants, chair
Prof. dr. Luc De Raedt, supervisor
Prof. dr. ing. Paolo Frasconi, supervisor
Prof. dr. ir. Johan Suykens
Prof. dr. Jan Ramon

Prof. dr. Alessandro Sperduti
(University of Padova)
Prof. dr. Thomas Gärtner
(University of Nottingham)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2017

Abstract

Real-world data often have a complex structure that can be naturally represented with graphs or logic. This thesis has four main contributions that address some challenges that arise when learning from structured data.

First, we introduce *graph invariant kernels* (GIKs), a framework that upgrades the Weisfeiler-Lehman and other graph kernels to effectively exploit high-dimensional and continuous vertex attributes. Graphs are first decomposed into subgraphs. Vertices of the subgraphs are then compared by a kernel that combines the similarity of their labels and the similarity of their structural role using a suitable vertex invariant. By changing this invariant we obtain a family of graph kernels that includes generalizations of Weisfeiler-Lehman, NSPDK, and propagation kernels. We demonstrate empirically that these kernels obtain state-of-the-art results on relational datasets.

Second, we introduce *shift aggregate extract networks* (SAEN) an architecture based on deep hierarchical decompositions to learn effective representations of large graphs. Our framework extends classic $\mathcal{R}$-decompositions used in kernel methods, enabling nested *part-of-part* relations. Unlike recursive neural networks, which unroll a template on input graphs directly, we unroll a neural network template over the decomposition hierarchy, allowing us to deal with the high degree variability that characterize social network graphs. Deep hierarchical decompositions are also amenable to domain compression, a technique that reduces both space and time complexity by exploiting symmetries. We show empirically that our approach is competitive with current state-of-the-art graph classification methods, particularly when dealing with social network datasets.

Third, we introduce kProbLog as a declarative logical language for machine learning. kProbLog is a simple algebraic extension of Prolog with facts and rules annotated by semiring labels. It allows to elegantly combine algebraic expressions with logic programs. We introduce the semantics of kProbLog, its inference algorithm, its implementation and provide convergence guarantees for a fragment of the language. We provide several code examples to illustrate its potential for a wide range of machine learning techniques. In particular, we show the encodings of state-of-the-art graph

kernels such as Weisfeiler-Lehman graph kernels, propagation kernels and an instance of GIKs. However, kProbLog is not limited to kernel methods and it can concisely express declarative formulations of tensor-based algorithms such as matrix factorization and energy-based models, and it can exploit semirings of dual numbers to perform automatic differentiation. Furthermore, experiments show that kProbLog is not only of theoretical interest, but can also be applied to real-world datasets. At the technical level, kProbLog extends aProbLog (an algebraic Prolog) by allowing multiple semirings to coexist in a single program and by introducing meta-functions for manipulating algebraic values.

Fourth, we provide a mathematical analysis of the preimage problem of the Weisfeiler-Lehman subtree kernel and show how to *morph* new graphs from graph datasets with a simple technique that employs off-the-shelf solvers. Preliminary results show that this technique is amenable for future constructive machine learning applications such as *de novo* synthesis of small molecules.

Beknopte samenvatting

In de praktijk gebruikte data hebben vaak een complexe structuur en kunnen natuurlijk worden voorgesteld door middel van grafen of logica. Deze thesis omvat vier voornamelijke contributies die aan sommige van de uitdagingen tegemoetkomen die optreden tijdens het leren vanuit gestructureerde data.

Ten eerste introduceren we graph invariant kernels (GIKs), een framework dat Weisfeiler-Lehman en andere graph kernels uitbreidt om efficiënt om te gaan met hoogdimensionale en continue attributen van knopen. Grafen worden eerst ontleed in kleinere subgrafen. De knopen van die subgrafen worden vergeleken met behulp van een kernel die de gelijkenis van hun labels en de gelijkenis van hun structurele rol combineert doormiddel van een geschikte knoop-invariant. Door deze invariant aan te passen verkrijgen we een familie van graph kernels die veralgemeningen van Weisfeiler-Lehman, NSPDK en propagation kernels bevat. Doormiddel van experimenten tonen we aan dat deze kernels state-of-the-art resultaten behalen op relationele datasets.

Ten tweede introduceren we shift aggregate extract networks (SAEN), een op diepe hiërarchische decomposities gebaseerde architectuur voor het leren van effectieve voorstellingen van grote grafen. Ons framework veralgemeendt klassieke $\mathcal{R}$-decomposities die in kernel methodes gebruikt worden en laat geneste *part-of-part* relaties toe. Terwijl recursieve neurale netten templates direct ontplooien op de invoer grafen, ontplooien wij een neuraal net template over de decompositie hiërarchie. Dit stelt ons in staat om beter om te gaan met de erg scheve verdeling van de graden van knopen, die karakteristiek is voor grafen van sociale netwerken. Diepe hiërarchische decomposities zijn ook vatbaar voor *domain compression*, een techniek die de tijd- en ruimte complexiteit verlaagt door symmetrieën uit te buiten. We tonen empirisch aan dat onze aanpak goed scoort in vergelijking met state-of-the-art technieken voor de classificatie van grafen, vooral voor sociale netwerk grafen.

Ten derde introduceren we kProbLog, een declaratieve logische taal voor machinaal leren. kProbLog is een simpele, algebraïsche uitbreiding van Prolog waarin feiten en regels geannoteerd worden door semiring labels. kProbLog combineert algebraïsche uitdrukkingen met logische programma's op een elegante manier. In deze thesis

stellen wij de semantiek van kProbLog voor, zijn inferentie-algoritme en voor een fragment van de taal convergentiegaranties. Met behulp van verschillende code voorbeelden illustreren wij de toepasbaarheid van deze taal voor een breed gamma aan machine learning technieken. In het bijzonder tonen wij hoe state-of-the-art graph kernels zoals Weisfeiler-Lehman graph kernels, propagation kernels en een voorbeeld van GIKs geëncodeerd kunnen worden. kProbLog is echter niet beperkt tot kernel methodes en kan declaratieve formuleringen van op tensoren gebaseerde algoritmes zoals matrixfactorisatie en *energy-based* modellen beknopt uitdrukken. Verder kan kProbLog de semiring van de duale getallen uitbuiten om automatische differentiatie uit te voeren. Experimenten tonen bovendien aan dat kProbLog niet enkel interessant is op theoretisch vlak, maar ook kan toegepast worden op real-world datasets. Op technisch vlak is kProbLog een uitbreiding van aProbLog (een algebraïsche versie van Prolog). kProbLog laat toe om verschillende semiringen binnen eenzelfde programma te gebruiken en introduceert de notie van meta-functies voor het manipuleren van algebraïsche waarden.

Ten vierde verstrekken we een wiskundige analyse van het preimage probleem van de Weisfeiler-Lehman subtree kernel en tonen wij aan hoe nieuwe grafen door *graph morphing* gegenereerd kunnen worden uit datasets van grafen met behulp van een simpele techniek die gebruikmaakt van bestaande solvers. Voorlopige resultaten bevestigen dat deze techniek toepasbaar is op toekomstige constructieve machine learning toepassingen, zoals *de novo* synthese van kleine moleculen.

Abstract in italiano

I dati del mondo reale spesso presentano una struttura complessa che può essere naturalmente rappresentata con grafi o fatti logici. Questa tesi presenta quattro contributi principali che affrontano alcune sfide dell'apprendimento automatico da dati strutturati.

I) Introduciamo i *graph invariant kernel*[1] (GIK), un framework che aggiorna il Weisfeiler-Lehman ed altri kernel su grafi al fine di gestire efficacemente vertici con attributi ad alta dimensionalità e continui. I grafi sono prima decomposti in sottografi, i vertici dei sottografi sono quindi comparati tramite un kernel che combina la somiglianza trai loro attributi e la somiglianza trai loro ruoli strutturali, utilizzando un'apposita invariante sui vertici. Cambiando tale invariante otteniamo una famiglia di kernel sui grafi che include generalizzazioni dei kernel Weisfeiler-Lehman, NSPDK e di propagazione. Dimostriamo empiricamente che i kernel proposti ottengono risultati allo stato dell'arte su dati relazionali.

II) Introduciamo *shift aggregate extract networks*[2] (SAEN), un'architettura basata su decomposizioni gerarchiche profonde allo scopo di apprendere rappresentazioni efficaci per grafi di considerevoli dimensioni. Il nostro framework estende le classiche $\mathcal{R}$-decomposizioni utilizzate dai metodi kernel, permettendo di annidare relazioni *parte-di-parte*. Diversamente dalle reti neurali ricorsive, che espandono una rete neurale direttamente sui grafi in ingresso, noi espandiamo una rete neurale sulla decomposizione gerarchica. Questo ci permette di trattare grafi con grado fortemente variabile che tipicamente caratterizza le reti sociali. Le decomposizioni gerarchiche profonde si prestano anche alla compressione di dominio, una tecnica che riduce la complessità sia spaziale che temporale dell'addestramento attraverso lo sfruttamento delle simmetrie. Mostriamo empiricamente che il nostro approccio è competitivo con lo stato dell'arte nella classificazione di grafi. In particolare SAEN ha un'eccellente performance sui grafi sociali.

III) Introduciamo kProbLog come linguaggio logico dichiarativo per l'apprendimento

[1] NdT Letteralmente *kernel su invarianti di grafo.*

[2] NdT Letteralmente *reti di spostamento, aggregazione ed estrazione.*

automatico. kProbLog è una semplice estensione algebrica del linguaggio Prolog con fatti e regole annotati con label algebriche. Questi permette di combinare elegantemente espressioni algebriche con programmi logici. Introduciamo la semantica di kProbLog, il suo algoritmo di inferenza, la sua implementazione e forniamo garanzia di convergenza. Proponiamo svariati esempi di codice al fine di illustrare il suo potenziale per una vasta gamma di tecniche di apprendimento automatico. In particolare, mostriamo le specifiche di kernel allo stato dell'arte come il Weisfeiler-Lehman graph kernel, i kernel di propagazione ed un'istanza dei GIK. Tuttavia, kProbLog non è limitato ai metodi kernel e può esprimere in maniera coincisa formulazioni dichiarative di algoritmi basati sui tensori come la fattorizzazione di matrici utilizzando il semianello dei numeri duali per la differenziazione automatica. Inoltre, alcuni esperimenti mostrano che kProbLog non è un linguaggio di puro interesse teorico, ma può anche essere applicato a dati provenienti dal mondo reale. Da un punto di vista tecnico, kProbLog estende aProbLog (un Prolog algebrico) permettendo a molteplici semianelli di coesistere in un solo programma ed introducendo le meta-funzioni per manipolare valori algebrici.

IV) Proponiamo un'analisi matematica del problema della preimmagine del kernel sui sottoalberi di Weisfeiler-Lehman e mostriamo come fare il *morph* di nuovi grafi per mezzo di una semplice tecnica che utilizza solver esistenti. Risultati preliminari mostrano che questa tecnica è promettente per applicazioni di apprendimento automatico costruttivo come la sintesi *de novo* di piccole molecole.

Acknowledgements

Doing a Ph.D. is a challenging experience and achieving this goal would not have been possible without the help and support of many people. I wish to thank all of them for their valuable support.

It all started in Florence when I knocked to the office of professor Paolo Frasconi and I asked him for a contact for my Erasmus exchange. I probably had a very bad timing because I interrupted his meeting with professor Luc De Raedt who was visiting the University of Florence in those days. However, I also had a good timing since Paolo asked Luc if he wanted me as his student and he accepted. That is how I started my Belgian experience which continued after the master degree with a Ph.D. under the supervision of Luc and Paolo. I am very thankful to my promotors for the chances and the freedom that they gave me in research, allowing me to follow my own ideas.

I am thankful to all the members of my jury, Jan Ramon, Johan Suykens, Thomas Gaertner and Alessandro Sperduti for carefully reading the text of this thesis and helping me to improve it with their comments. I thank Patrick Wollants for chairing the jury.

During these years, I had the chance to benefit from knowledge and the experience of many people in the machine learning group.

I wish to thank Angelika for the enlightening conversations about the ProbLog language, Anton for the conversations about the implementation of the ProbLog language, Wannes for the advise that he gave when I was organizing the ML meetings, Sergey for sharing his deep knowledge about *answer set programming*, Monty Pythons and the Simpsons, Samuel for being an enthusiastic student of the Italian language and for translating the English abstract of this thesis in Dutch, Joris for teaching me how to teach UAI exercise sessions, Pedro for teaching UAI exercise sessions with me, Gust for sharing with me his technical ideas about program synthesis, Jonas that made me understand that the T_P-operator was more interesting than what I had expected at the beginning, Toon for our conversations about clustering and automated machine learning, Dries for giving me interesting insights about gene networks. A big thank to Jessa, Sebastijan

and Tom for all the content proposals that they made for the ML meetings.

Behrouz, Daniele B., Davide, Stefano T., Laura and Vladimir you have been good friends and I also had great technical conversations with you all. Thank you for that.

I also want to thank my friends Emanuele, Marta & PK, Elisabetta & Nitin, Maria Elena & Souradip, Lorenzo, Anna Lisa and Elisa that I met in Leuven and the friends Tiberio, Stefano D. and Enrico that I have been missing since I left Florence.

I am thankful to my family, to my grandfather Paolo and my grandmother Nada who were my first teachers, to grandmother Maresca who bought my first computer, to my mother Laura that since I was a kid taught me how to turn my homework into small research activities, to my father Alessio that always encouraged me to buy books.

Finally, I want to say a big thank to my wife Yasamin for supporting me during all these years and for being the mother of our future son.

Francesco Orsini
Leuven, Belgium
June 2017

List of Abbreviations

ADME absorption, distribution, metabolism, and excretion. 101

AI artificial intelligence. 1, 3

AMC algebraic model counting. 87, 96

AMND average maximum node degree. 49, 53

ANN artificial neural networks. 2, 3

BDD binary decision diagram. 125

CNGK color neighborhood graph kernel. 102, 104, 110

CNN convolutional neural network. 2, 3, 36, 53, 55, 125

CN color neighborhood. 104, 105, 110

CSP constraint satisfaction problem. 102, 111

DAG directed acyclic graph. 2, 125

DGK deep graph kernels. 36, 50, 53–55

DNNF decomposable negation normal form. 125

DSL domain-specific language. 127

EGNN ego graph network. 44, 49–52, 124

GED graph edit distance. 100, 118, 119

GIK graph invariant kernel. i–vi, 4, 5, 11, 18, 19, 22, 27–29, 31–33, 78, 79, 97, 123, 124, 127

GK graph kernels. 35, 55

GSGK global spectral graph kernel. 27, 29

GWL global Weisfeiler-Lehman. 27, 29–32

LSGK local spectral graph kernel. 27, 29

LWL local Weisfeiler-Lehman. 27, 29–32, 79

MIL meta-interpretative learning. 126, 127

MKLG marginalized kernels between labeled graphs. 100, 104, 105, 118, 119

NN4G neural networks for graphs. 35, 54

NSK neighborhood subgraph kernel. 27–29, 31

NSPDK neighborhood subgraph pairwise distance kernel. i, iii, v, 4, 20, 24, 32, 118, 119

OBDD ordered binary decision diagram. 87

PGM probabilistic graphical model. 120, 121

PROP propagation kernels. 23, 30, 32

PSD positive semidefinite. 8, 9, 22

QCQP quadratically constrained quadratic programing. 102, 109, 119, 121

QP quadratic programing. 109, 113, 119, 121

RBF radial basis functions. 2

RNN recurrent/recursive neural networks. 2, 3, 35, 36, 54, 125

RW random walks. 17

ReLU rectified linear unit. 14, 51

SAEN shift aggregate extract network. i, iii, v, 4, 5, 11, 36, 38, 40, 41, 44, 46, 48, 50, 51, 53–55, 124–127

SDD sentential decision diagrams. 67, 86–88, 96, 125

SP shortest paths. 17, 20

SVM support vector machines. 13, 14, 27, 30, 92, 94, 102, 119, 121

WLST Weisfelier-Lehman subtree. 100–104, 110–112, 114–122, 127

WL Weisfeiler Lehman. 22, 23, 27, 29–32, 76, 77, 102, 103, 110–112, 120, 121

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years we witnessed a huge increase of the volume of available data and computational power. Social networks are populated by millions of users that interact forming a number of connections which is even larger. Mobile devices allow users to access the internet $24/7$ and publish content such as photos, videos, tweets and blog posts. Often we refer to these large datasets as big data. Hiring enough humans beings to make sense of such a huge volume of data would be impractical. For this reason machines can not limit themselves to load, store and retransmit data, but should make sense of them and exhibit intelligent behavior.

In bio-/chemo-informatics we find completely different kinds of challenges such as *de novo* synthesis of small molecules that has applications to perhaps drug discovery. In this setting data collection requires domain experts and its availability is low compared to the very large space of the possible small molecules. Machines can be trained on data to decide whether or not examples have the desired properties and through the use of combinatorial optimization is possible to search in such a huge space for new molecules that satisfy the desired properties.

1.1 Machine Learning

Machine learning is the subfield of AI that groups together all the methodologies that allow to write software that can learn from data and improve its performance automatically. More than one school of thought provided different answers on how machines should be built in order to learn from data. The schools of thought that we shall consider are the one of learning with separators and the one of logic. While the

former conceives learning as a numerical optimization problem the latter regards it as the inverse of logical deduction.

According to the school of learning with separators, inputs are mapped to vectors and then semantic concepts are discriminated using hyperplanes. The two main exponents of the school of learning with separators are kernel methods and neural networks. The main difference between kernel methods and neural networks relies on how features are built. While in neural network features are made by stacking tunable layers of features, in kernel methods the feature space is defined by the kernel which in turn may be learnt.

In this dissertation we will focus on learning from structured domains such as social networks, networks of proteins, molecules and natural-language sentences.

1.1.1 Features and structure

Vectors of attributes provide a natural representation for the input of neural networks and also for kernel methods. Kernel methods define a similarity measure between inputs (i.e. a kernel) and most of the times this definition relies on standard choices such as the linear, the polynomial and radial-basis-function (RBF) kernels. However, learning from structured input data such as sequences, trees or graphs is less trivial and has received different answers in both the kernel and the neural network literature.

In 1999 Haussler unified existing kernel methods for structured data under the framework of convolution kernels. Since then we assisted to a huge number of works that expanded and extended that framework. Convolution kernels employ *part-of* relations to decompose structured inputs into sets of parts and define the similarity measure as the count on common parts. Haussler's framework [Haussler, 1999] is very general and only few graph kernels do not fit in it (e.g. [Riesen and Bunke, 2009]). However, the kind of relational information that needs to be captured may change a lot from dataset to dataset and the choice of the decomposition relation is crucial in order to achieve good generalization performance. Another interesting problem in Haussler's kernels arises when inputs are represented as graphs with continuous attributes and we need to combine together the discrete structure of the graphs together with the continuous attributes that are possibly high dimensional.

Neural networks on structured data embrace a different point of view from the one of graph kernels. Rather than decomposing the input structure in subparts they unfold the neural network computation over the input structure. The principal example are *recurrent neural networks* for sequences and *recursive neural networks* (RNNs) which generalize recurrent neural networks to directed acyclic graphs (DAGs) [Frasconi et al., 1998]. Convolutional neural networks (CNNs) are another popular ANN architecture

specific to images.[1] CNNs where inspired by the animal visual cortex and differ from RNNs in the way they capture the structure of the inputs. The input images are decomposed into overlapping square windows that mimic biological receptive fields. A feature is then extracted from each square window applying artificial neurons with shared weights. Pixel adjacency in CNN's receptive fields is a very simple structural pattern, but this was no limitation to the success of these architectures in computer vision [Krizhevsky et al., 2012]. While Haussler's kernels allow to choose an appropriate decomposition relation that better captures the signal in the input data, RNNs and CNNs are mostly tailored for input data with a specific structure. Perhaps RNNs were mostly used for sequence data and parse trees in natural language processing, while CNNs are specific to the pixel adjacency relations in images. However, the application of graph kernels and ANNs to social network data started only recently [Perozzi et al., 2014, Yanardag and Vishwanathan, 2015, Niepert et al., 2016].

Adapting in kernel methods and ANNs to the structure of the input data requires a certain degree of technical expertise that might be beyond the reach of domain experts. A fundamental role in bridging the gap between problem specification and solving techniques is played by *declarative languages*.

1.1.2 Declarative languages

Declarative languages and imperative programming languages are in sharp contrast. While the aim of the former is to allow the user to focus on the problem specification (i.e. *what is the problem*), the latter focusses on the control flow of the solution (i.e. *how to solve the problem*).

One of the most elegant formalisms for the formulation of declarative specifications is logic. The biggest strength of logic as declarative language for machine learning is *interpretability*. Relational data are naturally represented by logical facts and background knowledge can easily be added by the user specifying logical rules. As we mentioned earlier in the introduction, learning in the logical setting is equivalent to inverting the deductive process by inducing the rules that allows the conclusions to follow from the premises. A considerable advantage of these kinds of systems is that they can provide *explanations* for the decisions they make.

While in the 80s and 90s, logic based systems were dominating AI, purely logical systems were found hard to update when rules were specified manually, and non-scalable and non-robust to noise when rules were learnt.

The machine learning research in the following years either focused on other techniques such as kernel methods and neural networks or invested in hybrid systems that could

[1] CNNs where also applied to 1-dimensional signals

handle uncertainty by integrating logic with statistics and/or probability.

Today there exist many frameworks and formalisms that tightly integrate these two paradigms; they support probabilistic and logical inference as well as learning. Prominent examples include PRISM [Sato and Kameya, 1997], Dyna [Eisner et al., 2004, Eisner and Filardo, 2011], Markov Logic [Richardson and Domingos, 2006], BLOG [Milch et al., 2005], and ProbLog [De Raedt et al., 2007]. While there has been a lot of research on integrating probabilistic and logic reasoning, the combination of kernel-based methods with logic has been much less investigated with the notable exceptions of kLog [Frasconi et al., 2014], kFOIL [Landwehr et al., 2006] and Gärtner et al.'s work [Gärtner et al., 2003, 2004]. kLog is a relational language for specifying kernel-based learning problems. It produces a graph representation of a relational learning problem in the spirit of knowledge-based model construction and then employs a graph kernel on the resulting representation. kFOIL is a variation on the rule learner FOIL [Quinlan, 1990] that can learn kernels defined as the number of clauses that succeed in both interpretations. Gärtner et al. developed kernels within a typed higher order language and used it on some inductive logic programming benchmarks.

1.2 Contribution

This thesis makes three contributions to machine learning proposing kernels and neural networks to learn with structured data and a declarative language for kernel programming.

The **first contribution** introduces *graph invariant kernels* (GIKs) [Orsini et al., 2015a], a novel family of kernels that upgrades the Weisfeiler-Lehman and other graph kernels to effectively exploit high-dimensional and continuous vertex attributes. Graphs are first decomposed into subgraphs. Vertices of the subgraphs are then compared by a kernel that combines the similarity of their labels and the similarity of their structural role, using a suitable vertex invariant. By changing this invariant we obtain a family of graph kernels which includes generalizations of Weisfeiler-Lehman [Shervashidze et al., 2011], NSPDK [Costa and De Grave, 2010], and propagation kernels [Neumann et al., 2012a]. We demonstrate empirically that these kernels obtain state-of-the-art results on relational datasets.

The **second contribution** proposes *shift aggregate extract networks* (SAEN) [Orsini et al., 2017], an architecture based on deep hierarchical decompositions that can learn effective representations of large graphs. Our framework extends classic $\mathcal{R}$-decompositions used in kernel methods, enabling nested *part-of-part* relations. Unlike recursive neural networks, which unroll a template on input graphs directly, we unroll a neural network template over the decomposition hierarchy, allowing us to deal with the high degree variability that typically characterize social network graphs. Deep

hierarchical decompositions are also amenable to domain compression, a technique that reduces both space and time complexity by exploiting symmetries. We show empirically that our approach is competitive with state-of-the-art graph classification methods, particularly when dealing with social network datasets.

The **third contribution** is kProbLog, a declarative logical language for machine learning [Orsini et al., 2015b]. kProbLog is a simple algebraic extension of Prolog with facts and rules annotated by semiring labels. It allows to elegantly combine algebraic expressions with logic programs. We introduce the semantics of kProbLog, its inference algorithm, its implementation. We provide several code examples to illustrate its potential for a wide range of machine learning techniques. In particular, we show the encodings of state-of-the-art graph kernels such as Weisfeiler-Lehman graph kernels, propagation kernels and an instance of GIKs. However, kProbLog is not limited to kernel methods and it can concisely express declarative formulations of tensor-based algorithms such as matrix factorization and energy-based models, and it can exploit semirings of dual numbers to perform automatic differentiation. Furthermore, experiments show that kProbLog is not only of theoretical interest, but can also be applied to real-world datasets.

At the technical level, kProbLog extends aProbLog [Kimmig et al., 2011] (an algebraic Prolog) by allowing multiple semirings to coexist in a single program and by introducing meta-functions for manipulating algebraic values.

1.3 Thesis roadmap

Chapter 2 introduces some preliminary concepts on kernel methods and neural networks.

Chapter 3 introduces GIKs and shows how to upgrade existing and novel kernels to continuous values and an empirical evaluation achieving state-of-the-art performance on a number of datasets. The chapter consists of research previously published in:

- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. Graph Invariant Kernels. In: International Joint Conference on Artificial Intelligence. 2015. p. 3756-3762.

Chapter 4 presents SAEN, an architecture based on deep hierarchical decompositions to learn effective representations of large graphs. The paper is publicly available on `arxiv.org`.

- Orsini, Francesco; Baracchi Daniele; Frasconi, Paolo. Shift Aggregate Extract Networks. Preprint arXiv:1703.05537 [cs.LG] 2017.[2]

[2] `https://arxiv.org/pdf/1703.05537.pdf`

Chapter 5 introduces kProbLog: an algebraic Prolog for machine learning. The content of this chapter is part of a paper currently submitted to the Special issue on Inductive Logic Programming of the Machine Learning Journal. An earlier version of this work was published in:

- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. kProbLog: An Algebraic Prolog for Kernel Programming. In: International Conference on Inductive Logic Programming. Springer International Publishing, 2015. p. 152-165. **Best Student Paper, Machine Learning Journal Award.**

Chapter 6 consists of a mathematical analysis of the preimage problem of the Weisfeiler-Lehman subtree kernel and shows how to *morph* new graphs from graph datasets with a simple technique that employs off-the-shelf solvers. Preliminary results show that this technique is amenable for future constructive-machine-learning applications such as *de novo* synthesis of small molecules.

Chapter 7 concludes and discusses directions for the future work.

Chapter 2

Background

In this chapter we will introduce some preliminary machine learning concepts. While for a complete coverage of the concepts introduced in this chapter we recommend the textbooks [Scholkopf and Smola, 2001] and [Bishop, 2006], our goal will be to focus the attention of the reader on some aspects that will be used in the rest of this thesis.

We can machine learn from a set of examples $\{(x_i, y_i)\}_{i=1}^N \subset \mathcal{X} \times \mathcal{Y}$ inducing a function $f : \mathcal{X} \to \mathcal{Y}$ that maps each input $x \in \mathcal{X}$ to its target $y \in \mathcal{Y}$. In case we want to do regression, we set the target space $\mathcal{Y}$ to $\mathbb{R}$, while if we wish to do classification we set $\mathcal{Y}$ to a finite set of target classes $\mathcal{C}$.

As we anticipated in the introduction, the approach of learning with separators employs a feature function $\phi : \mathcal{X} \to \mathcal{H}$ and maps the inputs to a Hilbert space $\mathcal{H}$ (a Euclidean space when $\mathcal{H} = \mathbb{R}^d$) called feature space.

In order to classify examples we learn a classifier that partitions the feature space in decision regions R_c corresponding to the classes $c = 0, \ldots, |\mathcal{C}| - 1$. Once the classifier is learnt inputs x are classified (i.e. assigned to a class) by mapping them to a point $\phi(x)$ in the feature space $\mathbb{R}^d$ and then finding the class c such that its feature vector $\phi(x)$ belongs to the classification region $\mathcal{R}_c$ [Bishop, 2006].

Provided that the feature map ϕ is sufficiently expressive, classification regions can be distinguished with linear separators (i.e. hyperplanes in the feature space).

In the case of binary classification the decision function takes the form

$$f(x) = \begin{cases} 1 & \text{if } \langle \mathbf{w}, \phi(x) \rangle + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $\mathbf{w} \in \mathbb{R}^d$ is a vector of coefficients and b is a scalar such that when $b = 0$ the hyperplane passes through the origin $\mathbf{0}$.

A dataset made of points $\{(x_i, y_i)\}_{i=i}^N$ with class labels $y_i \in \{-1, 1\}$ is linearly separable w.r.t. a feature space induced by a feature map ϕ if there exists a hyperplane $\langle \mathbf{w}, \phi(x) \rangle + b$ such that $y_i(\langle \mathbf{w}, \phi(x_i) \rangle + b) > 0, \ \forall i = 1, \ldots, N$.

In kernel methods the the feature map ϕ usually maps the inputs to a high-dimensional feature space so that quite likely the dataset will be linearly separable. Neural networks on the other hand use a nonlinear feature map ϕ that is obtained by stacking multiple layers of linear separators whose output is transformed by an activation function which is non-linear.

In § 2.1 we will focus on two important aspects of kernel methods: one is how to build feature spaces both combining similarities (§ 2.1.1) and combining features (§ 2.1.2). In § 2.2 we will briefly go through perceptrons, support vector machines and neural networks explaining how to these machines are trained for classification.

2.1 Kernel methods

Kernel methods use as decision function the linear model of Eq. 2.1 and further assume that $\mathbf{w}$ lies in the span of the training examples and can be expressed as the linear combination:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i \phi(x_i). \tag{2.2}$$

If we substitute $\mathbf{w}$ from Eq. 2.2 into Eq 2.1 we will find the following decision function:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} \alpha_i k(x_i, x) + b \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

Where $k(x, z) = \langle \phi(x), \phi(z) \rangle$ is a *kernel* function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that encodes a similarity measure between its arguments x and z as their inner product $\langle \phi(x), \phi(z) \rangle$ in the feature space.

Definition 1. *A kernel $k(x, y)$ is positive semidefinite (*PSD*) if it is symmetric (i.e. $k(x, z) = k(z, x)\ \forall x, z$) and for any set $\{x_i\}_{i=1}^N$ of N examples we have that*

$$\sum_{i=1}^{N} \sum_{j=1}^{N} c_i c_j k(x_i, x_j) \geq 0 \tag{2.4}$$

for all $c_i \in \mathbb{R}$ [Scholkopf and Smola, 2001].

Definition 2. *Given a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ and a set of examples $\{x_1\}_{i=1}^N$ the gram matrix $K \in \mathbb{R}^{N \times N}$ has elements*

$$K_{i,j} = k(x_i, x_j), \ \forall i, j = 1, \ldots, N \tag{2.5}$$

Whenever a kernel $k(x, z)$ is PSD it admits an a feature map ϕ such that $k(x, z) = \langle \phi(x), \phi(z) \rangle$ called *reproducing kernel map*. The reproducing kernel map $\phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}$ is obtained by transforming each input $x \in \mathcal{X}$ into a function $f_x \in \mathbb{R}^{\mathcal{X}}$ where $\mathbb{R}^{\mathcal{X}} := \{\mathcal{X} \to \mathbb{R}\}$ [Scholkopf and Smola, 2001].

2.1.1 Combining kernels

Since if we can prove that if $k(x, z)$ is PSD then ϕ exists, we do not need to make ϕ explicit and we can just work on similarity measures. Indeed, starting from PSD kernels we combine them with operations which ensure that the result is PSD.

Let $k_1(x, z)$ and $k_2(x, z)$ two PSD kernels the kernels are also PSD (cf. Bishop [2006]):

1) $k(x, z) = ck_1(x, z)$ where $c \geq 0$,

2) $k(x, z) = f(x)k_1(x, z)f(z)$,

3) $k(x, z) = k_1(x, z) + k_2(x, z)$,

4) $k(x, z) = k_1(x, z)k_2(x, z)$,

5) $k(x, z) = p(k_1(x, z))$ where p is a polynomial with nonnegative coefficients,

6) $k(x, z) = e^{k_1(x,z)}$.

By combining **1)** and **3)** we can show that any linear combinations $\sum_i c_i k_i(x, z)$ of PSD kernels with nonnegative coefficients $c_i \geq 0$ are PSD. Convex combinations are a special case in which $\sum_i c_i = 1$.

Property **2)** can be used for kernel normalization perhaps we can define

$$f(x) = \frac{1}{\sqrt{k_1(x, x)}} \tag{2.6}$$

and obtain the PSD kernel

$$k(x, z) = \frac{k_1(x, z)}{\sqrt{k_1(x, x), k_1(z, z)}}. \tag{2.7}$$

Let us also notice that **1)** is a particular case of **2)** where $f(\mathbf{x}) = \sqrt{c}$.

The Gaussian kernel $k(\mathbf{x}, \mathbf{z}) = e^{-\gamma \|\mathbf{x}-\mathbf{z}\|_2^2}$ is easily derived starting from the linear kernel $\langle \mathbf{x}, \mathbf{z} \rangle$ which is PSD and using the following sequence of transformations:

$$\langle \mathbf{x}, \mathbf{z} \rangle \xrightarrow{\textbf{1) and } \gamma>0} 2\gamma\langle \mathbf{x}, \mathbf{z} \rangle \xrightarrow{\textbf{6)}} e^{2\gamma\langle \mathbf{x}, \mathbf{z} \rangle} \xrightarrow{\textbf{2) and } f(\mathbf{x})=\frac{1}{e^{\gamma\langle \mathbf{x}, \mathbf{x} \rangle}}} \frac{e^{2\gamma\langle \mathbf{x}, \mathbf{z} \rangle}}{e^{\gamma\langle \mathbf{x}, \mathbf{x} \rangle} e^{\gamma\langle \mathbf{z}, \mathbf{z} \rangle}} =$$

$$e^{-\gamma(\langle \mathbf{x},\mathbf{x}\rangle + \langle \mathbf{z},\mathbf{z}\rangle - 2\langle \mathbf{x},\mathbf{z}\rangle)} = e^{-\gamma\|\mathbf{x}-\mathbf{z}\|_2^2}$$

Property **4)** is reminiscent of a logical conjunction, perhaps we could say that two objects $\mathbf{x}$ and $\mathbf{z}$ are more similar the more they are similar w.r.t. to both k_1 and k_2. [1] This property will be used in chapter 3 to define kernels on graphs with continuous vertex attributes in order to combine structural similarity and continuous attribute similarity.

2.1.2 Combining features like a kernel designer

Kernel methods allow to design very expressive classes of functions for learning with linear separators and they are intuitive because they rely on the concept of similarity. However, a drawback of kernel methods is that they often require to store the Gram matrix whose space complexity is quadratic in the number of the examples. The space complexity can improve when the kernel admits a finite-dimensional feature map whose dimension is lower than the number of the examples.

In order to improve the space complexity in kernel design we start with elementary kernels $k_1(x,z)$ and $k_2(x,z)$ that admit finite-dimensional feature maps $\phi_1 : \mathcal{X} \to \mathbb{R}^{d_1}$ and $\phi_2 : \mathcal{X} \to \mathbb{R}^{d_2}$ respectively. Then we will find out which properties **1)**-**6)** lead to a kernel $k(x,z)$ that admits a finite-dimensional feature map ϕ.

For properties **1)** and **2)** is trivial to see that the feature $\phi_1(x)$ is rescaled by the multiplicative factors $\sqrt{c}$ and $f(x)$ respectively and the dimension of the feature space is not affected.

When using **3)** we obtain a kernel $k(x,z) = k_1(x,z) + k_2(x,z)$ that admits a finite-dimensional feature map $\phi : \mathcal{X} \to \mathbb{R}^{d_1+d_2}$ whose output can be expressed as the concatenations of the outputs of ϕ_1 and ϕ_2 (i.e. $\phi(x) = \phi_1(x)\|\phi_2(x)$). While when using **4)** we obtain a kernel $k(x,z) = k_1(x,z)k_2(x,z)$ that admits a finite-dimensional feature map $\phi : \mathcal{X} \to \mathbb{R}^{d_1 d_2}$ whose output can be expressed as the Kronecker product between the outputs of ϕ_1 and ϕ_2 (i.e. $\phi(x) = \phi_1(x) \otimes \phi_2(x)$).

The best known instance of **5)** is the polynomial kernel $k(\mathbf{x},\mathbf{z}) = \langle \mathbf{x},\mathbf{z}\rangle^d$ there the degree parameter d is a positive integer. $k(\mathbf{x},\mathbf{z})$ admits an explicit feature whose dimension is $\binom{d+N-1}{d} = \frac{(d+N-1)!}{d!(N-1)!}$ (cf. Scholkopf and Smola [2001]).

Property **6)** produces kernels that generally admit infinite-dimensional feature maps only. Using Taylor series we can write $e^{k(x,z)}$ as $\sum_{i=0}^{\infty} \frac{k(x,z)^i}{i!}$. Willing to derive an explicit feature for $e^{k(x,z)}$ we would need to concatenate an infinite number of polynomial features. However, when $k(x,z) < 1$ the higher order terms tend to 0 and

[1] As we shall see in chapter 5 the kProbLog language draws an analogy between logical connectives and

for $k(x, z)$ sufficiently small we can approximate $e^{k(x,z)}$ with a polynomial or even linear kernel. A similar consideration holds for the Gaussian kernel $e^{-\gamma\|\mathbf{x}-\mathbf{z}\|_2^2}$ when γ tends to 0.

The concepts explained so far are also useful for graph kernel design and will be used in chapter 3 where we introduce GIKs to combine the discrete structure of the graph together with continuos values and in chapter 5 where we introduce kProbLog as a language to combine logical and algebraic labels in order to declaratively specify kernels on relational data.

However, if the structure of the inputs is complex we need multiple kernel combinations and except for **1)** and **2)** all the other properties increase the dimensionality of the feature space. For GIKs we avoid the problem and just compute the kernel without using explicit features.

In chapter 4 we will introduce the SAEN neural network framework for learning with structured data. SAEN works on $\mathcal{H}$-decompositions which are a deep variant of Haussler's convolution kernels that decompose structured inputs in hierarchies of part-of relations. In that chapter we will make a parallel between SAEN and feature combination in kernel methods and we will show that the neural network approach is better for $\mathcal{H}$-decompositions. Indeed, not only neural networks allow us to combine representations in a way that is reminiscent of kernel methods, but also to contain the dimensionality of the features.

2.2 Perceptrons, support vector machines and neural networks

When a binary classification dataset is linearly separable we can learn to discriminate among classes by fitting the parameters $\mathbf{w}$ and b of an hyperplane $\langle \mathbf{w}, \mathbf{x} \rangle + b$. In this section we will briefly introduce the best known algorithms to learn with separators.

2.2.1 Perceptron

One of the best known algorithms for learning with separators is the perceptron. The perceptron minimizes the $0/1$-loss function defined as:

$$J_{0/1}(\mathbf{w}, b) = \frac{1}{N} \sum_{i=1}^{N} L_{0/1}(y_i, sign(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)) \tag{2.8}$$

tensor operations.

where

$$L_{0/1}(y, y') = \begin{cases} 1 & \text{if } y = y' \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The perceptron algorithm (cf. [Rosenblatt, 1958, 1962, Bishop, 2006]) converges only when the dataset is linearly separable. One way to overcome the problem of the convergence is to make the dataset linearly separable using the feature combination tricks explained in § 2.1.2. Indeed, if the dimensionality of the feature space is sufficiently large it is more likely that the dataset will be linearly separable. Nevertheless, two problems arise: first there could still be outliers that affect the linear separability; second when the dimensionality of the feature space is large, multiple hyperplanes may be able to separate the data. With respect to the first problem we need a robust algorithm, while for the second problem we need a good selection criterion for the hyperplane that best generalize on test data.

2.2.2 Support vector machines

Support vector machines, originally introduced by Vapnik [1963], exploit a clever trick to cope with high-dimensional features restricting the class of decision functions to maximum-margin hyperplanes.

Maximum-margin hyperplanes $\langle \mathbf{w}, \mathbf{x} \rangle + b$ linearly separate the data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ maximizing the distance $dist_{\mathbf{w},b}(\mathbf{x})$ of the point which is closer to the hyperplane of separation between the two classes. There are two canonical hyperplanes which run parallel to the hyperplane of separation on the left and right half space respectively. The space between the canonical hyperplanes is a *forbidden* zone in which no example can fall.

Using computational geometry we can derive the formulation of the optimization problem of support vector machines. Finding a maximum-margin hyperplane corresponds to solving the following $\max - \min$ problem:

$$\mathbf{w}^*, b^* = \arg\max_{\mathbf{w},b} \min_{i=1,\ldots,N} dist_{\mathbf{w},b}(\mathbf{x}_i) \tag{2.10}$$

in which all the examples are correctly classified. The distance $dist_{\mathbf{w},b}(\mathbf{x}_i)$ of point $\mathbf{x}_i$ from hyperplane $\langle \mathbf{w}, \mathbf{x} \rangle + b$ is:

$$dist_{\mathbf{w},b}(\mathbf{x}_i) = \frac{|\langle \mathbf{w}, \mathbf{x}_i \rangle + b|}{\|\mathbf{w}\|_2}. \tag{2.11}$$

Because we assume that the dataset is linearly separable and all the points are correctly classified we can rewrite $dist_{\mathbf{w},b}(\mathbf{x}_i)$ as $\frac{y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)}{\|\mathbf{w}\|_2}$ assuming that $y_i \in \{-1, 1\}$ and that when an example is correctly classified the signs of y_i and $\langle \mathbf{w}, \mathbf{x}_i \rangle + b$ agree (i.e. $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 0$).

We can then rewrite Eq. 2.10 as:

$$\begin{aligned} \mathbf{w}^*, b^* &= \arg\max_{\mathbf{w},b} \min_{i=1,\ldots,N} \frac{y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)}{\|\mathbf{w}\|_2} \\ \text{s.t. } & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 0, \ \forall i = 1, \ldots, N \end{aligned} \tag{2.12}$$

However, the above problem is underspecified since for any scalar $\alpha > 0$, if $\mathbf{w}^*, b^*$ is a solution also $\alpha\mathbf{w}^*, \alpha b^*$ is a solution. For this reason instead of $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 0$ we use $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$, in this way we also ensure that the margin is never below $\frac{1}{\|\mathbf{w}\|_2}$ and we can rewrite Eq. 2.12 as

$$\begin{aligned} \mathbf{w}^*, b^* &= \arg\max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|_2} \\ \text{s.t. } & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1, \ \forall i = 1, \ldots, N \end{aligned} \tag{2.13}$$

Eq. 2.13 can be equivalently written as the optimization problem

$$\begin{aligned} \mathbf{w}^*, b^* &= \arg\max_{\mathbf{w},b} \|\mathbf{w}\|_2^2 \\ \text{s.t. } & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1, \ \forall i = 1, \ldots, N \end{aligned} \tag{2.14}$$

that corresponds to the quadratic optimization program of hard-margin SVMs. When Eq 2.14 is formulated in the dual the decision function $f(x) = \langle \mathbf{w}, \mathbf{x} \rangle + b$ can be rewritten as

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i k(x_i, x) + b \tag{2.15}$$

where $\alpha_i \geq 0$ are dual coefficients. Interestingly, only the support vectors i.e. the examples lying on the canonical hyperplanes have nonzero coefficients, this means that it is possible to throw away the other data points without affecting the result of the optimization.

The soft-margin version is a variation that allows some examples $\mathbf{x}_i$ to violate the maximum-margin condition by a nonnegative slack variable ζ_i. These violations are penalized in the cost function leading to the optimization problem:

$$\begin{aligned} \mathbf{w}^*, b^* &= \arg\max_{\mathbf{w},b,\zeta_i \geq 0} \|\mathbf{w}\|_2^2 + \frac{C}{N} \sum_{i=1}^{N} \zeta_i \\ \text{s.t. } & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \zeta_i, \ \forall i = 1, \ldots, N. \end{aligned} \tag{2.16}$$

The above optimization problem is robust to outliers that perhaps could make the dataset not linearly separable. The introduction of the slack variables ζ_i implies an additional

term $\frac{1}{N}\sum_{i=1}^{N}\zeta_i$ in the cost function which is reweighed by the hyper-parameter $C > 0$. In order to obtain a good generalization error is fundamental to carefully tune C using model selection, and keeping in mind that the higher the value of C the more the trained model will be sensitive to outliers. SVMs have been used for our experiments in chapters 3 and 5. Eqs. 2.14 and 2.16 are quadratic programs that can be solved perhaps with sequential minimal optimization [Platt, 1998].

Equation 2.16 can be rewritten as:

$$\mathbf{w}^*, b^* = \arg\max_{\mathbf{w},b,\zeta_i \geq 0} \|\mathbf{w}\|_2^2 + CJ_{hinge}(\mathbf{w}, b) \tag{2.17}$$

Where $\|\mathbf{w}\|_2^2$ is an ℓ_2-regularization term and

$$J_{hinge}(\mathbf{w}, b) = \frac{1}{N}\sum_{i=1}^{N}\max(0, 1 - y_i(\langle \mathbf{w}, \mathbf{x}_i\rangle + b)) \tag{2.18}$$

it called *hinge* loss. Even if the *hinge* loss is not differentiable Eq. 2.17 can be optimized with subgradient methods [Shalev-Shwartz et al., 2007].

2.2.3 Neural networks

While the decision function $f(x)$ of perceptrons and SVMs is a linear combination $f(x) = \sum_{j=1}^{d} w_j\phi_i(\mathbf{x})$ of features which are usually fixed during the training, multilayer neural networks can jointly learn the feature space and to separate the target.

Indeed, multilayer neural networks are obtained by stacking multiple layers as follows:

$$f(\mathbf{x}) = F(F(\ldots F(\mathbf{x}; W^0, \mathbf{b}^0)\ldots; W^{K-1}, \mathbf{b}^{K-1}); W^K, \mathbf{b}^K) \tag{2.19}$$

each function $F : \mathbb{R}^s \to \mathbb{R}^d$ with parameters $W \in \mathbb{R}^{d\times s}$ and $\mathbf{b} \in \mathbb{R}^s$ represents a layer that can be expressed as:

$$F(\mathbf{x}; W, \mathbf{b}) = a(W\mathbf{x} + \mathbf{b}) \tag{2.20}$$

where $a : \mathbb{R} \to \mathbb{R}$ is a nonlinear activation function applied element-wise. Popular choices for a are the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and the hyperbolic tangent $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. More recently the rectified linear units $\mathrm{ReLU}(x) = \max(0, x)$ gained more interest in the deep learning community [Glorot et al., 2011, Maas et al., 2013].

Because of the nonlinear activation functions, multilayer neural networks are non-convex and are usually optimized by gradient descent with backpropagation.

A popular loss function that is used when the last layer uses sigmoid activation functions is:

$$J_{ce} = \frac{1}{N} \sum_{i=1^N} L_{ce}(y_i, \sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)) \tag{2.21}$$

where the term $L_{ce}(y, y') = -y \log(y') - (1 - y) \log(1 - y')$ is the cross-entropy between the target label y and the prediction y' and we assume that $y_i \in \{0, 1\}$.

Chapter 3

Graph Invariant Kernels

In recent years there has been renewed interest in graph kernels which can handle continuous (possibly high dimensional) attributes. A vast body of literature on graph kernels is devoted to symbolic only structures, which means that vertices (and possibly edges) are labeled by a number of discrete attributes. A popular approach to define graph kernels in this case is to count the number of common substructures (which we call *patterns*). Examples include tree-structured patterns [Ramon and Gärtner, 2003, Mahé and Vert, 2009], paths [Kashima et al., 2003] or shortest paths [Borgwardt and Kriegel, 2005], cyclic and tree patterns [Horváth et al., 2004], neighborhoods [Costa and De Grave, 2010], or arbitrary frequent patterns [Deshpande et al., 2005].

Graphs with continuous attributes have been much less investigated, but one graph kernel that handles continuous attributes, is based on random walks (RW) labels [Kashima et al., 2003]. The RW kernel can be computed in $O(V^6)$. Borgwardt and Kriegel [2005] proposed a variant based on shortest-paths (SP), which avoids the tottering[1] problem of RW kernels. The SP kernel has a running time of $O(V^4)$. This has been recently improved by Feragen et al. [2013] who introduced the GRAPHHOPPER kernel, also based on shortest-paths, reporting a running time of $O(V^2(E+\log V+\Delta^2))$ (where Δ is the graph diameter). Kriege and Mutzel [2012] propose graph kernels in which subgraphs are matched with a score which is computed as the product between a weight function and the kernels on vertex and edge attributes.

We now upgrade existing graph kernels to continuous attributes by using graph and vertex invariants. Vertex invariants are functions that color vertices of a graph in a way that is not affected by isomorphism. They form the basis for several

[1] Tottering is a a weakness of RW kernels that occurs when walks of infinite length go back and forth along the same edge creating an artificially inflated similarity between two graphs that share a common edge [Mahé et al., 2004].

practical isomorphism checking algorithms [McKay and Piperno, 2014]. We consider the commonalities between graph kernels like the Weisfeiler-Lehman graph kernel (WLGK) [Shervashidze et al., 2011], the neighborhood subgraph pairwise distance kernel (NSPDK) [Costa and De Grave, 2010], the propagation kernels [Neumann et al., 2012b] and GRAPHHOPPER [Feragen et al., 2013] and summarize them in a general formulation which we call Graph Invariant Kernels (GIK, pronounce "Geek"). GIKs decompose graphs into sets of vertices which are compared by a kernel that measures both their attribute and their structural similarity. The structural similarity indicates to which extent vertices play the same role in the graph they belong to. Our formulation allows arbitrary patterns (e.g. other than the shortest paths used by GRAPHHOPPER) and arbitrary graph and vertex invariants that can be obtained with color propagation schemas (e.g. Weisfeiler-Lehman, Propagation kernel). We also propose spectral coloring which exploits eigen-decompositions.

We show that the upgrade to continuous values provided by GIKs performs very well on a number of new and existing benchmarks. We experiment with different types of vertex invariants, including Weisfeiler-Lehman and spectral colors and compare the shortest paths used by GRAPHHOPPER with neighborhood subgraphs.

The content of this chapter consists of research previously published in:

- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. Graph Invariant Kernels. In: International Joint Conference on Artificial Intelligence. 2015. p. 3756-3762.

3.1 Notation and definitions

We introduce some specific notation that we use within this chapter and some definitions. We consider undirected graphs $G = (V, E)$ where V is the vertex set and E the edge set. Both vertices and edges may be labeled. $\ell : V \mapsto \mathcal{X}$ is the vertex labeling function where $\mathcal{X}$ may incorporate both discrete and continuous attributes. When necessary, we distinguish continuous and discrete labels as $\ell_c(v)$ and $\ell_d(v)$, respectively. Similarly, we denote by $l : E \mapsto \mathcal{Z}$ the edge labeling function. When needed we represent the connectivity of the graph G with the adjacency matrix A.

Definition 3. *Two graphs G and G' are isomorphic, written $G \approx G'$, if there exist a bijection $f : V \mapsto V'$ (called an isomorphism) such that $\{u, v\} \in E$ iff $\{f(u), f(v)\} \in E'$. In the case of labeled graphs, we also require $\ell(v) = \ell(f(v))$ and $l(u, v) = l(f(u), f(v))$.*

Definition 4. *An invariant $\mathcal{I}$ is a function on graphs such that $G \approx G'$ implies $\mathcal{I}(G) = \mathcal{I}(G')$. If the reverse implication is also true, then $\mathcal{I}$ is called a complete invariant.*

Definition 5. *A vertex invariant is a function $\mathcal{L} : V \mapsto \mathcal{C}$ that assigns each vertex v of G a value $\mathcal{L}(v)$, called the color of v, that is preserved under any isomorphism f, i.e. $\mathcal{L}(v) = \mathcal{L}(f(v))$.*

Definition 6. *For any two isomorphic graphs G and G', a vertex invariant $\mathcal{L}$ is complete if any bijective function $f : V_G \to V_{G'}$ that satisfies $\mathcal{L}(v) = \mathcal{L}(f(v))$ is a graph isomorphism map between G and G'.*

3.2 Graph Invariant Kernels

GIKs measure the similarity between two attributed graphs G and G' by comparing their vertices with a kernel function $k_{\text{ATTR}}(v, v')$ between the continuous attributes ℓ_c and reweighing their similarity with a function $w(v, v')$:

$$k(G, G') = \sum_{v \in V(G)} \sum_{v' \in V(G')} w(v, v') k_{\text{ATTR}}(v, v'). \tag{3.1}$$

The weight $w(v, v')$ measures the structural similarity between vertices and can be designed combining an $\mathcal{R}$-decomposition relation [Haussler, 1999], a function $\delta_m(g, g')$ and a kernel on vertex invariants k_{INV}.

We first define $w(v, v')$ as a count on common graph invariants:

$$w(v, v') = \sum_{\substack{g \in \mathcal{R}^{-1}(G) \\ g' \in \mathcal{R}^{-1}(G')}} k_{\text{INV}}(v, v') \frac{\delta_m(g, g')}{|V_g||V_{g'}|} \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}. \tag{3.2}$$

An $\mathcal{R}$-decomposition relation is a binary relation $\mathcal{R}(G, g)$ which encodes that "g is part of G" and specifies a decomposition of G into its *parts* (*patterns*). We denote with $\mathcal{R}^{-1}(G)$ the multiset of all patterns in G.

According to Equation 3.2 (also pictorially illustrated in Figure 3.1) the weight between a pair of vertices increases whenever the two vertices appear in the same pattern with the same structural role.

The function $\delta_m(g, g')$ is used to determine whether the two patterns match, while the indicator function $\mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}$ is introduced to select only the subgraphs g and g' in which the vertices v and v' are involved respectively.

The kernel function $k_{\text{INV}}(v, v')$ is used to measure the similarity between the vertex colors $\mathcal{L}(v)$ and $\mathcal{L}(v')$ produced by a vertex invariant $\mathcal{L}$ and encodes the extent to which the vertices play the same structural role in the pattern.

A complete vertex invariant gives the most fine-grained matches and has the same effect as using an isomorphism map f, while weaker invariants induce spurious matches. Weaker invariants can be desirable because they allow to compare non isomorphic graphs. Vertex invariants can be weakened by exploiting the $\mathcal{R}$-decomposition relation and computing the vertex invariants with respect to the patterns. We specialize the notation for $k_{\text{INV}}(v, v')$ into *local* $k^g_{\text{INV}}(v, v')$ and *global* $k^G_{\text{INV}}(v, v')$ as to distinguish whether the vertex invariants in question were computed with respect to the patterns $g \in \mathcal{R}^{-1}(G)$ or the whole graph G respectively. We now explain how a $\delta_m(g, g')$ function can define a hard-match between subgraphs generated by some $\mathcal{R}$-decomposition relation, while we refer the reader to § 3.3 for the vertex invariants that can be used to define structural similarity $k_{\text{INV}}(v, v')$ between vertices.

3.2.1 Hard-match kernels for discrete labels

If graphs are labeled by discrete symbols, a simple choice for $\delta_m(g, g')$ is the hard match function

$$\delta(g, g') \doteq \begin{cases} 1 & \text{if } g \equiv g' \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

for a given definition of the equivalence relation $\equiv$. By instantiating the definition of $\mathcal{R}$ and $\equiv$, we can obtain a fairly large family of hard pattern-match graph kernels. For example, in the case of SP kernels [Borgwardt and Kriegel, 2005] $\mathcal{R}(G, g)$ iff g is a shortest-path between some pair of vertices u and v in G. If edges are unlabeled, then a special case of the kernel presented in [Borgwardt and Kriegel, 2005] can be interpreted as a hard pattern-match kernel by defining $g \equiv g'$ iff the two serializations (obtained by concatenating the vertex labels) of g and g' are identical. A second example is the kernel described in [Horváth et al., 2004], where $\mathcal{R}(G, g)$ iff g is either a cycle in G, or a tree in the forest obtained by deleting all cycles from G. For this kernel, the equivalence $g \equiv g'$ is established by taking all possible serializations (for example all cyclic permutations in the case of cycles) and checking that the lexicographical minima are identical. As a last example, in the NSPDK [Costa and De Grave, 2010] the decomposition relation is parameterized by two natural numbers r and d and $\mathcal{R}_{r,d}(G, g)$ iff g is a pair of neighborhood subgraphs of G, g_1 and g_2, rooted at vertices v_1 and v_2, respectively, such that the two following conditions hold: (1) the shortest-path distance between v_1 and v_2 is d, and (2) for $i = 1, 2$, g_i is the neighborhood subgraph rooted in v_i. A neighborhood subgraph rooted in v_i consists of the subgraph induced by all vertices of G whose shortest-path distance from v_i is at most r. In [Costa and De Grave, 2010], $\equiv$ is defined as the graph isomorphism relation, approximated by hashing a canonical representation of the patterns.

A weaker hard-match kernel can be obtained using a graph invariant $\mathcal{I}$, i.e. $g \equiv g'$ if $\mathcal{I}(g) = \mathcal{I}(g')$. A simple graph invariant $\mathcal{I}_{\text{V}}$ is the number of vertices of a graph. Vertex colors can be used to construct stronger graph invariants because they are preserved

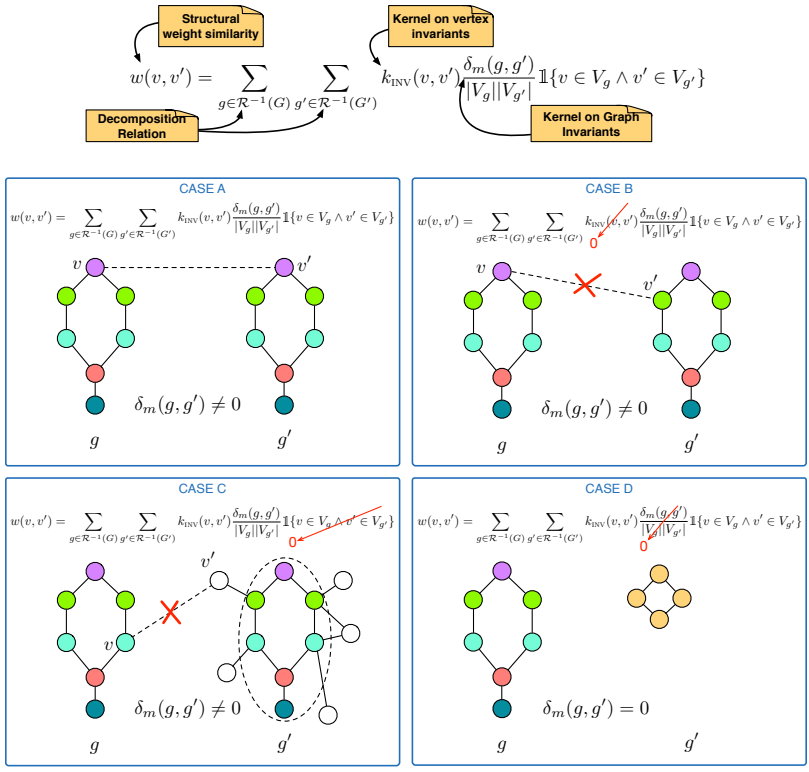


Figure 3.1: We propose a pictorial representation of the structural weight similarity $w(v, v')$ among vertices. The computation of $w(v, v')$ involves a sum over the subgraphs g and g' generated by $\mathcal{R}^{-1}(G)$ and $\mathcal{R}^{-1}(G')$ respectively that give a contribution to the similarity score. We distinguish among four different cases:
case A is the only one that gives a contribution to $w(v, v')$ because v and v' are nodes in V_g and $V_{g'}$ respectively, their kernel on vertex invariants $k_{inv}(v, v')$ is nonzero and g and g' match (i.e. $\delta_m(g, g') \neq 0$);
case B gives no contribution since $k_{inv}(v, v') = 0$ (i.e. v and v' play different structural roles in g, g');
case C gives no contribution to $w(v, v')$ because either $v \notin V_g$ or $v' \notin V_{g'}$;
case D gives no contribution to $w(v, v')$ because g and g' do not match (i.e. $\delta_m(g, g') = 0$).

under graph isomorphism. A graph invariant can be obtained as the lexicographically sorted set of vertex colors or edge colors, where the color of an edge $\{u, v\} \in E$ can be represented as $(\mathcal{L}(u), \mathcal{L}(v), l(u, v))$. The latter method was introduced by [Costa

and De Grave, 2010] combined with distance coloring as in § 3.3.3. Graph invariants are hashed to integers so that we can compute the subgraph match function $\delta_m(g, g')$ in constant time. Collisions due to hashing can only weaken the graph invariant.

On the other hand graph invariants used by δ_m are always computed on patterns (i.e. at local level). Computing δ_m at the global level would not allow to capture meaningful correlations between a pair of graphs G and G' leading to poor generalization performance.

GIKs can also be expressed as $\mathcal{R}$-convolutional kernels on graphs [Haussler, 1999] and this exercise is left to the reader. Furthermore, GIKs are positive semidefinite (PSD) since they are by combining PSD using operations that are internal with respect to the class of PSD kernels.

3.3 Vertex invariants

In this section we review some possible vertex invariants to construct specific instances of the vertex coloring kernel.

3.3.1 Weisfeiler-Lehman coloring

The 1-dimensional Weisfeiler-Lehman (WL) refinement algorithm [Weisfeiler, 1976] finds a partition of the vertices of a graph in an iterative fashion:

- Initialization: All colors are initialized using vertex labels, i.e. $\forall v \in V\ \mathcal{L}^{(0)}(v) \doteq \ell_d(v)$
- Recoloring step: $\forall v \in V$

$$\mathcal{L}^{(t+1)}(v) = id(\{\mathcal{L}^{(t)}(w) | w \in \mathcal{N}_G(v)\}) \tag{3.4}$$

  where $\mathcal{N}(v)$ is the set of vertices adjacent to v and the injective function id returns the string of the vertex colors of the neighbors sorted in lexicographic order. Often a hash function is used to represent this string with an integer. [2]

- Termination criterion: recoloring converges when the number of distinct colors stops increasing, which means that the vertices in the graph cannot be further partitioned. In practice the recoloring process is stopped after a predefined number h of iterations.

[2] In general hash functions are not injective unless they are *perfect*. However, they work well in practice when they have low collision probability.

The 1-dimensional WL color of a vertex is finally obtained as the concatenation of the colors at all time steps:

$$\mathcal{L}(v) = \left[\mathcal{L}^{(0)}(v)|\ldots|\mathcal{L}^{(h)}(v)\right] \tag{3.5}$$

For edge-labeled graphs, we propagate the vertex colors of the neighbors together with the edge labels:

$$\mathcal{L}^{(t+1)}(v) = id(\{(\mathcal{L}^{(t)}(w), l(v,w))|w \in \mathcal{N}_G(v)\}) \tag{3.6}$$

A pictorial representation of the above equation is shown in Figure 3.2. The positive definite kernel on this kind of vertex invariant can be defined as:

$$\langle \mathcal{L}(v), \mathcal{L}(v')\rangle = \sum_{i=0}^{h} \mathbb{1}\left\{\mathcal{L}^{(i)}(v) = \mathcal{L}^{(i)}(v')\right\} \tag{3.7}$$

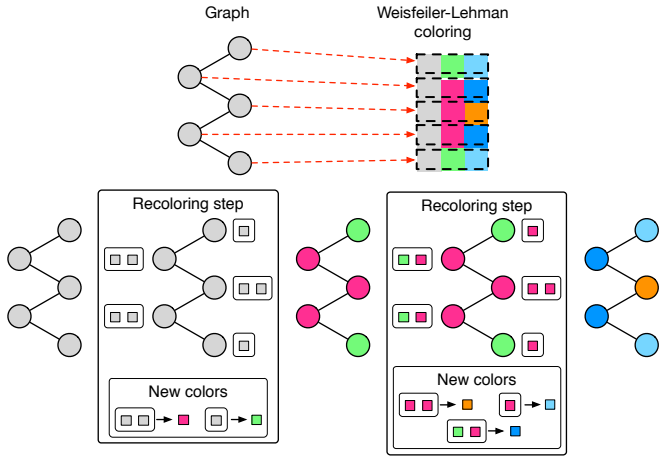


Figure 3.2: In the upper part of the picture we show an example graph and the Weisfeiler-Lehman colors of its vertices, which derive from the concatenation of the initial labels and two steps of the Weisfeiler-Lehman algorithm. In the lower part we show two steps of Weisfeiler-Lehman relabeling. The rectangle with the caption "new colors" represents how the id function associates multisets of colors to new colors. Since the id function is injective, when choosing the colors, we ensured that distinct multisets of colors were associated to distinct colors.

3.3.2 Label updates in propagation kernels

When introducing propagation kernels (PROP), Neumann et al. [2012b] proposed two WL-like label update approaches:

- Diffusion updates:

$$\mathcal{L}^{(t+1)}(v_i) = \sum_{v_j \in \mathcal{N}_G(v_i)} T_{ij}\mathcal{L}^{(t)}(v_j) \tag{3.8}$$

  where $T = D^{-1}A$ is the transition matrix derived from the normalization of the rows of the adjacency matrix, and D is the diagonal degree matrix $D_{ii} = \sum_{k=0}^{V} a_{ik}$

- Label propagation: as above except that before each iteration $t+1$ the labels $\ell_d(v_i)$ of the originally labeled vertices overwrite the value assigned to $\mathcal{L}^{(t)}(v_i)$ by the propagation process.

3.3.3 Distance coloring

This method was initially suggested in [Costa and De Grave, 2010] to compute the NSPDK and assumes connected and rooted patterns, i.e. g has a distinguished vertex r.

- Starting from r, a breadth-first visit is used to compute all-pairs distances $d(v, w)$ between the vertices.
- Each vertex is colored as follows:

$$\mathcal{L}(v) = id(\{(\ell_d(w), d(v, w))|w \in V(G)\})\ \forall v \in V(G) \tag{3.9}$$

Information about the root of the subgraph can be included by adding to each vertex color information about its distance from the root vertex.

3.3.4 Spectral coloring

Eigenvalues and eigenvectors of a graph adjacency matrix are insensitive to vertex permutation. For this reason, starting from the seminal work of Umeyama [1988], spectral methods have been popular in pattern recognition as graph matching tool. Spectra can be also used for isomorphism testing if eigenvalues have bounded multiplicity [Babai et al., 1982].

A weighted adjacency matrix W can be defined by using vertex labels $\ell_c(v)$, assuming that $\mathcal{X}$ is a metric space endowed with a distance function d:

$$w_{ij} = a_{ij}e^{-\gamma d^2(\ell_c(v_i),\ell_c(v_j))} \tag{3.10}$$

where a_{ij} are the entries in the adjacency matrix A of the graph and γ is a non-negative scalar hyperparameter of the heat kernel. In spectral coloring, a vertex invariant is

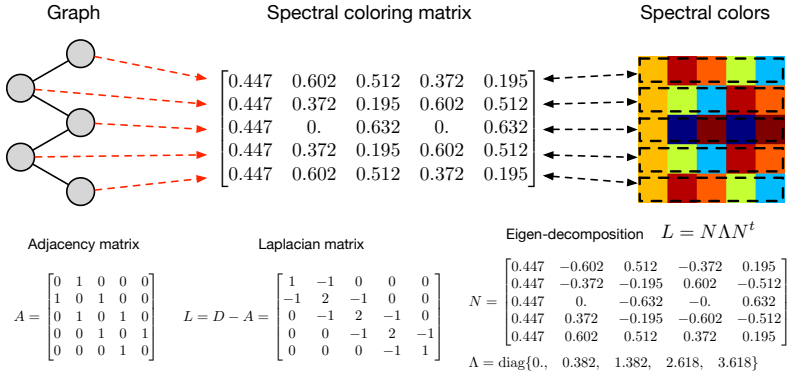


Figure 3.3: In the upper part of the picture we show an example graph and the spectral coloring features of its vertices (Spectral coloring matrix) the spectral coordinates are also pictorially represented as a color matrix using the `matplotlib` function `implot` and the `interpolation` parameter set to "`nearest`". In the lower part we show the adjacency matrix and the combinatorial laplacian of the example graph. The spectral coloring matrix (upper part) is obtained via a canonization of the eigenvectors of the combinatorial laplacian of the example graph as described in § 3.3.4.

obtained from the the Laplacian embedding of the graph as follows. Let $L \doteq D - W$ denote the graph Laplacian matrix, with D the diagonal degree matrix. The embedding $X \in \mathbb{R}^{V \times V}$ is the solution of

$$\min_{X : X^t X = I} \sum_{i=1}^{V} \sum_{j=1}^{V} \|\mathbf{x}(v_i) - \mathbf{x}(v_j)\|^2 w_{ij}. \tag{3.11}$$

This equation finds vertex coordinates $\mathbf{x}(v)$ (rows of X) whose euclidean distance preserves the graph connectivity and is equivalent to:

$$\min_{X : X^t X = I} tr(X^t L X) \tag{3.12}$$

the solution can be obtained by solving an eigenvalue problem:

$$L\mathbf{x}_i = \lambda_i \mathbf{x}_i \tag{3.13}$$

where the eigenvectors $\mathbf{x}_i$ (columns of X) are sorted according to the corresponding eigenvalues, i.e. $0 = \lambda_1 \leq \lambda_2 \cdots \leq \lambda_V$. The row $\mathbf{x}(v)$ of X is the embedding coordinate of the vertex v in the graph. The Spectral vertex invariant is obtained by switching to zero the λ-eigenvectors whose eigenvalue λ has multiplicity $\mu > 1$ and

taking the absolute value of the components of the eigenvalues otherwise.

$$\mathcal{L}^{(i)}(v) = \begin{cases} |x_i(v)| & \text{if } \mu_i = 1 \\ 0 & \text{if } \mu_i > 1 \end{cases} \tag{3.14}$$

Some lifted inference approaches [Mladenov et al., 2012] perform color passing on factor graphs to cluster symmetric variables and speed up intractable inference problems. We solve a complementary problem, and propose the canonized Laplacian embedding to reveal the symmetries between the vertices of a graph. The graph Laplacian is preferred over the one of the adjacency matrix because its spectrum gives more insight on the connectivity of the graph. The smallest eigenvalue λ_1 of the Laplacian spectrum of a connected graph is 0, has multiplicity 1 and its λ_1-eigenvector is constant. All the other eigenvectors are orthogonal to the constant eigenvector and thus balanced. It is actually this property that allows spectral coloring features to encode the symmetries of a graph. From an implementation point of view it is possible to limit the dimensionality of the embedding to the first h components. In case there are less than h vertices the embedding can be padded with zeros.

3.4 Algorithmic issues and running time

We limit our analysis to kernels that use an $\mathcal{R}$-decomposition relation that generates connected subgraphs whose enumeration scales linearly with the number of vertices in the graph G. The time complexity of our method is $V^2(C_1 + C_2\tau V_g^2)$ where V is the number of vertices, V_g is the number of vertices in a subgraph $g \in \mathcal{R}^{-1}(G)$, C_1 and C_2 are two costs and τ is the subgraph matching count:

$$\tau = \sum_{g \in \mathcal{R}^{-1}(G)} \sum_{g' \in \mathcal{R}^{-1}(G')} \delta_m(g, g'). \tag{3.15}$$

We assume that the cost of computing k_{INV} or k_{ATTR} scales with the dimension d_{INV} and d_{ATTR} of the corresponding features. In particular we have cost d^G_{INV} when the kernel k^G_{INV} is on global graph invariants and d^g_{INV} when the kernel k^g_{INV} is on local graph invariants. According to Equation 3.2 we have $C_1 = d_{\text{ATTR}}$ and $C_2 = d^g_{\text{INV}}$. If we use global instead of local graph invariants we can rewrite Equation 3.2 as:

$$w(v, v') = k^G_{\text{INV}}(v, v') \sum_{\substack{g \in \mathcal{R}^{-1}(G) \\ g' \in \mathcal{R}^{-1}(G')}} \frac{\delta_m(g, g')}{|V_g||V_{g'}|} \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\} \tag{3.16}$$

and thus we obtain $C_1 = d_{\text{ATTR}} + d^G_{\text{INV}}$ and $C_2 = 1$.

The worst case scenario for τ occurs when the subgraph matching function $\delta_m(g, g')$ is always 1, in this case we have $\tau = V^2$. This can happen in two cases: 1) when the

Table 3.1: Comparison between different GIKs and GRAPHHOPPER. For all datasets we used SVM-classifiers. Except for QC the accuracy was estimated by 10-times 10-fold cross-validation reporting means and standard deviations as in [Feragen et al., 2013]. QC has predefined train-test splits. The SVM regularization parameter was selected with an internal k-fold cross-validation on the training data ($k = 3$ except $k = 10$ for QC).

	$\text{ENZYMES}_{\text{SYM}}$		PROTEIN		$\text{SYNTHETIC}_{\text{NEW}}$		FRANKENSTEIN		QC	
	WITHOUT CONT.	WITH CONT.	WITHOUT CONT.	WITH CONT.	WITHOUT CONT.	WITH CONT.	WITHOUT CONT.	WITH CONT.	WITHOUT CONT.	WITH CONT.
NSK_{V}	25.9 ± 1.1	71.8 ± 1.0	72.1 ± 0.4	74.2 ± 0.7	78.4 ± 1.9	81.9 ± 1.1	67.9 ± 0.2	72.9 ± 0.3	37.8	92.6
NSK_{WL}	56.5 ± 1.1	72.2 ± 0.8	71.7 ± 0.4	76.2 ± 0.4	50.0 ± 0.0	50.0 ± 0.0	74.2 ± 0.3	77.3 ± 0.1	47.8	91.0
GWL_{V}	55.7 ± 1.0	72.6 ± 0.8	$\mathbf{74.9 \pm 0.6}$	76.1 ± 0.8	$\mathbf{80.8 \pm 1.2}$	82.8 ± 1.0	73.5 ± 0.3	77.3 ± 0.2	49.8	93.6
GWL_{WL}	$\mathbf{58.6 \pm 1.4}$	71.3 ± 1.1	73.6 ± 0.5	75.8 ± 0.6	50.0 ± 0.0	50.0 ± 0.0	$\mathbf{75.1 \pm 0.2}$	$\mathbf{78.9 \pm 0.3}$	48.6	89.6
LWL_{V}	54.5 ± 1.1	$\mathbf{73.3 \pm 0.9}$	74.4 ± 0.4	$\mathbf{76.6 \pm 0.6}$	80.6 ± 1.5	$\mathbf{83.0 \pm 1.0}$	73.0 ± 0.2	77.6 ± 0.2	47.2	**94.6**
LWL_{WL}	57.0 ± 1.1	72.0 ± 0.9	71.9 ± 0.6	76.5 ± 0.5	50.0 ± 0.0	50.0 ± 0.0	74.1 ± 0.2	78.3 ± 0.3	47.4	91.8
GSGK_{V}	29.8 ± 0.6	71.8 ± 1.0	73.2 ± 0.3	74.7 ± 0.5	78.2 ± 2.1	82.4 ± 0.9	70.1 ± 0.3	74.0 ± 0.3	44.4	92.6
GSGK_{WL}	56.7 ± 1.2	72.2 ± 0.7	72.9 ± 0.5	76.4 ± 0.4	50.0 ± 0.0	50.0 ± 0.0	75.0 ± 0.3	77.6 ± 0.2	47.8	91.0
LSGK_{V}	31.9 ± 1.0	71.9 ± 1.0	72.3 ± 0.4	74.4 ± 0.6	78.7 ± 2.0	82.2 ± 1.1	72.1 ± 0.2	74.9 ± 0.2	42.4	92.2
LSGK_{WL}	56.6 ± 1.3	72.1 ± 0.8	71.7 ± 0.3	76.1 ± 0.5	50.0 ± 0.0	50.0 ± 0.0	74.2 ± 0.2	77.4 ± 0.2	**51.4**	91.0
GRAPHHOPPER		69.5 ± 0.7		72.7 ± 0.3		73.9 ± 1.7		68.7 ± 0.4		91.4

graph invariant $\mathcal{I}$ is constant and every pair of patterns matches and 2) when the patterns are all identical. In the former case we should resort to a more selective invariant, while in the latter case we should avoid computing $w(v, v')$ because its value is constant. This case in which all the patterns are identical is verified for 0-neighborhood subgraphs (vertices) or r-neighborhood subgraphs in which $r \geq \frac{\Delta}{2}$, where Δ is the diameter of the G. When the radius r is at least twice the diameter Δ of the graph G, all the corresponding r-neighborhood subgraphs are the graph G itself. In both cases there is no structural contribution, these cases are easy to detect and being aware of this fact we can compute the kernel in $O(V^2 C_1)$. Another option is to add structural information in order to avoid the extreme cases. We can incorporate discrete labels if present when $r = 0$ and root information when $r \geq \frac{\Delta}{2}$. Two rooted patterns must have both same graph invariant and same root vertex invariant in order to match . We obtain an upper bound on the number of vertices V_g of a r-neighborhood subgraph with maximum vertex degree d and diameter $\Delta = 2r$ using the Moore bound [Miller and Širán, 2005]:

$$V_g \leq 1 + d \sum_{i=0}^{\Delta-1} (d-1)^i = d^{\Delta} + O(d^{\Delta-1}) \tag{3.17}$$

So $V_g \in O(d^{2r})$ and the complexity of our method is $O(V^2(C_1 + C_2 \tau d^{4r}))$.

3.5 Experimental evaluation

We answer the following experimental questions:

Q1 How do GIKs compare to the state of the art?

Q2 Do our kernels produce more accurate classifiers when continuous attributes are introduced?

Q3 Can the graph representation of a sentence benefit from word vector attributes instead of discrete word attributes?

Q4 What are the best graph invariants?

3.5.1 Datasets

PROTEINS and **ENZYMES$_{\text{SYMM}}$** are sets of proteins from Dobson and Doig [2003] and from the BRENDA database [Schomburg et al., 2004], respectively. Vertices are secondary structure elements as in [Borgwardt et al., 2005].

SYNTHETIC$_{\text{NEW}}$ is a dataset of 300 examples with two classes, it is based on a random graph G with 100 nodes and 196 edges, whose vertices were annotated with scalar attributes sampled from $\mathcal{N}(0, 1)$. It was introduced in [Feragen et al., 2013].

FRANKENSTEIN is a dataset created by the fusion of the BURSI and MNIST datasets. BURSI is made by 4337 molecules with mutagenicity (AMES) classification, with 2401 mutagens and 1936 nonmutagens [Kazius et al., 2005]. Each molecule is represented as a graph whose vertices are labeled by the chemical atom symbol and edges by the bond type.

FRANKENSTEIN is a modified version of the BURSI dataset: we discarded bond type information and remapped the most frequent atom symbols (vertex labels) to MNIST digit images. The original atom symbols can only be recovered through the high dimensional MNIST vectors of pixel intensities, in this sense this is a challenging problem for a graph kernel that can handle continuous attributes.

QC and **WEASEL** are natural language processing datasets: QC is a dataset for question classification with fixed split (5452 train / 500 test) originally proposed in [Li and Roth, 2002]. We consider the classification task with six coarse labels. WEASEL is part of the CoNNL-2010 shared task dedicated to the detection of hedge cues, it is a binary classification task with fixed split (11111 train / 9634 test).

For both QC and WEASEL we represent a sentence as a graph whose vertices are tokens annotated with 300-dimensional word-vector attributes, and whose edges represent the word adjacency and the typed dependency relations extracted with the Stanford dependency parser [De Marneffe et al., 2006]. Word vectors were obtained using the WORD2VEC software [Mikolov et al., 2013] and the whole Wikipedia corpus for training.

3.5.2 Kernels

We combined vertex invariants to construct some GIKs that can handle continuous attributes (see also Table 3.2).

NSK [Costa and De Grave, 2010] was instantiated using an $\mathcal{R}$-decomposition relation that generates neighborhood subgraphs (NS) of increasing radius $r = 0, \ldots, R$ and δ_m as an indicator function that matches two r-neighborhood subgraphs only if they have the same radius r and $\mathcal{I}_{\text{WL}}$ subgraph invariant. Costa and De Grave [2010] originally used distance coloring $\mathcal{I}_{\text{DC}}$.

LWL and **GWL** are the local and global variant of the WL subtree kernel respectively. Strictly speaking, GWL is not equivalent to the WL subtree kernel. In fact instead of using an $\mathcal{R}$-decomposition relation which generates vertex patterns we used r-neighborhood subgraphs.

LSGK and **GSGK** are local and global version of Spectral Graph Kernel in which we used our spectral coloring method.

Table 3.2: Some instances of GIKs. For each instance we report the kernel on vertex invariants $k_{\text{INV}}(v, v')$ and whether the vertex invariant was applied at the global or the local level (see § 3.2).

kernel	$k_{\text{INV}}(v, v')$	
NSK	1	global
GWL	$\langle \mathcal{L}^G_{\text{WL}}(v), \mathcal{L}^G_{\text{WL}}(v') \rangle$	global
LWL	$\langle \mathcal{L}^g_{\text{WL}}(v), \mathcal{L}^g_{\text{WL}}(v') \rangle$	local
GSGK	$e^{-\gamma \| \mathcal{L}^G_{\text{SC}}(v) - \mathcal{L}^G_{\text{SC}}(v') \|^2}$	global
LSGK	$e^{-\gamma \| \mathcal{L}^g_{\text{SC}}(v) - \mathcal{L}^g_{\text{SC}}(v') \|^2}$	local

Each GIK was instantiated in combination with two different kernels on subgraph invariants: $\mathcal{I}_{\text{V}}$ and $\mathcal{I}_{\text{WL}}$. $\mathcal{I}_{\text{WL}}$ is more selective and reduces the subgraph matching count τ. When necessary we use the subscript to specify which subgraph invariant was employed. We used an $\mathcal{R}$-decomposition relation which generates neighborhood subgraphs of increasing radius $r = 0, \ldots, R$. Based on preliminary experiments, we fixed $R = 3$. The Gram matrices were normalized as proposed in [Costa and De Grave, 2010]: for each radius a different Gram matrix is extracted then normalized, we compute their sum and apply normalization again. For $\text{ENZYMES}_{\text{SYMM}}$, PROTEINS, $\text{SYNTHETIC}_{\text{NEW}}$ we chose the parameter γ of the RBF kernel as in [Feragen et al., 2013], to be $1/d_{\text{ATTR}}$ which is the inverse of the number of dimensions of the attributes. We did the same for the γ_{INV} of the RBF kernel on the continuous vertex invariants derived from spectral graph coloring $\mathcal{L}_{\text{SC}}$. The number of dimensions d_{INV} of $\mathcal{L}_{\text{SC}}$ was

set equal to the average number of vertices in a graph for GSGK and to the average number of vertices in a 3-neighborhood subgraph for LSGK. For FRANKENSTEIN we set $\gamma = 0.0073$, this value was taken from [Sevakula and Verma, 2012], while for QC we set $\gamma = 1$. This value was selected with 10-fold cross-validation on the training set of QC during preliminary experiments in which we used a kernel on bag of word vectors. Word vectors were matched with the RBF kernel.

Table 3.3: QC dataset using words as features. We report the accuracies obtained on the original implementations of WL and PROP using words instead of word vectors. The symbol T denotes the number of iterations and TV and w are parameters (see [Neumann et al., 2012b]). QC has fixed train/test split and the regularization SVM parameter was found with 10-fold cross-validation (only using the training set). These results can be compared to the ones obtained for QC in Table 3.1.

KERNEL		ACCURACY $T=1$	$T=2$	$T=3$
WL		89.2	89.0	88.0
PROP	(w = 0.01)	86.8	88.0	85.4
PROP	(w = 0.1)	77.4	78.2	79.8
PROP	(w = 1.0)	58.0	58.0	57.2
PROP_{TV}	(w = 0.01)	87.8	89.0	88.6
PROP_{TV}	(w = 0.1)	87.8	88.6	88.4
PROP_{TV}	(w = 1.0)	79.0	81.8	81.0

3.5.3 Experiments

E1 In Table 3.1 we show the classification accuracy that we achieved without ($k_{\text{ATTR}} = 1$) and with ($k_{\text{ATTR}} = \text{RBF}(\gamma)$) kernel on continuous attributes. We use the bold font for the results of each column that have the highest mean accuracy. For FRANKENSTEIN we also measured the area under the roc curve (AUROC) which is 0.74 for GRAPHHOPPER with continuous attributes. With GWL_{WL} we obtained 0.82 AUROC without continuous attributes and 0.86 with continuous attributes. For all datasets we used SVM-classifiers. Except for QC and WEASEL the accuracy was estimated by 10-times 10-fold cross-validation reporting means and standard deviations [Feragen et al., 2013]. QC and WEASEL have predefined train-test splits. The SVM regularization parameter was selected with an internal k-fold cross-validation on the training data ($k = 3$ except $k = 10$ for QC and WEASEL).

E2 In Table 3.3 we report the accuracies obtained on the original implementations of WL and PROP using words instead of word vectors. The symbol T denotes the number

of iterations and TV and w are parameters (see [Neumann et al., 2012b]). These results can be compared with Table 3.1.

E3 In Table 3.4 we show the results on WEASEL for LWL_V and GRAPHHOPPER. We used word vectors and no task specific knowledge. We compare to the state of the art which exploits task specific knowledge encoded in word lists. We set the radius of the r-neighborhood subgraphs to $r = 0, 1$ as done by Verbeke et al. [2012].

Table 3.4: WEASEL sentence classification. We show the results on WEASEL for LWL_V and GRAPHHOPPER. We used word vectors and no task specific knowledge. We compare to the state of the art which exploits task specific knowledge encoded in word lists. We set the radius of the r-neighborhood subgraphs to $r = 0, 1$ as done by Verbeke et al. [2012].

METHOD	F1-SCORE
LWL_V (r = 0, 1)	55.7%
LWL_V (r = 0)	41.9%
LWL_V (r = 1)	58.3%
GRAPHHOPPER	48.8%
[Verbeke et al., 2012]	61.5%

The experiments were run on a 16 cores machine (Intel Xeon CPU E5-2665@2.40GHz and 96GB of RAM). The code of GIKs was written in the Python programming language, while for GRAPHHOPPER we used the MATLAB implementation from Feragen et al. [2013]. The difference between the programming languages makes hard to perform fine-grained time measurements. We measured the wall-clock execution time.

Table 3.5: Runtime of our most accurate and fast GIK selected from Table 3.1 compared to the runtime of GRAPHHOPPER.

DATASET	MOST ACCURATE AND FAST GIK		GRAPHHOPPER
$\text{ENZYMES}_\text{SYMM}$	NSK_WL	2' 36"	4' 53"
PROTEIN	NSK_WL	11' 00"	35' 27"
$\text{SYNTHETIC}_\text{NEW}$	GWL_V	15' 22"	9' 48"
FRANKENSTEIN	GWL_WL	1h 20'	2h 5'
QC	LWL_V	2h 30'	1h 30'

In Table 3.5 we show the runtime of our most accurate and fast GIK selected from Table 3.1 compared to the runtime of GRAPHHOPPER. The experiment that had the highest runtime was on the dataset WEASEL: LWL_V terminated in 73h 47' while

GRAPHHOPPER terminated in 75h 43'. We also mention that the spectral kernels tend to be slower and in some cases (e.g. on FRANKENSTEIN) can have a slowdown of a factor of 2 with respect to GRAPHHOPPER.

3.5.4 Discussion

A1 Our experiments show that we obtained state-of-the-art results for all the datasets in Table 3.1. $\text{ENZYMES}_{\text{SYMM}}$, PROTEINS, QC and WEASEL are are real-world datasets. On $\text{SYNTHETIC}_{\text{NEW}}$ we verified that its random nature combined with the graph invariant $\mathcal{I}_{\text{WL}}$ leads to diagonal kernel matrices and this explains the poor classification performance. On the other hand if we use the subgraph invariant $\mathcal{I}_{\text{V}}$, we outperform GRAPHHOPPER on its own artificial benchmark. In our experiments on FRANKENSTEIN (see Table 3.1), in some cases (GWL_{WL}) we have an increase of 0.12 points of AUROC with respect to GRAPHHOPPER. This performance mismatch is also due to the nature of the dataset from which FRANKENSTEIN was originated. Indeed neighborhood subgraphs are known to perform well on BURSI [Costa and De Grave, 2010], and shortest paths are probably not the best pattern for this kind of dataset. We consider as upper bound the result obtained with NSPDK ($R = 3$, $D = 0$) on BURSI, which is 0.91 AUROC score. The best result obtained on FRANKENSTEIN with continuous attributes (0.86 AUROC score) is significantly higher than the best result obtained without (0.82 AUROC score). On QC we obtain the 94.6% (see Table 3.1) which is the state of the art. Our result can be compared to the 94.8% obtained by Croce et al. [2011], when they combine kernels on lexical centered trees LCT with a word vector representation obtained with latent semantic analysis. Nevertheless GIKs are a generic tool, not specialized for natural language as LCT and we did not rely on manually built word lists. The 58.3% f1-score (see Table 3.4) obtained on WEASEL with $r = 1$ is comparable with the top five results of the CoNNL-2010 shared task. The state of the art for the task is 61.5% and was obtained by Verbeke et al. [2012] also exploiting task specific word lists not used by GIKS. The runtime measurements in Table 3.5 show that we could always instantiate GIKs obtaining comparable or higher accuracy with the same or inferior runtime except for $\text{SYNTHETIC}_{\text{NEW}}$ and QC in which the higher runtime is compensated by higher accuracies.

A2 Comparing the results in Table 3.1 we see that it is always the case that continuous attributes increase the accuracy.

A3 On QC WL and PROP use words as discrete attributes and cannot obtain more than 89% of accuracy (see Table 3.3). GIKs successfully use word vectors, indeed LWL_{V} achieves state-of-the-art results (see Table 3.1).

A4 If we consider the results in Table 3.1 we notice that LWL_{V} generally tends to give the best results, indeed local versions favor spurious matches and also the subgraph invariant $\mathcal{I}_{\text{V}}$ is weaker than $\mathcal{I}_{\text{WL}}$. Weaker invariants lower down the dimensionality

of the kernel and thus make the learning system less prone to overfitting. This effect was probably beneficial due to the relatively small size of some of the datasets that we used. The spectral color invariant can obtain good results, but other GIKs yield equally or more accurate classifiers and have better runtime. What makes spectral color invariants interesting is the nice property of providing vertex invariants embedded in the euclidean space. This makes Spectral Graph Kernel amenable for future work on improving scalability using sketching techniques.

3.6 Conclusion

The GIK reformulation of well known graph kernels allows to obtain more insights in the exploration of graph kernels with continuous attributes. The underlying idea was to employ vertex invariants for soft subgraph matching. We contributed new insights into graph kernels and to upgrade existing ones for use with continuous attributes. Several graph-kernel instances were then empirically evaluated on a number of new and existing benchmark datasets. The results showed that some combinations of graph and vertex invariants with continuous attributes lead to excellent performance.

Chapter 4

Shift Aggregate Extract Networks

Structured data representations are common in application domains such as chemistry, biology, natural language, and social network analysis. In these domains, one can formulate a supervised learning problem where the input portion of the data is a graph, possibly with attributes on vertices and edges. While learning with graphs of moderate size (tens up to a few hundreds of nodes) can be afforded with many existing techniques, scaling up to large networks poses new significant challenges that still leave room for improvement, both in terms of predictive accuracy and in terms of computational efficiency.

Devising suitable representations for graph learning is crucial and nontrivial. A large body of literature exists on the subject, where graph kernels (GKs) and recurrent neural networks (RNNs) are among the most common approaches. GKs follow the classic R-decomposition approach of Haussler [1999]. Different kinds of substructures (e.g., shortest-paths [Borgwardt and Kriegel, 2005], graphlets [Shervashidze et al., 2009] or neighborhood subgraph pairs [Costa and De Grave, 2010]) can be used to compute the similarity between two graphs in terms of the similarities of their respective sets of parts. RNNs [Sperduti and Starita, 1997, Goller and Kuchler, 1996, Scarselli et al., 2009] unfold a template (with shared weights) over each input graph and construct the vector representation of a node by recursively composing the representations of its neighbors. These representations are typically derived from a loss minimization procedure, where gradients are computed by the backpropagation through structure algorithm [Goller and Kuchler, 1996]. Micheli [2009] proposed the architecture *neural networks for graphs* (NN4G) to learn from graph inputs with feedforward neural networks.

Most GK- and RNN-based approaches have been applied to relatively small graphs,

such as those derived from molecules [Ralaivola et al., 2005, Bianucci et al., 2000, Borgwardt and Kriegel, 2005], natural language sentences [Socher et al., 2011] or protein structures [Vullo and Frasconi, 2004, Baldi and Pollastri, 2003, Borgwardt et al., 2005]. On the other hand, large graphs (especially social networks) typically exhibit a highly skewed degree distribution that originates a huge vocabulary of distinct subgraphs. This scenario makes finding a suitable representation much harder: kernels based on subgraph matching would suffer diagonal dominance [Schoelkopf et al., 2002], while RNNs would face the problem of composing a highly variable number of substructure representations in the recursive step. Recent work by Yanardag and Vishwanathan [2015] proposes deep graph kernels (DGK) to upgrade existing graph kernels with a feature reweighing schema that employs CBOW/Skip-gram embedding of the substructures. Another recent work by Niepert et al. [2016] casts graphs into a format suitable for learning with convolutional neural networks (CNNs). These methods have been applied successfully to small graphs but also to graphs derived from social networks.

In this chapter, we introduce a novel architecture for machine learning with structured inputs, called shift-aggregate-extract network (SAEN). First structured inputs are decomposed into hierarchical decompositions called $\mathcal{H}$-decompositions (see § 4.1) second a feedforward neural network is *unfolded* over the hierarchical decompositions using *shift*, *aggregate* and *extract* operations (see § 4.2) and third we perform learning by gradient descent.

Like the flat $\mathcal{R}$-decompositions commonly used to define kernels on structured data [Haussler, 1999], $\mathcal{H}$-decompositions are based on the *part-of* relation, but allow us to introduce a deep recursive notion of *parts of parts*. At the top level of the hierarchy lies the *whole* data structure. Objects at each intermediate level are decomposed into parts that form the subsequent level of the hierarchy. The bottom level consists of atomic objects, such as individual vertices or edges of a graph.

SAEN compensates some limitations of recursive neural networks by adding two synergetic degrees of flexibility. First, it unfolds a neural network over a hierarchy of parts rather than using the edge set of the input graph directly; this makes it easier to deal with very high degree vertices. Second, it imposes weight sharing and fixed size of the learned vector representations on a per level basis instead of globally; in this way, more complex parts may be embedded into higher dimensional vectors, without forcing to use excessively large representations for simpler parts.

A second contribution of this work is a *domain compression* algorithm that can significantly reduce memory usage and runtime. It leverages mathematical results from lifted linear programming [Mladenov et al., 2012] in order to exploit symmetries and perform a lossless compression of $\mathcal{H}$-decompositions.

The chapter is organized as follows. In § 4.1 we introduce $\mathcal{H}$-decompositions, a generalization of Haussler's [Haussler, 1999] $\mathcal{R}$-decomposition relations. In § 4.2 we

describe SAEN, a neural network architecture for learning vector representations of $\mathcal{H}$-decompositions. Furthermore, in § 4.3 we explain how to exploit symmetries in $\mathcal{H}$-decompositions in order to reduce memory usage and runtime. In § 4.4 we report experimental results on several number of real-world datasets. Finally, in § 4.5 we discuss some related works and draw some conclusions in § 4.6.

4.1 $\mathcal{H}$-decompositions

In this section we define a deep hierarchical extension of Haussler's [Haussler, 1999] $\mathcal{R}$-decomposition relation called $\mathcal{H}$-decomposition.

An $\mathcal{H}$-decomposition represents structured data as a hierarchy of π-parametrized parts. It is formally defined as the triple $(\{S_l\}_{l=0}^{L}, \{\mathcal{R}_{l,\pi}\}_{l=1}^{L}, X)$ where:

- $\{S_l\}_{l=0}^{L}$ are disjoint sets of objects S_l called levels of the hierarchy. The bottom level S_0 contains atomic (i.e. non-decomposable) objects, while the other levels $\{S_l\}_{l=1}^{L}$ contain compound objects, $s \in S_l$, whose parts $s' \in S_{l-1}$ belong to the preceding level, S_{l-1}.
- $\{\mathcal{R}_{l,\pi}\}_{l=1}^{L}$ is a set of l, π-parametrized $\mathcal{R}_{l,\pi}$-convolution relations, where $\pi \in \Pi_l$ is a membership type from a finite alphabet Π_l of size $n(l) = |\Pi_l|$. At the bottom level, $n(0) = 1$. A pair $(s, s') \in S_l \times S_{l-1}$ belongs to $\mathcal{R}_{l,\pi}$ iff s' is part of s with membership type π. For notational convenience, the parts of s are denoted as $\mathcal{R}_{l,\pi}^{-1}(s) = \{s' | (s', s) \in \mathcal{R}_{l,\pi}\}$.
- X is a set $\{\mathbf{x}(s)\}_{s \in S_0}$ of p-dimensional vectors of attributes assigned to the elements s the bottom layer S_0.

The membership type π is used to represent the roles of the parts of an object. For example, we could decompose a graph as a multiset of π-neighborhood subgraphs[1] in which π is the radius of the neighborhoods. Another possible use of the π-membership type is to distinguish the root from the other vertices in a rooted neighborhood subgraph. Both uses of π-membership type are shown in Figure 4.6.

An $\mathcal{H}$-decomposition is a multilevel generalization of $\mathcal{R}$-convolution relations, and it reduces to an $\mathcal{R}$-convolution relation for $L = 1$.

Example 1. *For example we could produce a* 4*-level decomposing graph* $Graph \in S_3$ *into a set of* $radius$*-neighborhood (*$radius \in \{1, 2\}$*) subgraphs* $Ball \in S_2$ *and employ their* $radius$ *as membership type. Furthermore, we can extract*

[1] The r-neighborhood subgraph (or ego graph) of a vertex v in a graph G is the induced subgraph of G consisting of all vertices whose shortest-path distance from v is at most r.

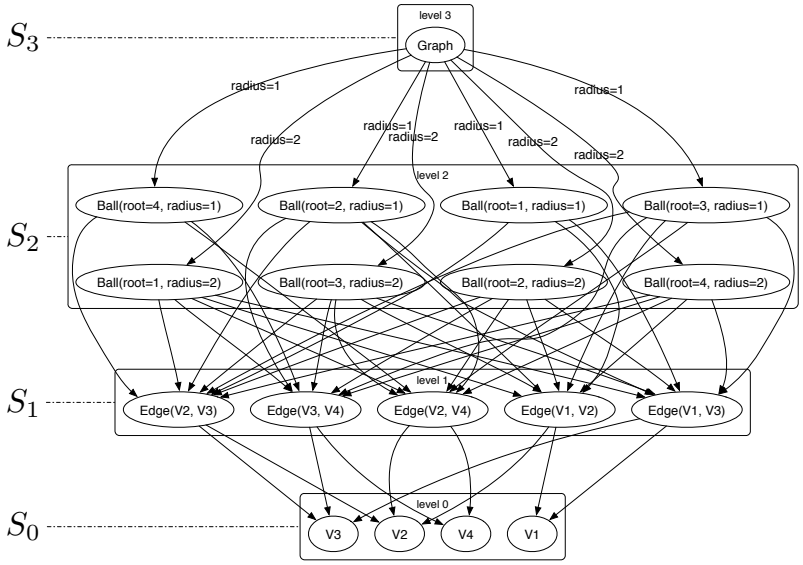


Figure 4.1: Pictorial representation of the $\mathcal{H}$-decomposition of Example 1. We produce a 4-level $\mathcal{H}$-decomposing graph $Graph \in S_3$ into a set of $radius$-neighborhood ($radius \in \{1, 2\}$) subgraphs $Ball \in S_2$ and employ their $radius$ as membership type. Furthermore, we extract edges $Edge \in S_1$ from the $radius$-neighborhood subgraphs. Finally, each edge is decomposed in vertices $V \in S_0$. The elements of the $\mathcal{R}_{l,\pi}$-convolution are pictorially shown as directed arcs. Since membership types π for edges and vertices would be all identical their label is not represented in the picture.

edges $Edge \in S_1$ from the $radius$-neighborhood subgraphs. Finally, each edge $Edge \in S_1$ could be decomposed in vertices $V \in S_0$. In Figure 4.1 we provide a pictorial representation of the above $\mathcal{H}$-decomposition applied to an example graph, while in Figure 4.2 we show the substructure that is contained in each node of the $\mathcal{H}$-decomposition in Figure 4.1.

Another example of decomposition could come from text processing, documents $d \in S_3$ could be decomposed in sentences $s \in S_2$ which are themselves represented as graphs of dependency relations and further decomposed as bags of shortest paths $p \in S_1$ in the dependency graph. Finally, the words $w \in S_0$ (which are the vertices of the dependency graph) constitute the bottom layer and can be represented in attributed form as word vectors.

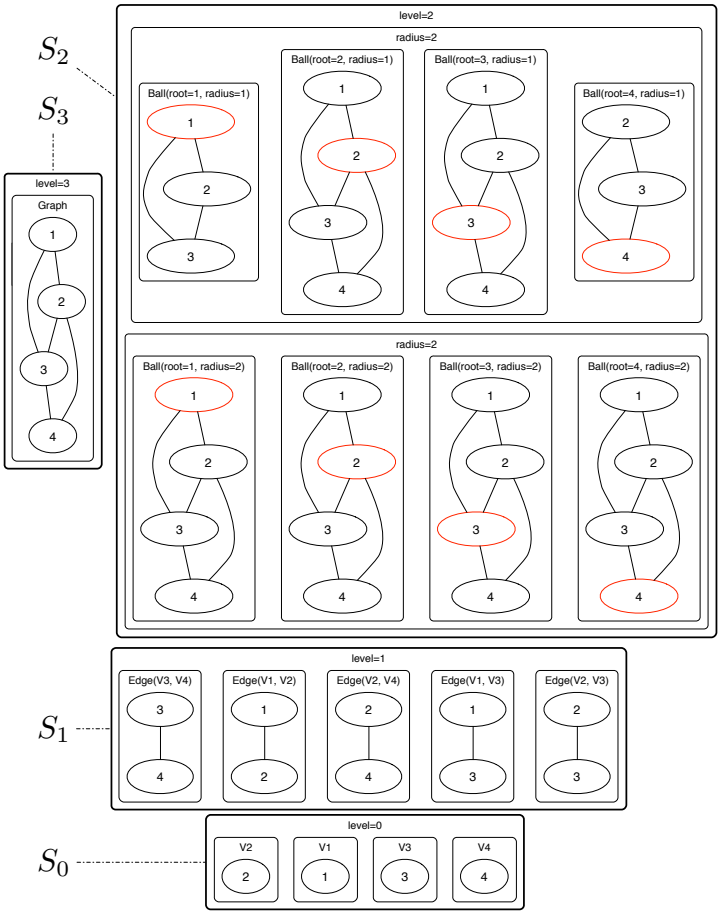


Figure 4.2: Pictorial representation of the substructures that are contained in each node of the $\mathcal{H}$-decomposition explained in Example 1 and showed in Figure 4.1. The objects of the $\mathcal{H}$-decomposition are grouped to according their S_l sets ($l = 0, \ldots, 3$). For each $radius$-neighborhood subgraph we show the root node in red.

4.2 Learning representations with SAEN

A shift-aggregate-extract network (SAEN) is a composite function that maps objects at level l of an $\mathcal{H}$-decomposition into $d(l)$-dimensional real vectors. It uses a sequence of parametrized functions $\{f_0, \ldots, f_L\}$, for example a sequence of neural networks with parameters $\theta_0, \ldots, \theta_L$ that will be trained during the learning. At each level, $l = 0, \ldots, L$, each function $f_l : \mathbb{R}^{n(l)d(l)} \to \mathbb{R}^{d(l+1)}$ operates as follows:

1. It receives as input the *aggregate* vector $\mathbf{a}_l(s)$ defined as:

$$\mathbf{a}_l(s) = \begin{cases} \mathbf{x}(s) & \text{if } l = 0 \\ \sum_{\pi \in \Pi_l} \sum_{s' \in \mathcal{R}_{l,\pi}^{-1}(s)} \mathbf{z}_\pi \otimes \mathbf{h}_{l-1}(s') & \text{if } l > 0 \end{cases} \tag{4.1}$$

   where $\mathbf{x}(s)$ is the vector of attributes for object s.

2. It *extracts* the vector representation of s as

$$\mathbf{h}_l(s) = f_l(\mathbf{a}_l(s); \theta_l). \tag{4.2}$$

The vector $\mathbf{a}_l(s)$ is obtained in two steps: first, previous level representations $\mathbf{h}_{l-1}(s')$ are *shifted* via the Kronecker product $\otimes$ using an indicator vector $\mathbf{z}_\pi \in \mathbb{R}^{n(l)}$. This takes into account of the membership types π. Second, shifted representations are *aggregated* with a sum. Note that all representation sizes $d(l)$, $l > 0$ are hyper-parameters that need to be chosen or adjusted.

The shift and aggregate steps are identical to those used in kernel design when computing the explicit feature of a kernel $k(x, z)$ derived from a sum $\sum_{\pi \in \Pi} k_\pi(x, z)$ of base kernels $k_\pi(x, z),\ \pi \in \Pi$. In principle, it would be indeed possible to turn SAEN into a kernel method by removing the extraction step and define the explicit feature for a kernel on $\mathcal{H}$-decompositions. Removing the extraction step from Eq. 4.1 results in:

$$\mathbf{a}_l(s) = \begin{cases} \mathbf{x}(s) & \text{if } l = 0 \\ \sum_{\pi \in \Pi_l} \sum_{s' \in \mathcal{R}_{l,\pi}^{-1}(s)} \mathbf{z}_\pi \otimes \mathbf{a}_{l-1}(s') & \text{if } l > 0 \end{cases} \tag{4.3}$$

However, that approach would increase the dimensionality of the feature space by a multiplicative factor $n(l)$ for each level l of the $\mathcal{H}$-decomposition, thus leading to an exponential number of features. When the number of features is exponential, their explicit enumeration is impractical. A possible solution would be to directly define the kernel similarity and keep the features implicit. However, this solution would have space complexity that is quadratic in the number of graphs in the dataset.

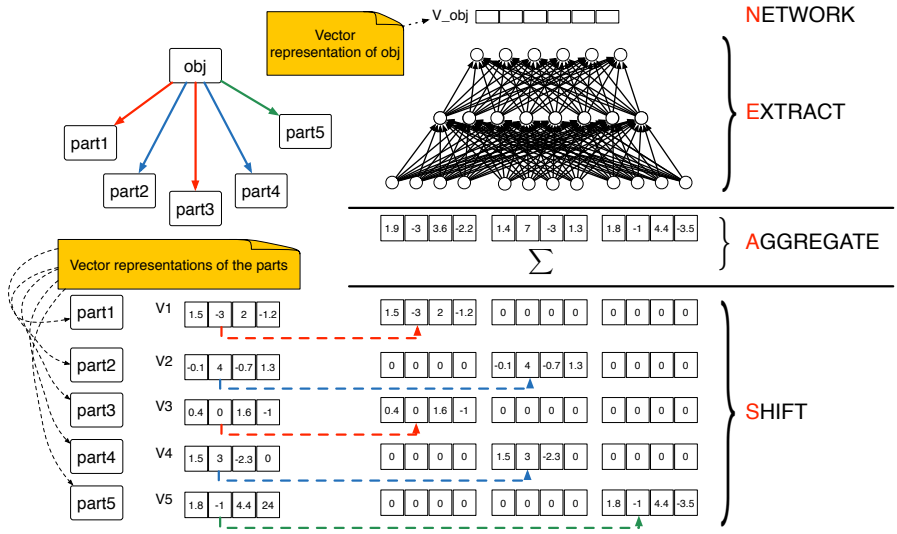


Figure 4.3: Pictorial representation of the SAEN computation explained in Eq. 4.1 and Eq. 4.2. The SAEN computation is unfolded over all the levels of an $\mathcal{H}$-decomposition. On the top-right part we show an object $obj \in S_l$ decomposed into its parts $\{part_i\}_{i=1}^{5} \subseteq S_{l-1}$ from the level below. The parametrized "part of" relation $\mathcal{R}_{l,pi}$ is represented by directed arrows, we use colors (red, blue and green) to distinguish among π-types. In the bottom-left part of the picture we show that each part is associated to a vectorial representation. In the bottom-right part of the picture we show the *shift* step in which the vector representations of the parts are shifted using the Kronecker product in Eq. 4.1. Then the shifted representation are summed in the aggregation step and in the extract step a feedforward neural is applied in order to obtain the vector representation of object obj.

When using SAEN, the feature space growth is prevented by exploiting a distributed representation (via a multilayered neural network) during the extraction step. As a result, SAEN can easily cope with $\mathcal{H}$-decompositions consisting of multiple levels.

4.3 Exploiting symmetries for domain compression

In this section we propose a technique, called *domain compression*, which allows us to save memory and speed up the SAEN computation. Domain compression exploits symmetries in $\mathcal{H}$-decompositions to compress them without information loss. This technique requires that the attributes $\mathbf{x}(s)$ of the elements s in the bottom level S_0 are categorical.

Definition 7. *Two objects a, b in a level S_l are collapsible, denoted $a \sim b$, if they share the same representation, i.e., $\mathbf{h}_l(a) = \mathbf{h}_l(b)$ for all the possible values of the parameters $\theta_0, \ldots, \theta_l$.*

According to Definition 7, objects in the bottom level S_0 are collapsible when their attributes are identical, while objects at any level $\{S_l\}_{l=1}^{L}$ are collapsible if they are made of the same sets of parts for all the membership types π.

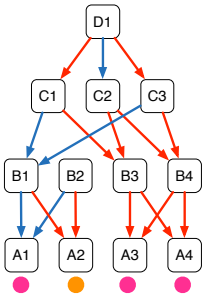
A compressed level S_l^{comp} is the quotient set of level S_l with respect to the collapsibility relation $\sim$.

Before providing a mathematical formulation of domain compression we provide two examples: in Example 2 we explain the intuition beyond domain showing in Figure 2 the steps that need to be taken to compress a $\mathcal{H}$-decomposition, in Example 3 we provide a pictorial representation of the $\mathcal{H}$-decomposition of a real world graph and its compressed version.

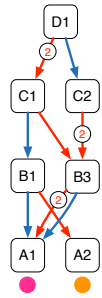
Example 2. *Figure 4.4 a) shows the pictorial representation of an $\mathcal{H}$-decomposition whose levels are denoted with the letters of the alphabet A, B, C, D. We name each object using consecutive integers and the name of the level as prefix. We use purple and orange circles to denote the categorical attributes of the objects of the bottom stratum. Directed arrows denote the "part of" relations whose membership type is distinguished using the colors blue and red.*

Figure 4.4 b) shows the domain compression of the $\mathcal{H}$-decomposition in a). When objects are collapsed the directed arcs coming from their parents are also collapsed. Collapsed arcs are labeled with their cardinality.

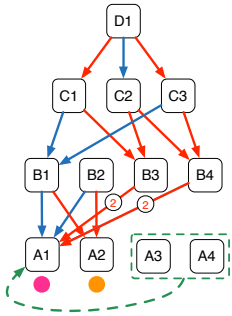
Figures 4.4 c), d), e) and f) describe the domain compression steps starting from level A until level D.



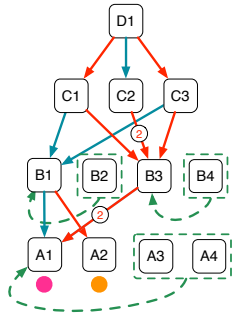
a)
$\mathcal{H}$-decomposition



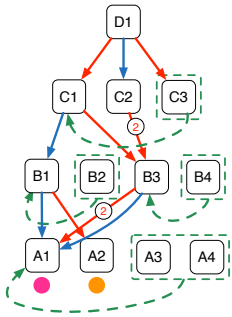
b)
compressed $\mathcal{H}$-decomposition



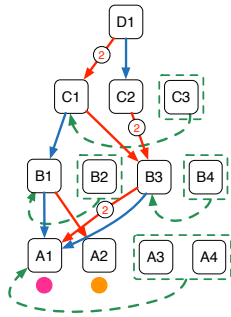
c)
compression of level A



d)
compression of level B



e)
compression of level C



f)
compression of level D

Figure 4.4: Intuition of the domain compression algorithm explained in Example 2.

- *Figure 4.4 c) shows that since* `A3` *and* `A4` *have the same categorical attribute of* `A1` *(i.e. purple) they are grouped and collapsed to* `A1`*. Furthermore, the arrows in the fan-in of* `A3` *and* `A4` *are attached to* `A1` *with the consequent cardinality increase of the red arrows that come from* `B3` *and* `B4`*.*
- *In Figure 4.4 d) we show the second iteration of domain compression in which objects made of the same parts with the same membership types are collapsed. Both* `B1` *and* `B2` *in Figure 4.4 c) were connected to* `A1` *with a blue arrow and to* `A2` *with a red arrow and so they are collapsed. In the same way* `B3` *and* `B4` *are collapsed because in c) they were connected to* `A1` *with a red arrow with cardinality 2.*
- *In Figure 4.4 e)* `C1` *and* `C3` *are collapsed because in d) they were both connected to* `B1` *with a blue arrow and* `B3` *with a red arrow.*
- *Finally in f) since* `C1` *and* `C3` *were collapsed in the previous step we increase to* 2 *the cardinality of the red arrow that connects* `D1` *and* `C1` *and remove the red arrow from* `D1` *to* `C3` *since* `C3` *was collapsed to* `C1` *in Figure 4.4 e).*

The final result of domain compression is illustrated in Figure 4.4 b).

Example 3. *In Figure 4.5 we provide a pictorial representation of the domain compression of an $\mathcal{H}$-decomposition (*EGNN*, described in § 4.4.2). On the left we show the $\mathcal{H}$-decomposition of a graph taken from the* IMDB-BINARY *dataset (see § 4.4.1) together with its compressed version on the right.*

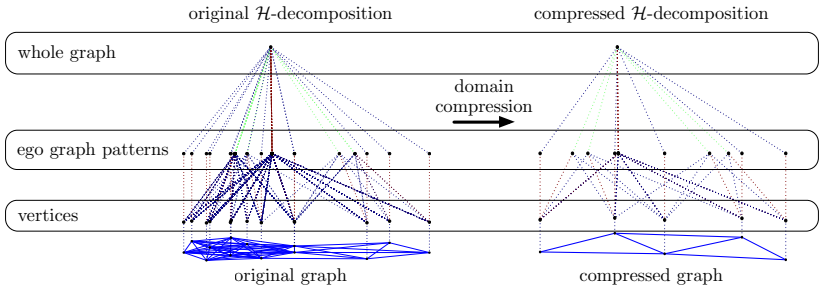


Figure 4.5: Pictorial representation of the $\mathcal{H}$-decomposition of a graph taken from the IMDB-BINARY dataset (see § 4.4.1) together with its compressed version.

In order to compress $\mathcal{H}$-decompositions we adapt the lifted linear programming technique proposed by [Mladenov et al., 2012] to the SAEN architecture. A matrix $M \in \mathbb{R}^{n \times p}$ with $m \leq n$ distinct rows can be decomposed as the product DM^{comp} where M^{comp} is a compressed version of M in which the distinct rows of M appear exactly once.

Definition 8. *The Boolean decompression matrix, D, encodes the collapsibility relation among the rows of M so that $D_{ij} = 1$ iff the i^{th} row of M falls in the equivalence class j of $\sim$, where $\sim$ is the equivalence relation introduced in Definition 7.*[2]

Example 4. *(Example 2 continued)*

The bottom level of the $\mathcal{H}$-decomposition in Figure 4.4 a) has 4 objects `A1`, `A2`, `A3` and `A4` with categorical attributes indicated with colors.

Objects `A1`, `A2`, `A4` have a purple categorical attribute while `A3` has a orange categorical attribute. If we give to purple the encoding $[0, 3]$ and to orange the encoding $[4, 1]$ we obtain an attribute matrix

$$X = \begin{bmatrix} 0 & 3 \\ 0 & 3 \\ 4 & 1 \\ 0 & 3 \end{bmatrix} \tag{4.4}$$

in which each row contains the encoding of the categorical attribute of an object of the bottom stratum and objects were taken with the order `A1`, `A2`, `A3`, `A4`.

Since the rows associated to `A1`, `A3`, `A4` are identical we can compress matrix X to matrix

$$X^{comp} = \begin{bmatrix} 0 & 3 \\ 4 & 1 \end{bmatrix} \tag{4.5}$$

as we can notice this is the attribute matrix of the compressed $\mathcal{H}$-decomposition shown in Figure 4.4 b).

Matrix X can be expressed as the matrix product DX^{comp} between the decompression matrix D and the compressed version of X^{comp} where

$$D = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{4.6}$$

and was obtained applying Definition 8.

As explained in Mladenov et al. [2012] a pseudo-inverse C of D can be computed by dividing the rows of $D^\top$ by their sum (where $D^\top$ is the transpose of D).

However, it is also possible to compute a pseudo-inverse C' of D by transposing D and choosing one representer for each row of $D^\top$. For each row of $D^\top$ we can simply choose a nonzero element as representer and set all the other to zero.

[2] Mladenov et al. [2012] lifts linear programming and defines the equivalence relation induced from the

Example 5. *The computation of the pseudo-inverse C of the D matrix of Example 4 results in the following equation:*

$$C = \begin{bmatrix} 1/3 & 1/3 & 0 & 1/3 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.7}$$

the matrix multiplication between the compression matrix C and the X leads to the compressed matrix X^{comp} (i.e. $X^{comp} = CX$).

In the first row of matrix C there are 3 nonzero entries that correspond to the objects `A1`*,* `A2`*,* `A4`*, while on the second row there is a nonzero entry that corresponds to object* `A3`*.*

As we said above, since we know that the encodings of those objects are identical instead of making the average we could just take a representer.

For example in Figure 4.4 c) we chose `A1` *as representer for* `A2` *and* `A4`*, obtaining the compression matrix*

$$C' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{4.8}$$

In the first row of matrix C' there is a nonzero entry that correspond to the object `A1` *(which is the chosen representer), while on the second row there is a nonzero entry that corresponds to object* `A3` *(as in C).*

While from the compression point of view we still have $X^{comp} = C'X$, choosing a representer instead of averaging equivalent objects is advantageous when using sparse matrices because the number of nonzero elements decreases.

We apply domain compression to SAEN by rewriting Eqs. 4.1 and 4.2 in matrix form.

We rewrite Eq. 4.1 as:

$$A_l = \begin{cases} X & \text{if } l = 0 \\ \mathbf{R}_l \mathbf{H}_{l-1} & \text{if } l > 0 \end{cases} \tag{4.9}$$

where:

- $A_l \in \mathbb{R}^{|S_l| \times n(l-1)d(l)}$ is the matrix that represents the *shift-aggregated* vector representations of the object of level S_{l-1};
- $X \in \mathbb{R}^{|S_0| \times p}$ is the matrix that represents the p-dimensional encodings of the vertex attributes in V (i.e. the rows of X are the $\mathbf{x}_{v_i}$ of Eq. 4.1);
- $\mathbf{R}_l \in \mathbb{R}^{|S_l| \times n(l)|S_{l-1}|}$ is the concatenation

$$\mathbf{R}_l = \left[R_{l,1}, \ldots, R_{l,\pi}, \ldots, R_{l,n(l)}\right] \tag{4.10}$$

labels obtained by performing color passing on a Gaussian random field. We use an the equivalence relation

of the matrices $R_{l,\pi} \in \mathbb{R}^{|S_l| \times |S_{l-1}|}\ \forall \pi \in \Pi_l$ which represent the $\mathcal{R}_{l,\pi}$-convolution relations of Eq. 4.1 whose elements are $(R_{l,\pi})_{ij} = 1$ if $(s', s) \in \mathcal{R}_{l,\pi}$ and 0 otherwise.

- $\mathbf{H}_{l-1} \in \mathbb{R}^{n(l)|S_{l-1}| \times n(l)d(l)}$ is a block-diagonal matrix

$$\mathbf{H}_{l-1} = \begin{bmatrix} H_{l-1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & H_{l-1} \end{bmatrix} \tag{4.11}$$

whose blocks are formed by matrix $H_{l-1} \in \mathbb{R}^{|S_{l-1}| \times d(l)}$ repeated $n(l)$ times. The rows of H_{l-1} are the vector representations $\mathbf{h}_j$ in Eq. 4.1.

Eq. 4.2 is simply rewritten to $H_l = f_l(A_l; \theta_l)$ where $f_l(\cdot; \theta_l)$ is unchanged w.r.t. Eq. 4.2 and is applied to its input matrix A_l row-wise.

Algorithm 1 DOMAIN-COMPRESSION

DOMAIN-COMPRESSION(X, R)
1 $C_0, D_0 =$ COMPUTE-CD(X)
2 $X^{comp} = C_0 X$
3 $R^{comp} = \{\}$
4 **for** $l = 1$ **to** L
5 $\quad R^{col\_comp} = [R_{l,\pi} D_{l-1},\ \forall \pi = 1, \dots, n(l)]$
6 $\quad C_l, D_l =$ COMPUTE-CD(R^{col_comp})
7 $\quad$ **for** $\pi = 1$ **to** $n(l)$
8 $\quad\quad R^{comp}_{l,\pi} = C_l R^{col\_comp}_{\pi}$
9 **return** X^{comp}, R^{comp}

Domain compression on Eq. 4.9 is performed by the DOMAIN-COMPRESSION procedure (see Algorithm 1). which takes as input the attribute matrix $X \in \mathbb{R}^{|S_0| \times p}$ and the part-of matrices $R_{l,\pi}$ and returns their compressed versions X^{comp} and the $R^{comp}_{l,\pi}$ respectively. The algorithm starts by invoking (line 1) the procedure COMPUTE-CD on X to obtain the compression and decompression matrices C_0 and D_0 respectively. The compression matrix C_0 is used to compress X (line 2) then we start iterating over the levels $l = 0, \dots, L$ of the $\mathcal{H}$-decomposition (line 4) and compress the $R_{l,\pi}$ matrices. The compression of the $R_{l,\pi}$ matrices is done by right-multiplying them by the decompression matrix D_{l-1} of the previous level $l - 1$ (line 5). In this way we collapse the parts of relation $\mathcal{R}_{l,\pi}$ (i.e. the columns of $R_{l,\pi}$) as these

in Definition 7 because we are working with $\mathcal{H}$-decompositions.

were identified in level S_{l-1} as identical objects (i.e. those objects corresponding to the rows of X or $R_{l-1,\pi}$ collapsed during the previous step). The result is a list $R^{col\_comp} = [R_{l,\pi}D_{l-1},\ \forall\pi = 1,\dots,n(l)]$ of column compressed $R_{l,\pi}-$matrices. We proceed collapsing equivalent objects in level S_l, i.e. those made of identical sets of parts: we find symmetries in R^{col_comp} by invoking COMPUTE-CD (line 6) and obtain a new pair C_l, D_l of compression, and decompression matrices respectively. Finally the compression matrix C_l is applied to the column-compressed matrices in R^{col_comp} in order to obtain the Π_l compressed matrices of level S_l (line 8).

Algorithm 1 allows us to compute the domain compressed version of Eq. 4.9 which can be obtained by replacing: X with $X^{comp} = C_0X$, $R_{l,\pi}$ with $R^{comp}_{l,\pi} = C_lR_{l,\pi}D_{l-1}$ and H_l with H^{comp}_l. Willing to recover the original encodings H_l we just need to employ the decompression matrix D_l on the compressed encodings H^{comp}_l, indeed $H_l = D_lH^{comp}_l$.

As we can see by substituting S_l with S^{comp}_l, the more are the symmetries (i.e. when $|S^{comp}_l| \ll |S_l|$) the greater the domain compression will be.

4.4 Experimental evaluation

We perform an experimental evaluation of SAEN on graph classification datasets and answer the following questions:

Q1 How does SAEN compare to the state of the art?

Q2 Can SAEN exploit symmetries in social networks to reduce the memory usage and the runtime?

4.4.1 Datasets

In order to answer the experimental questions we tested our method on six publicly available datasets first proposed by Yanardag and Vishwanathan [2015] and some bioinformatic datasets.

• **COLLAB** is a dataset where each graph represent the ego-network of a researcher, and the task is to determine the field of study of the researcher between *High Energy Physics*, *Condensed Matter Physics* and *Astro Physics*.

• **IMDB-BINARY**, **IMDB-MULTI** are datasets derived from IMDB where in each graph the vertices represent actors/actresses and the edges connect people which have performed in the same movie. Collaboration graphs are generated from movies belonging to genres *Action* and *Romance* for IMDB-BINARYand *Comedy*, *Romance* and *Sci-Fi* for IMDB-MULTI, and for each actor/actress in those genres an ego-graph

Table 4.1: Statistics of the datasets used in our experiments. For each dataset we report the number of nodes, the average number of nodes and average maximum node degree (AMND).

DATASET	SIZE	AVG. NODES	AVG. MAX. DEGREE
COLLAB	5000	74.49	73.62
IMDB-BINARY	1000	19.77	18.77
IMDB-MULTI	1500	13.00	12.00
REDDIT-BINARY	2000	429.62	217.35
REDDIT-MULTI5K	5000	508.51	204.08
REDDIT-MULTI12K	11929	391.40	161.70
MUTAG	188	17.93	3.01
PTC	344	25.56	3.73
NCI1	4110	29.87	3.34
PROTEINS	1113	39.06	5.79
D&D	1178	284.32	9.51

is extracted. The task is to identify the genre from which the ego-graph has been generated.

• **REDDIT-BINARY**, **REDDIT-MULTI5K**, **REDDIT-MULTI12K** are datasets where each graph is derived from a discussion thread from Reddit. In those datasets each vertex represent a distinct user and two users are connected by an edge if one of them has responded to a post of the other in that discussion. The task in REDDIT-BINARY is to discriminate between threads originating from a discussion-based subreddit (*TrollXChromosomes*, *atheism*) or from a question/answers-based subreddit (*IAmA*, *AskReddit*). The task in REDDIT-MULTI5K and REDDIT-MULTI12K is a multiclass classification problem where each graph is labeled with the subreddit where it has originated (*worldnews, videos, AdviceAnimals, aww, mildlyinteresting* for REDDIT-MULTI5K and *AskReddit, AdviceAnimals, atheism, aww, IAmA, mildlyinteresting, Showerthoughts, videos, todayilearned, worldnews, TrollXChromosomes* for REDDIT-MULTI12K).

• **MUTAG**, **PTC**, **NCI1**, **PROTEINS** and **D&D** are bioinformatic datasets. MUTAG [Debnath et al., 1991] is a dataset of 188 mutagenic aromatic and heteroaromatic nitro compounds labeled according to whether or not they have a mutagenic effect on the Gramnegative bacterium *Salmonella typhimurium*. PTC [Toivonen et al., 2003] is a dataset of 344 chemical compounds that reports the carcinogenicity for male and female rats and it has 19 discrete labels. NCI1 [Wale et al., 2008] is a dataset of 4100 examples and is a subset of balanced datasets of chemical compounds screened for ability to suppress or inhibit the growth of a panel of human tumor cell lines, and has 37 discrete labels. PROTEINS [Borgwardt et al., 2005] is a binary classification dataset made of 1113 proteins. Each protein is represented as a graph where nodes are secondary structure elements (i.e. helices, sheets and turns). Edges connect nodes if they are neighbors in the amino-acid sequence or in the 3D space. D&D is a binary

Table 4.2: Comparison of accuracy results on social network datasets. The classification accuracy of SAEN was measured with 10-times 10-fold cross-validation. We obtained 10 accuracy values (one for each 10-fold cross-validation run) and reported their mean and standard deviation. The results for DGK and PATCHY-SAN where taken from the papers of Yanardag and Vishwanathan [2015] and Niepert et al. [2016] respectively (see § 4.5 for a discussion of the related works).

DATASET	DGK [Yanardag and Vishwanathan, 2015]	PATCHY-SAN [Niepert et al., 2016]	SAEN (our method)
COLLAB	73.09 ± 0.25	72.60 ± 2.16	75.63 ± 0.31
IMDB-BINARY	66.96 ± 0.56	71.00 ± 2.29	71.26 ± 0.74
IMDB-MULTI	44.55 ± 0.52	45.23 ± 2.84	49.11 ± 0.64
REDDIT-BINARY	78.04 ± 0.39	86.30 ± 1.58	86.08 ± 0.53
REDDIT-MULTI5K	41.27 ± 0.18	49.10 ± 0.70	52.24 ± 0.38
REDDIT-MULTI12K	32.22 ± 0.10	41.32 ± 0.42	46.72 ± 0.23

Table 4.3: Parameters used for the EGNN decompositions for each datasets.

DATASET	RADIUSES	HIDDEN UNITS		
	r	S_0	S_1	S_2
COLLAB	$0, 1$	$15-5$	$5-2$	$5-3$
IMDB-BINARY	$0, 1, 2$	2	$5-2$	$5-3-1$
IMDB-MULTI	$0, 1, 2$	2	$5-2$	$5-3$
REDDIT-BINARY	$0, 1$	$10-5$	$5-2$	$5-3-1$
REDDIT-MULTI5K	$0, 1$	10	10	$6-5$
REDDIT-MULTI12K	$0, 1$	10	10	$20-11$
MUTAG	$0, 1, 2, 3$	10	$5-5$	$5-5-1$
PTC	$0, 1$	15	15	$15-1$
NCI1	$0, 1, 2, 3$	15	15	$15-10-1$
PROTEINS	$0, 1, 2, 3$	$3-2$	$6-5-4$	$6-3-1$
D&D	$0, 1, 2, 3$	10	$5-2$	$5-3-1$

classification dataset of 1178 graphs. Each graph represents a protein nodes are amino acids which are connected by an edge if they are less than 6 Angstroms apart.

4.4.2 Experiments

In our experiments we chose an $\mathcal{H}$-decomposition called Ego Graph Neural Network (EGNN) (shown in Figure 4.6), that mimics the graph kernel NSPDK with the distance parameter set to 0. Before applying EGNN we turn unattributed graphs (V, E) into attributed graphs (V, E, X) by annotating their vertices $v \in V$ with attributes $\mathbf{x}_v \in X$. We label vertices v of G with their degree and encode this information into the attributes $\mathbf{x}_v$ by employing the 1-hot encoding.

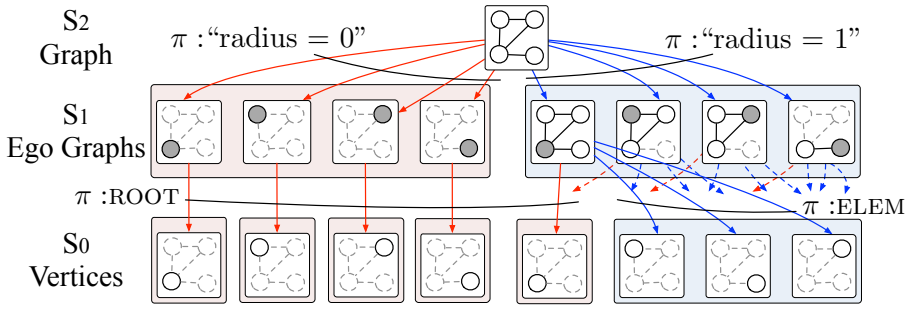


Figure 4.6: Example of Ego Graph decomposition.

Table 4.4: Comparison of accuracy on bio-informatics datasets. The classification accuracy of SAEN was measured with 10-times 10-fold cross-validation. We obtained 10 accuracy values (one for each 10-fold cross-validation run) and reported their mean and standard deviation. The results for PATCHY-SAN where taken from the paper of Niepert et al. [2016].

DATASET	PATCHY-SAN [Niepert et al., 2016]	SAEN (our method)
MUTAG	92.63 ± 4.21	84.99 ± 1.82
PTC	62.29 ± 5.68	57.04 ± 1.30
NCI1	78.59 ± 1.89	77.80 ± 0.42
PROTEINS	75.89 ± 2.76	75.31 ± 0.70
D&D	77.12 ± 2.41	77.69 ± 0.96

EGNN decomposes attributed graphs $G = (V, E, X)$ into a 3 level $\mathcal{H}$-decomposition with the following levels:

- level S_0 contains objects s_v that are in one-to-one correspondence with the vertices $v \in V$.
- level S_1 contains v_{root}-rooted r-neighborhood subgraphs (i.e. ego graphs) $e = (v_{root}, V_e, E_e)$ of radius $r = 0, 1, \ldots, R$ and has part-of alphabet $\Pi_1 = \{\text{ROOT}, \text{ELEM}\}$. Objects $s_v \in S_0$ are “ELEM-part-of” ego graph e if $v \in V_e \setminus \{v_{root}\}$, while the are “ROOT-part-of” ego graph e if $v = v_{root}$.
- level S_2 contains the graph G that we want to classify and has part-of alphabet $\Pi_2 = \{0, 1\}$ which correspond to the radius of the ego graphs $e \in S_1$ of which G is made of.

The EGNN decomposition is exemplified for a small graph shown in Figure 4.6.

E1 We experiment with SAEN applying the EGNN $\mathcal{H}$-decomposition on all the datasets. In order to perform classification we add a cross-entropy loss on the extraction step $h_L(s)$ (see Eq. 4.2) of the top level L (i.e. $L = 2$) of the EGNN $\mathcal{H}$-decomposition. We used Leaky RELUs [Maas et al., 2013] as activation function on all the units of the neural networks $\{f_l(.;\Theta_l)\}_{l=0}^2$ of the extraction step (cf. Eq. 4.2).

SAEN was implemented in TensorFlow and in all our experiments we trained the neural network parameters $\{\Theta_l\}_{l=0}^2$ by using the Adam algorithm [Kingma and Ba, 2014] to minimize a cross-entropy loss, we used 0.001 as learning rate.

The classification accuracy of SAEN was measured with 10-times 10-fold cross-validation. We obtained 10 accuracy values (one for each 10-fold cross-validation run). For each social network dataset we report the mean and the standard deviation of these accuracy values in Table 4.2 where we compare our results with those by Yanardag and Vishwanathan [2015] and by Niepert et al. [2016]. In Table 4.4 we compare the results obtained by our method on bioinformatic datasets with those obtained by Niepert et al. [2016] reporting mean and the standard deviation obtained with the same statistical protocol.

With respect to the selection of the hyper-parameters we chose the number of layers and units for each level of the part-of decomposition, the size of each layer and the maximum radius R by training on $8/9^{\text{th}}$ of the training set of the first split of the 10-times 10-fold cross-validation and using as validation set the remaining $1/9^{\text{th}}$ to evaluate the chosen parameters. In Table 4.3 we report for each dataset the radiuses r of the neighborhood subgraphs used in the EGNN decomposition and the number of units in the hidden layers for each level.

E2 In Table 4.5 we show the file sizes of the preprocessed datasets before and after the compression together with the data compression ratio. [3] We also estimate the benefit of domain compression from a computational time point of view and report the measurement of the runtime for 1 run with and without compression together with the speedup factor.

For the purpose of this experiment, all tests were run on a computer with two 8-cores Intel Xeon E5-2665 processors and 94 GB RAM. Uncompressed datasets which exhausted our server's memory during the test are marked as "OOM" (out of memory) in the table, while those who exceeded the time limit of 100 times the time needed for the uncompressed version are marked as "TO" (timeout).

[3] The size of the uncompressed files are shown for the sole purpose of computing the data compression ratio. Indeed the last version of our code compresses the files on the fly.

Table 4.5: Comparison of sizes and runtimes of the datasets before and after the compression.

DATASET	SIZE (MB)			RUNTIME		
	ORIGINAL	COMP.	RATIO	ORIGINAL	COMP.	SPEEDUP
COLLAB	1190	448	0.38	43' 18"	8' 20"	5.2
IMDB-BINARY	68	34	0.50	3' 9"	0' 30"	6.3
IMDB-MULTI	74	40	0.54	7' 41"	1' 54"	4.0
REDDIT-BINARY	326	56	0.17	TO	2' 35"	≥ 100.0
REDDIT-MULTI5K	952	162	0.17	OOM	9' 51"	–
REDDIT-MULTI12K	1788	347	0.19	OOM	29' 55"	–

4.4.3 Discussion

A1 As shown in Table 4.2, EGNN performs consistently better than the other two methods on all the social network datasets. This confirms that the chosen $\mathcal{H}$-decomposition is effective on this kind of problems. Table 4.1 shows that the average maximum node degree (AMND) [4] of the social network datasets is in the order of 10^2. SAEN can easily cope with highly skewed node degree distributions by aggregating distributed representation of patterns while this is not the case for DGK and PATCHY-SAN. DGK uses the same patterns of the corresponding non-deep graph kernel used to match common substructures. If the pattern distribution is affected by the degree distribution most of those patterns will not match, making it unlikely for DGK to work well on social network data. PATCHY-SAN employs as patterns neighborhood subgraphs truncated or padded to a size k in order to fit the size of the receptive field of a CNN. However, since Niepert et al. [2016] experiment with $k = 10$, it is not surprising that they perform worst than SAEN on COLLAB, IMDB-MULTI, REDDIT-MULTI5K and REDDIT-MULTI12K since a small k causes the algorithm to throw away most of the subgraph; a more sensible choice for k would have been the AMND of each graph (i.e. 74, 12, 204 and 162 respectively, cf. Tables 4.1 and 4.2).

Table 4.4 compares the results of SAEN with the best PATCHY-SAN instance on chemoinformatics and bioinformatics datasets. SAEN is in line with the results of Niepert et al. [2016] on PROTEINS and D&D, two datasets where the degree is in the order of 10 (see Table 4.1). Small molecules, on the other hand, have very small degrees. Indeed, in NCI1, MUTAG and PTC SAEN does not perform very well and is outperformed by PATCHY-SAN, confirming that SAEN is best suited for graphs with large degrees. Incidentally, we note that for small molecules, graph kernels attain even better accuracies (e.g. the Weisfeiler-Lehman graph kernel [Shervashidze et al., 2011] achieves 80.13% accuracy on NCI1).

[4] The AMND for a given dataset is obtained by computing the maximum node degree of each graph and then averaging over all graphs.

A2 The compression algorithm has proven to be effective in improving the computational cost of our method. Most of the datasets improved their runtimes by a factor of at least 4 while maintaining the same expressive power. Moreover, experiments on REDDIT-MULTI5K and REDDIT-MULTI12K have only been possible thanks to the size reduction operated by the algorithm as the script exhausted the memory while executing the training step on the uncompressed files.

4.5 Related works

When learning with graph inputs two fundamental design aspects that must be taken into account are: the choice of the pattern generator and the choice of the matching operator. The former decomposes the graph input in substructures while the latter allows to compare the substructures.

Among the patterns considered from the graph kernel literature we have paths, shortest paths, walks [Kashima et al., 2003], subtrees [Ramon and Gärtner, 2003, Shervashidze et al., 2011] and neighborhood subgraphs [Costa and De Grave, 2010]. The similarity between graphs G and G' is computed by counting the number of matches between their common the substructures (i.e. a kernel on the sets of the substructures). The match between two substructures can be defined by using graph isomorphism or some other weaker graph invariant.

When the number of substructures to enumerate is infinite or exponential with the size of the graph (perhaps this is the case for random walks and shortest paths respectively) the kernel between the two graphs is computed without generating an explicit feature map. Learning with an implicit feature map is not scalable as it has a space complexity quadratic in the number of training examples (because we need to store in memory the gram matrix).

Other graph kernels such as the Weisfeiler-Lehman subtree kernel (WLST) [Shervashidze et al., 2011] and the Neighborhood Subgraph Pairwise Distance Kernel (NSPDK) [Costa and De Grave, 2010] deliberately choose a pattern generator that scales polynomially and produces an explicit feature map. However the vector representations produced by WLST and NSPDK are handcrafted and not learned.

Neural networks for graphs (NN4G) [Micheli, 2009] propose a feedforward neural network architecture for $\mathbf{l}$-attributed graphs that first applies a single layer neural network to the vertex attributes $\mathbf{l}(v)$ to produce the an initial encoding $x_1(v)$ for the vertices v in the graph G and then iteratively find new vector representations $x_i(v)$ for the vertices of the input graph G. During the successive iterations the state encoding $x_i(v)$ of a vertex v is obtained by stacking a single neural network layer with sigmoid

activation functions that take as input the continuous attributes $\mathbf{l}(v)$ of v and the state encodings $x_{i'}(u)$ of the neighbors u of v during all the previous iterations $i' < i$. Finally, NN4G can either learn an output representation $y_o(p)$ for the vertices (i.e. $p = v$) or for the whole graph (i.e. $p = G$). While the former is obtained by stacking a single layer neural network over the encoding of the vertices produced across all the iterations, the latter is obtained by aggregating for each iteration i the vertex representations $x_i(v)$ over the vertices v of G, producing a graph representation $X_i(G)$ for each iteration i and then stacking stacking a single layer neural network.

Differently from RNNs, both SAEN and NN4G can learn from graph inputs without imposing weight sharing and using feedforward neural networks. However, while in both NN4G and RNNs the computation is bound to follow the connectivity of the input graph, SAEN has a computation model that follows the connectivity of $\mathcal{H}$-decompositions which can be specified by the user. Moreover, the SAEN user can specify how the vector encoding should be shifted before the aggregation by using the π-membership types of the $\mathcal{H}$-decompositions. Furthermore, SAEN can be trained end-to-end with backpropagation while NN4G was not. Indeed, at each iteration of the computation of a state encoding NN4G *freezes* the weights of the previous iterations.

Deep graph kernels (DGK) [Yanardag and Vishwanathan, 2015] upgrade existing graph kernels with a feature reweighing schema. DGKs represent input graphs as a corpus of substructures (e.g. graphlets, Weisfeiler-Lehman subtrees, vertex pairs with shortest path distance) and then train vector embeddings of substructures with CBOW/Skip-gram models. [5] Each graph-kernel feature (i.e. the number of occurrences of a substructure) is reweighed by the 2-norm of the vector embedding of the corresponding substructure. Experimental evidence shows that DGKs alleviate the problem of diagonal dominance in GKs. However, DGKs inherit from GKs a flat representation (i.e. just one layer of depth) of the input graphs and the vector representations of the substructures are not trained end-to-end as SAEN would do.

PATCHY-SAN [Niepert et al., 2016] casts graphs into a format suitable for learning convolutional neural networks (CNNs): 1) graphs are decomposed into a fixed number of neighborhood subgraphs; 2) which are then casted to a fixed-size receptive field. Both 1) and 2) involve either padding or truncation in order to meet the fixed-size requirements. The truncation operation can be detrimental for the statistical performance of the downstream CNN since it throws away part of the input graph. On the other hand SAEN is able to handle structured inputs of variable sizes without throwing away part of the them. And this is one of the reasons because SAEN has better statistical performance than PATCHY-SAN (See § 4.4).

[5] The CBOW/Skip-gram models receive as inputs cooccurrences among substructures sampled from the input graphs.

4.6 Conclusions

Hierarchical decompositions introduce a novel notion of depth in the context of learning with structured data, leveraging the nested part-of-parts relation. In this chapter, we defined a simple architecture based on neural networks for learning representations of these hierarchies. We showed experimentally that the approach is particularly well-suited for dealing with graphs that are large and have high degree, such as those that naturally occur in social network data. Our approach is also effective for learning with smaller graphs, such as those occurring in chemoinformatics and bioinformatics, although in these cases the performance of SAEN does not exceed the state-of-the-art established by other methods. A second contribution of this chapter is the domain compression algorithm, which greatly reduces memory usage and allowed us to speed up the training time of a factor of at least 4.

Chapter 5

kProbLog: an algebraic Prolog for machine learning

The field of logical and relational learning has already a long tradition, cf. [Sammut, 1993, De Raedt, 2008, Muggleton et al., 2012]. In the 80s and 90s, the goal of this field was to use purely logical and relational representations within machine learning and in this way, provide more expressive representations that allow to represent complex datasets and background knowledge. The key challenge at the time was to tightly integrate these representations with symbolic machine learning methods that were then popular such as rule-learning and decision trees [Van Laer and De Raedt, 2001]. But the field of machine learning has evolved and broadened; in the last two decades it has focussed more on statistical and probabilistic approaches, on kernel and support vector machines and on neural networks. These trends in machine learning have inspired logical and relational learning researchers to extend their goals and to investigate how logical and relational learning principles can be exploited within probabilistic, kernel-based and neural networks.

This is best illustrated by the success of statistical relational learning and probabilistic programming [De Raedt et al., 2016, Getoor and Taskar, 2007], which combine logical and relational learning and programming with probabilistic graphical models. Today there exist many frameworks and formalisms that tightly integrate these two paradigms; they support probabilistic and logical inference as well as learning. Prominent examples include PRISM [Sato and Kameya, 1997], Dyna [Eisner et al., 2004, Eisner and Filardo, 2011], Markov Logic [Richardson and Domingos, 2006], BLOG [Milch et al., 2005], and ProbLog [De Raedt et al., 2007]. Statistical relational learning and probabilistic programming have enabled an entirely new generation of applications.

While there has been a lot of research on integrating probabilistic and logic reasoning,

the combination of kernel-based methods with logic has been much less investigated with the notable exceptions of kLog [Frasconi et al., 2014], kFOIL [Landwehr et al., 2006] and Gärtner et al.'s work [Gärtner et al., 2003, 2004]. kLog is a relational language for specifying kernel-based learning problems. It produces a graph representation of a relational learning problem in the spirit of knowledge-based model construction and then employs a graph kernel on the resulting representation. kFOIL is a variation on the rule learner FOIL [Quinlan, 1990] that can learn kernels defined as the number of clauses that succeed in both interpretations. Gärtner et al. [2004] developed kernels within a typed higher order language and used it on some inductive logic programming benchmarks.

Also for what concerns neural networks, there is a stream of research work that combines neural with logical and symbolic representations, which is often referred to as neural-symbolic learning and reasoning [Garcez et al., 2015, 2008].

This research on probabilistic models, kernel-based methods and neural networks shows that it is important for logical and relational learning to integrate its principles and techniques with those of other schools in machine learning. Furthermore, the power of logical and relational learning is not only concerned with the expressiveness of the logical and relational representations but also with their declarativeness. Indeed, it has been repeatedly argued that logical and relational learning allows one to declaratively specify and solve problems by specifying background knowledge and declarative bias [De Raedt, 2008, Muggleton et al., 2012]. This property of logical and relational learning has turned out to be essential for many successes in applications as making small changes to the background knowledge or bias allows one to easily control the learning algorithm. While in the above mentioned probabilistic, kernel-based and neural approaches to logical and relational learning, it is typically possible to tune the logical and relational part in a declarative way, the probabilistic, kernel or neural components are typically built-in and hardcoded into the underlying formalisms and are very hard to modify. For instance, kLog was designed to allow different graph kernels to be plugged in, but support to declaratively specify the kernel is missing. Standard probabilistic programming languages such as PRISM and ProbLog have clear and fixed semantics (the distribution semantics) that cannot be changed. These limitations have motivated the development of algebraic logical languages such as Dyna [Eisner et al., 2004, Eisner and Filardo, 2011] and aProbLog [Kimmig et al., 2011]. While standard probabilistic programming languages such as PRISM and ProbLog label facts with probabilities, Dyna and aProbLog use algebraic labels belonging to a semiring, which allows the use of other algebraic structures than the probabilistic semi-ring on top of the underlying logic programs. Dyna has been used to encode many AI problems, particularly in the area of natural language processing.

But so far, the expressiveness of these languages is still limited, which explains why many contemporary machine learning techniques involving probabilistic models, kernels and support-vector machines or neural networks cannot yet be modeled in

these languages. Although Dyna and aProbLog have already been used to represent probabilistic models[1]), and the Dyna papers mention some simple neural networks, there is – to the best of our knowledge – not yet work on using such languages for kernel-based learning. It is precisely this gap that we want to fill in this work.

The key contribution of this work is the introduction of kProbLog an algebraic extension of Prolog, which can be used to declaratively model a wide range of problems and components from contemporary machine learning. More specifically, we shall show that kProbLog enables the declarative specification of four types of models that are central to machine learning today:

1. *tensor-based operations*: kProbLog allows to encode tensor operations in a way that is reminiscent of tensor relational algebra [Kim and Candan, 2011]. kProbLog supports recursion and is therefore more expressive than tensor relational algebra and related representations that have been proposed for relational learning [Nickel et al., 2011].

2. *a wide family of kernel functions*: Declarative programming of kernels on structured data can be achieved via algebraic labels in the semiring of polynomials. Polynomials were previously used in combination with logic programming for sensitivity analysis by Kimmig et al. [2011] and for data provenance by Green et al. [2007]. In this chapter, we show that polynomials as kProbLog's algebraic labels enable the specification of label propagation and feature extraction schemas as those used in recent graph kernels such as Weisfeiler-Lehman graph kernels [Shervashidze et al., 2011], propagation kernels [Neumann et al., 2012b] and graph kernels with continuous attributes such as graph invariant kernels [Orsini et al., 2015a]. Other graph kernels such as those based on random walks [Kashima et al., 2003, Mahé et al., 2004] can be also easily declared in our language.

3. *probabilistic programs*: kProbLog is, as we show in § 5.5, a generalization of the ProbLog probabilistic programming language.

4. *automatic differentiation*: kProbLog supports automatic differentiation by means of dual numbers [Eisner, 2002]. Many learning strategies (ranging from collaborative filtering to neural networks and deep learning) that combine tensor-based operations with gradient descent parameter tuning can therefore be implemented within the language.

The ability to define tensors, kernels, probabilistic models and support automatic differentiation are essential to contemporary machine learning. By supporting

[1] Dyna does not handle the disjoint-sum problem; a more detailed explanation about reasoning about possible worlds and the disjoint-sum can be found in §5.5.

declarative modeling of such techniques in a relational setting, kProbLog contributes towards briding the gap between logical and relational learning and contemporary machine learning. We also provide an implementation of kProbLog and show using a number of experiments that it can be applied in practice, especially for prototyping.

At the more technical level, the key novelty of kProbLog as compared to Dyna and aProbLog is the introduction of two simple yet powerful mechanisms: the coexistence of multiple semirings within the same program, and the use of meta-functions for combining and manipulating algebraic values beyond simple "sum" and "product" operations. This allows to use kProbLog for declaratively specifying not only the logical component but also the algebraic one. The underlying idea being that the logic captures the structural aspect of the problem while the atom labels capture the algebraic aspect (including counts of substructures). We shall formally define the underlying semantics, provide an implementation of the language and prove its convergence properties.

The chapter is an extended version of

- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. kProbLog: An Algebraic Prolog for Kernel Programming. In: International Conference on Inductive Logic Programming. Springer International Publishing, 2015. p. 152-165. **Best Student Paper, Machine Learning Journal Award.**

and is organized as follows.

First, we provide the necessary algebraic background §5.1. We then introduce kProbLog in §5.2 in three steps: §5.2.1 describes a simplified version of the language based on a single semiring, the full kProbLog described in §5.2.2 allows for multiple semirings and meta-functions, while §5.2.3 introduces the inference algorithm, its implementation and provides a convergence analysis. Section §5.2 also illustrates the relationship with tensor algebra. In §5.3 we then explain how kProbLog can be used to declaratively specify some complex state-of-the-art graph kernels, § 5.4 shows that it is possible to perform automatic differentiation in kProbLog, while § 5.5 shows that kProbLog is a proper generalization of ProbLog and hence, can be used as a probabilistic programming language. The work on kProbLog is then evaluated in § 5.6: we show that kProbLog is expressive enough to allow for encoding kernels for some real world application domains and that the implementation is usable in that we obtain good statistical performance and runtimes on some benchmarks. Finally, in § 5.7, we offer a comparative analysis of kProbLog and related languages, and draw some conclusions in §5.8.

5.1 Algebraic background

We now review some mathematical definitions.

Definition 9. *A* monoid *is an algebraic structure* $(\mathbb{S}, \cdot, e)$*, where* $\mathbb{S}$ *is a set and* $\cdot : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ *is a binary operation,* $e \in \mathbb{S}$ *is the neutral element and the following properties are satisfied:*

1. associativity $\forall a, b, c \in \mathbb{S}\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. neutral element $\exists e : \forall s \in \mathbb{S} :\ e \cdot a = a \cdot e = a$.

A monoid is called commutative *if* $\forall a, b \in \mathbb{S} : a \cdot b = b \cdot a$.

Definition 10. *A* semiring *is an algebraic structure* $S = (\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ *which satisfies the following properties:*

1. $(\mathbb{S}, \oplus, 0_S)$ *is a commutative monoid,*
2. $(\mathbb{S}, \otimes, 1_S)$ *is a monoid,*
3. distributive *multiplication left and right distributes over addition i.e.* $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.
4. annihilating element *the neutral element of the sum* 0_s *is the annihilating element of multiplication:* $0_s \otimes a = a \otimes 0_s = 0_s$.

A semiring is commutative *if* $\forall a, b \in \mathbb{S}\ a \otimes b = b \otimes a$ *(i.e.* $(\mathbb{S}, \otimes, 1_S)$ *is a commutative monoid).*

Definition 11. *A semiring* $S = (\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ *is* complete *if it is possible to define sums for all families* $(a_i | i \in I)$ *of elements of* $\mathbb{S}$ *where* I *is an arbitrary index set, such that the following conditions are satisfied [Droste and Kuich, 2009]:*

1. $\bigoplus_{i \in \emptyset} a_i = 0_S$, $\bigoplus_{i \in \{j\}} a_i = a_j$, $\bigoplus_{i \in \{j,k\}} a_i = a_j \oplus a_k$ *for* $j \neq k$.
2. $\bigoplus_{j \in J} (\bigoplus_{i \in I_j} a_i) = \bigoplus_{i \in I} a_i$ *for* $\bigcup_{j \in J} I_j = I$ *and* $I_j \cap I_k = \emptyset, j \neq k$.
3. $\bigoplus_{i \in I} (c \otimes a_i) = c \otimes (\bigoplus_{i \in I} a_i)$, $\bigoplus_{i \in I} (a_i \otimes c) = (\bigoplus_{i \in I} a_i) \otimes c$.

These properties of a complete semiring S define *infinite sums* 1. that extend finite sums, 2. are associative and commutative 3. satisfy the distributive law [Droste and Kuich, 2009].

Definition 12. *A semiring* $(\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ *is* naturally ordered *if the set* $\mathbb{S}$ *is partially ordered by the relation* $\sqsubseteq$ *such that* $\forall a, b \in \mathbb{S} :\ a \sqsubseteq b$ *if* $\exists c \in \mathbb{S} : a \oplus c = b$*. The partial order relation* $\sqsubseteq$ *on* A *is called* natural order *[Kuich, 1997].*

Definition 13. *A semiring* $(\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ *is* ω-continuous *when: a) is complete b) is naturally ordered c) if* $\bigoplus_{i=1}^{n} a_i \sqsubseteq c\ \forall n \in \mathbb{N}$ *then* $\bigoplus_{i \in \mathbb{N}} a_i \sqsubseteq c$ *for all sequences* $\{a_n\}_{i \in \mathbb{N}}$ *in* $\mathbb{S}$ *and* $c \in \mathbb{S}$.

5.2 The kProbLog language

We introduce kProbLog in three different steps. In the first subsection, we assume that a single semiring is used; in the second subsection we introduce meta-functions and allow for for multiple semirings; in the third, we present the inference algorithm of kProbLog, and analyse its convergence in the fourth subsection.

5.2.1 kProbLog$^{\mathbb{S}}$

kProbLog$^{\mathbb{S}}$ is an algebraic extension of Prolog with *labeled* facts and rules, where labels are chosen from a semiring S.

Definition 14. *A kProbLog$^{\mathbb{S}}$ program P is a 4-tuple (F, R, S, ℓ) where:*

- *F is a finite set of facts;*
- *R is a finite set of definite clauses (also called rules);*
- *S is a semiring with sum $\oplus$ and product $\otimes$ operations; whose neutral elements are 0_S and 1_S respectively;*
- *$\ell : F \to \mathbb{S}$ is a function that maps facts to semiring values.*

Definition 15. *An algebraic interpretation $I_w = (I, w)$ of a ground kProbLog$^{\mathbb{S}}$ program P with facts F and atoms A is a set of tuples $(a, w(a))$ where a is an atom in the Herbrand base A and $w(a)$ is an algebraic formula over the fact labels $\{\ell(f) | f \in F\}$. We use the symbol $\emptyset$ to denote the empty algebraic interpretation, i.e. $\{(\mathit{true}, 1_S)\} \cup \{(a, 0_S) | a \in A\}$.*

In this definition and below we adapt the notation of Vlasselaer et al. [2015].

Definition 16. *Let P be a ground algebraic logic program with algebraic facts F and Herbrand base A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the $T_{(P,S)}$-operator is $T_{(P,S)}(I_w) = \{(a, w'(a)) | a \in A\}$ where:*

$$w'(a) = \begin{cases} \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\} \subseteq I \\ a :- b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) & \text{if } a \in A \setminus F \end{cases} . \tag{5.1}$$

Example 6. *use of the algebraic T_P-operator.*

kProbLog$^{\mathbb{S}}$	*algebraic* T_P*-operator*	*numerical example*
`a :- a, b.` `a :- c.`	$w(a) \otimes w(b)$ $\oplus$ $w(c)$	0.5×0.3 $+$ 0.9

$w(a) = 0.5$

$w(b) = 0.3$

$w(c) = 0.9$

The least fixed point can be computed using a semi-naive evaluation. When the semiring is non-commutative the product $\otimes$ of the weights $w(b_i)$ must be computed in the same order that they appear in the rule. kProbLog$^{\mathbb{S}}$ can represent matrices that in principle can have infinite size and can be indexed by using elements of the Herbrand universe of the program. We now show some elementary kProbLog$^{\mathbb{S}}$ programs that specify matrix operations:

	algebra	kProbLog$^{\mathbb{S}}$	numerical example

matrix A	A	`1::a(0, 0).`
		`2::a(0, 1).`
		`3::a(1, 1).`
			$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right]$

matrix B	B	`2::b(0, 0).`
		`1::b(0, 1).`
		`5::b(1, 0).`
		`1::b(1, 1).`
			$\left[\begin{smallmatrix}2 & 1\\ 5 & 1\end{smallmatrix}\right]$

matrix transpose	A^t	`c(I, J) :- a(J, I).`	$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right]^t = \left[\begin{smallmatrix}1 & 0\\ 2 & 3\end{smallmatrix}\right]$
matrix sum	$A + B$	`c(I, J) :- a(I, J).` `c(I, J) :- b(I, J).`	$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right] + \left[\begin{smallmatrix}2 & 1\\ 5 & 1\end{smallmatrix}\right] = \left[\begin{smallmatrix}3 & 3\\ 5 & 4\end{smallmatrix}\right]$
matrix product	AB	`c(I, J) :-` `  a(I, K), b(K, J).`	$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right]\left[\begin{smallmatrix}2 & 1\\ 5 & 1\end{smallmatrix}\right] = \left[\begin{smallmatrix}12 & 3\\ 15 & 3\end{smallmatrix}\right]$
Hadamard product	$A \odot B$	`c(I, J) :-` `  a(I, J), b(I, J).`	$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right] \odot \left[\begin{smallmatrix}2 & 1\\ 5 & 1\end{smallmatrix}\right] = \left[\begin{smallmatrix}2 & 2\\ 0 & 3\end{smallmatrix}\right]$
Kronecker product	kron(A, B)	`c(i(Ia, Ib), j(Ja, Jb)):-` `  a(Ia, Ja), b(Ib, Jb).`	$\left[\begin{smallmatrix}1 & 2\\ 0 & 3\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}2 & 1\\ 5 & 1\end{smallmatrix}\right] = \left[\begin{smallmatrix}2&1&4&2\\ 5&1&10&2\\ 0&0&6&3\\ 0&0&15&3\end{smallmatrix}\right]$

The compound terms [2] `i/2` and `j/2`, were used to create the new indices that are needed by the Kronecker product. These definitions of matrix operations are reminiscent of tensor relational algebra [Kim and Candan, 2011]. Each of the above programs can be evaluated by applying the $T_{(P,S)}(I_w)$ operator only once. For each program we have a different definition of the C matrix that is represented by the predicate `c/2`. As a consequence of Equation 5.1 all the algebraic labels of the `c/2`

[2] We use the notation $functor/arity$ for compound terms.

facts are polynomials in the algebraic labels of the `a/2` and `b/2` facts. We draw an analogy between the representation of a sparse tensor in coordinate format and the representation of an algebraic interpretation. A ground fact can be regarded as a tuple of indices/domain elements that uniquely identifies the cell of a tensor, the algebraic label of the fact represents the value stored in the cell.

Definition 17. *An algebraic interpretation $I_w = (I, w)$ is the fixed point of the $T_{(P,S)}(I_w)$-operator if and only if for all $a \in A$, $w(a) \equiv w'(a)$, where $w(a)$ and $w'(a)$ are algebraic formulae for a in I_w and $T_{(P,S)}(I_w)$ respectively.*

We denote with $T^i_{(P,S)}$ the function composition of $T_{(P,S)}$ with itself i times.

Corollary 1 (application of Kleene's theorem)**.** *If S is an ω-continuous semiring the algebraic system of fixed-point equations $I_w = T_{(P,S)}(I_w)$ admits a unique least solution $T^\infty_{(P,S)}(\emptyset)$ with respect to the partial order $\sqsubseteq$ and $T^\infty_{(P,S)}(\emptyset)$ is the supremum of the sequence $T^1_{(P,S)}(\emptyset), T^2_{(P,S)}(\emptyset), \ldots, T^i_{(P,S)}(\emptyset)$. So $T^\infty_{(P,S)}(\emptyset)$ can be approximated by computing successive elements of the sequence. If the semiring satisfies the* ascending chain property *(see [Esparza et al., 2014]) then $T^\infty_{(P,S)}(\emptyset) = T^i_{(P,S)}(\emptyset)$ for some $i \geq 0$ and $T^\infty_{(P,S)}(\emptyset)$ can be computed exactly [Esparza et al., 2014].*

Examples of ω-continuous semirings are the Boolean semiring $(\{\mathrm{T}, \mathrm{F}\}, \vee, \wedge, \mathrm{F}, \mathrm{T})$, the tropical semiring $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ and the fuzzy semiring $([0, 1], \max, \min, 0, 1)$ [Green et al., 2007].

Example 7. *Let us consider the following kProbLog$^{\mathbb{S}}$ program:*

```
1::edge(a, b).
3::edge(b, c).
7::edge(a, c).

path(X, Y):-
  edge(X, Y).
path(X, Y):-
  edge(X, Z), path(Z, Y).
```

Assuming that S is the Boolean semiring and that all the algebraic labels that are different from 0_S correspond to `true`$\in \mathbb{S}$; we obtain the Warshall algorithm for the transitive closure of a binary relation.

If S is the tropical semiring, we obtain a specification of the Floyd-Warshall algorithm for all-pair shortest paths on graphs.

Lehmann [1977] explains how the Floyd-Warshall algorithm can be employed to invert square matrices. The inverse A^{-1} of a square matrix A can be computed as the result of the transitive closure of $I - A$ where I is the identity matrix. The last example requires the capability to compute additive inverses which are not guaranteed to exist for semirings. In §5.2.2 we will introduce meta-functions and show how they overcome this problem.

5.2.2 kProbLog

kProbLog generalizes kProbLog$^{\mathbb{S}}$ in two ways: it allows *multiple semirings* to coexist in the same program, and it enriches the algebraic expressivity by means of *meta-functions* and *meta-clauses*.

Every algebraic predicate in a kProbLog program needs to be associated with its own semiring S via the built-in predicate `declare(P, S)` where `P` is either a predicate (written in the form *name/arity*) or a list of predicates and `S` specifies a member of the kProbLog semiring library[3]. For example, the directive

```
:- declare(vertex/2, polynomial(real)).
```

is used to associate `vertex/2` with the semiring of polynomials over real numbers.

Definition 18 (meta-function)**.** *A meta-function* `m`: $\mathbb{S}_1 \times \ldots \times \mathbb{S}_k \mapsto \mathbb{S}'$ *is a function that maps k semiring values $x_i \in \mathbb{S}_i,\ i = 1, \ldots, k$ to a value of type $\mathbb{S}'$, where $\mathbb{S}_1, \ldots, \mathbb{S}_k$ and $\mathbb{S}'$ can be distinct sets. If* `a_1,...,a_k` *are algebraic atoms, in kProbLog we use the syntax* `@m[a_1,...,a_k]` *to express the application of meta-function* `@m` *to the values w(`a_1`), ..., w(`a_k`) of the atoms* `a_1,...,a_k`*.*

Definition 19 (meta-clause)**.** *A meta-clause* `h :- b_1,...,b_n` *is a universally quantified expression where* `h` *is an atom and* `b_1,...,b_n` *can be either atoms or meta-functions applied to other algebraic atoms. The head predicate of a meta-clause, the algebraic atoms in the body, and the return types of the meta-functions in the body must all belong to the same semiring.*

The introduction of meta-functions in kProbLog allows us to deal with other algebraic structures such as rings that require the additive inverse `@minus/1` and fields that require the additive inverse and the multiplicative inverse `@inv/1`.

Definition 20 (kProbLog program)**.** *A kProbLog program P is a union of kProbLog$^{\mathbb{S}_i}$ programs and meta-clauses.*

5.2.2.1 kProbLog T_P-operator with meta-functions

The algebraic T_P-operator of kProbLog is defined on the meta-transformed program.

Definition 21 (meta-transformed program)**.** *A meta-transformed kProbLog program is a kProbLog program in which all the meta-functions are expanded to algebraic atoms. For each rule* `h :- b_1,...,@m[a_1,...,a_k],...,b_n` *in the program P each meta-function* `@m[a_1,...,a_k]` *is replaced by an atom* `b'` *and a meta-clause* `b' :-@m[a_1,...,a_k]` *is added to the program P.*

[3]The library contains can be extended with the Python language.

Definition 22 (algebraic T_P-operator with meta-functions)**.** *Let P be a meta-transformed kProbLog program with facts F and atoms A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the T_P-operator is $T_P(I_w) = \{(a, w'(a))|a \in A\}$ where:*

$$w'(a) = \begin{cases} & \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\}\subseteq I \\ a:-b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) \oplus & \bigoplus\limits_{\substack{\{b_1,\ldots,b_k\}\subseteq I \\ a:-@m[b_1,\ldots,b_k]}} m(w(b_1),\ldots,w(b_k)) & \text{if } a \in A \setminus F\,. \end{cases} \quad (5.2)$$

Example 8. *of the algebraic T_P-operator with meta-functions.*

kProbLog	*algebraic T_P-operator*	*numerical example*	
`a :- a, b.` `a :- @sin[c].`	$w(a) \otimes w(b)$ $\oplus$ $sin(w(c))$	0.5×0.3 $+$ $0.78...$	$w(a) = 0.5$ $w(b) = 0.3$ $w(c) = 0.9$

Where we used the identity $\sin(0.9) = 0.78\ldots$

5.2.2.2 Recursive kProbLog program with meta-functions

Recursion is a basic tool in logic programming. For our purposes, it is necessary in most useful computations on structured data such as shortest paths (see Example 7), or random walk graph kernels (See §5.3.4.3). Weights need to be updated whenever the groundings of a predicate appear in the cycles of the ground program. kProbLog allows both additive and destructive updates, as specified by the built-in predicate `declare(P, S, U)` where `U` can be either `additive` or `destructive`.

Definition 23. *Additive and destructive updates.*

If `r_1, ..., r_n` *are all ground cyclic rules with head* `h`*, the value of the weight update value $\Delta w(\mathtt{h})$ is computed as:*

$$\Delta w(\mathtt{h}) = \bigoplus_{i=1}^{n} T_P(\mathtt{r\_i}). \quad (5.3)$$

According to the declaration of the predicate of atom `h` *the update will be either*

- `additive` $w(\mathtt{h}) = w(\mathtt{h}) \oplus \Delta w(\mathtt{h})$ *or*

- `destructive` $w(\mathtt{h}) = \Delta w(\mathtt{h})$.

The distinction between `additive` and `destructive` is only relevant for cyclic rules. In § 5.2.3 we give the evaluation algorithm of kProbLog which uses this kind of update when necessary.

Programs such as the transitive closure of a binary relation (see Example 7) or the compilation of ProbLog programs with SDDs require additive updates (see §5.5). Destructive updates are necessary to specify iterated function composition, as shown in the next example.

Example 9. *Suppose we wish to compute*

$$\lim_{n \to +\infty} g^n(x_0)$$

where $g(x) = x(1-x)$ *and*

$$g^n(x) = \begin{cases} (g \circ g^{n-1})(x) & \text{if } n > 0, \\ x & \text{if } n = 0. \end{cases} \tag{5.4}$$

The value $g^n(x_0)$ *can be obtained in kProbLog as follows:*

```
:- declare(x, real, destructive).
:- declare(x0, real).
0.5::x0.
x :- x0.
x :- @g[x].
```

The above program has the following behavior: the weight $w(x)$ *of* `x` *is initialized to* $w(x_0) = 0.5$ *and then updated at each step according to the rule* $w'(x) = g(w(x))$ *(destructive update). An additive update* $w'(x) = w(x) + g(w(x))$ *would have produced an incorrect result in this case.*

5.2.2.3 The Jacobi method

We already showed that kProbLog can express linear algebra operations. We now combine recursion and meta-functions in an algebraic program that specifies the Jacobi method [Golub and Van Loan, 2012], an iterative algorithm used for solving diagonally dominant systems of linear equations $A\mathbf{x} = \mathbf{b}$.

We consider the field of real numbers $\mathbb{R}$ (i.e. kProbLog$^{\mathbb{R}}$) as semiring together with the meta-functions `@minus` and `@inv` that provide the inverse element of sum and product respectively.

The A matrix must be split according to the Jacobi method:

$D = \text{diag}(A)$
$R = A - D$

```
d(I, I) :- a(I, I).
r(I, J) :- a(I, J), I \= J.
```

The solution $\mathbf{x}^*$ of $A\mathbf{x} = \mathbf{b}$ is computed iteratively by finding the fixed point of $\mathbf{x} = D^{-1}(\mathbf{b} - R\mathbf{x})$. We call E the inverse of D. Since D is diagonal also E is a diagonal matrix:

$e_{ii} = \texttt{invert}(d_{ii}) = \frac{1}{d_{ii}}$

```
e(I, I) :- @invert[d(I, I)].
```

and the iterative step can be rewritten as $\mathbf{x} = E(\mathbf{b} - R\mathbf{x})$.

Making the summations explicit we can write:

$$x_i = \sum_k e_{ik} \left(b_k - \sum_l r_{kl} x_l \right) \tag{5.5}$$

then we can extrapolate the term $\sum_l r_{kl} x_l$ turning it into the aux_k definition:

$x_i = \sum_k e_{ik} \left(b_k - aux_k \right)$

$aux_k = \sum_l r_{kl} x_l$

```
:- declare(x/1, real, destructive).
:- declare(aux/1, real, destructive).
x(I) :-
  e(I, K), @subtraction[b(K), aux(K)].

aux(K) :-
  r(K, L),  x(L).
```

where `@subtraction/2` represents the subtraction between real numbers, `x/1` and `aux/1` are mutually recursive predicates. Because `x/1` needs to be initialized (perhaps at random) we also need the clause:

$x_i = init_i$

```
x(I) :- init(I).
```

where `init/1` is a unary predicate. This example also shows that kProbLog is more expressive than tensor relational algebra because it supports recursion.

The introduction of meta-functions makes the result of the evaluation of a kProbLog program dependent on the order in which rules and meta-clauses are evaluated. For this reason we explain the order adopted by the kProbLog language.

5.2.3 kProbLog implementation

Pseudo-code for the interpreter is given in Algorithm 2. A kProbLog program P is first grounded to a kProbLog program by the procedure GROUND and is then evaluated by

partitioning GROUND(P) into a topologically ordered sequence of strata $P_1, \ldots, P_n$ such that

- every stratum P_i is a set of ground atoms which is both is maximal and strongly connected (i.e. each ground atom in P_i depends on every other ground atom in P_i);
- a ground atom in an acyclic stratum P_i can only depend[4] on the ground atoms from the previous strata $\bigcup_{j<i} P_j$;
- an ground atom in a cyclic stratum can depend on the ground atoms in $\bigcup_{j\leq i} P_j$.

Program evaluation starts by initializing the weight $w(a)$ of each ground atom $a \in$ GROUND(P) to 0_S, where S is the semiring of the atom. The strata are then visited in topological order and the weights are updated as follows: if the stratum P_i is acyclic, then the algebraic T_P-operator is applied only once per atom; if P_i is cyclic then the algebraic T_P-operator is first applied to the acyclic rules and meta-clauses and then, repeatedly until convergence, to the cyclic rules and meta-clauses. The procedure TPOPERATOR takes as input a $rule$ and the atom weights w and returns an algebraic value derived from the application of the algebraic T_P-operator.

The update for a weight $w(a)$ of a cyclic atom a is computed by accumulating the result of the application of the T_P-operator to all the cyclic rules with head a. The new weight is then computed as $w(a) = w(a) + \Delta w(a)$ (additive updates) or $w(a) = \Delta w(a)$ (destructive updates).

If P_i is a cyclic stratum, then it is the responsibility of the programmer to ensure convergence of the algebraic T_P-operator. Nevertheless if the P_i is a cyclic stratum in which only rules are cyclic then all the atoms in P_i are on the same semiring[5] S and so P_i has the same convergence properties of a kProbLog$^{\mathbb{S}}$ program (see Corollary 2 on page 72). Whenever we apply the algebraic T_P-operator we use the Jacobi evaluation. Jacobi and Gauss-Seidel evaluations are two alternatives to perform naive evaluation of Datalog programs and are also well known in numerical analysis [Ceri et al., 1989]. We choose Jacobi over Gauss-Seidel evaluation because the former produces side effects on the algebraic weighs w only after (and not during) the computation of the algebraic T_P-operator. In this way the execution of the program is not affected by the order in which rules and meta-clauses are evaluated.

This program evaluation procedure is an adaptation the work of Whaley et al. [2005] on Datalog and binary decision diagrams. kProbLog was implemented in Python 3.5

[4] We say that an atom *a directly depends* on an atom *b* if *a* is the head of a rule or a meta-clause and *b* is a body literal or an argument of a meta-function in the meta clause. We say that an atom *a depends* on an atom *b* either if *a* directly depends on *b* or there is an atom *c* such that *a* directly depends on *c* and *c* depends on *b*.

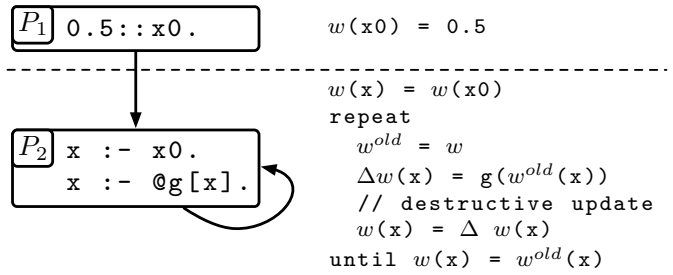
[5] Atoms of distinct semirings cannot be mutually dependent without using meta-clauses.

Algorithm 2 Pseudo-code for the kProbLog evaluation procedure.

kPROBLOG(P)

$F = \text{FACTS}(P)$
for $f \in F$
    $w(f) = \ell(f)$
$[P_1, \ldots, P_n] = \text{TOPSORT}(\text{GETSTRATA}(\text{GROUND}(P)))$
for $i = 1, \ldots, n$
    for $a \in P_i \setminus F$
        $w(a) = 0_s$
    $w_{old} = w$
    $ACYCLIC = \text{ACYCLICRULES}(P_i)$
    $CYCLIC = \text{CYCLICRULES}(P_i)$
    for $rule \in ACYCLIC$
        $h = \text{HEAD}(rule)$
        $w(h) = w_{old}(h) \oplus \text{TPOPERATOR}(rule, w_{old})$
    repeat
        $w_{old} = w$
        for $rule \in CYCLIC$
            $\Delta w(\text{HEAD}(rule)) = 0_s$
        for $rule \in CYCLIC$
            $h = \text{HEAD}(rule)$
            $\Delta w(h) = \Delta w(h) \oplus \text{TPOPERATOR}(rule, w_{old})$
        for $rule \in CYCLIC$
            if rule is additive
                $w(\text{HEAD}(rule)) = w_{old}(\text{HEAD}(rule)) \oplus \Delta w(\text{HEAD}(rule))$
            else // rule is destructive
                $w(\text{HEAD}(rule)) = \Delta w(\text{HEAD}(rule))$
    until $w_{old} = w$

Figure 5.1: Cyclic program of Example 9: dependency graph with stratification and corresponding weight updates.



using Gringo 4.5 [6] as grounder. The source code of our kProbLog implementation is available at `https://github.com/orsinif/kProbLogDSL`.

Example 9 (continued)**.** *Evaluation of a cyclic program. The cyclic program P in §5.2.2.2 is already ground and contains two ground atoms* `x0` *and* `x`*. The ground atoms* `x0` *and* `x` *correspond to two nodes in the dependency graph, while* `x0` *is a fact and does not have incoming arcs,* `x` *has two dependencies/incoming arcs which are* `x0` *and itself. As shown in Figure 5.1 P is then subdivided in two strata P_1 and P_2: P_1 contains* `x0` *and is acyclic, P_2 contains* `x` *and is cyclic.*

The algebraic T_P-operator is applied only once for acyclic rules and multiple times, until convergence, for cyclic rules (i.e. `x:- @g[x].`*)*

5.2.4 Convergence analysis of the kProbLog interpreter

In order to analyze the convergence of the kProbLog interpreter on a kProbLog program P, we assume that all the meta-functions in P terminate and that the *finite support* condition [Sato, 1995] holds.

The *finite support* condition is commonly used in probabilistic logic programming and ensures that the GROUND procedure outputs a finite ground program.

The convergence properties of kProbLog are characterized by the following theorems.

Theorem 1 (Convergence of acyclic kProbLog programs)**.** *The evaluation of an acyclic kProbLog program P invokes the algebraic T_P-operator exactly once for each ground rule in* GROUND(P) *and terminates.*

[6] `https://potassco.org`

Theorem 2 (Convergence of kProbLog programs)**.** *The evaluation of a kProbLog program is guaranteed to terminate only if all the cyclic strata are kProbLog$^{\mathbb{S}_i}$ programs where $\mathbb{S}_i$ are ω-continuous semirings.*

The proofs of Theorems 1 and 2 are reported in Appendix § 5.9.1.

Theorem 1 can be used to prove the convergence of the elementary programs that specify matrix operations in § 5.2.1 and the convergence of the Weisfeiler-Lehman algorithm that we shall se in § 5.3.2. Theorem 2 ensures the convergence of the cyclic program in Example 7 when an ω-continuous semiring is used for the algebraic labels, but not the convergence of the program in Example 9. While the cyclic program in Example 9 actually converges, this property cannot be entailed from Theorem 2. Indeed the program in Example 9 has a cyclic stratum P_2 (see Figure 5.1) involving a meta-function (i.e. `@g`). Stratum P_2 is not a kProbLog$^{\mathbb{S}}$ program because kProbLog$^{\mathbb{S}}$ programs do not admit meta-functions and for this reason we cannot apply Theorem 2 to Example 9.

5.3 Kernel programming

We now show that kProbLog can be used to declaratively encode state-of-the-art graph kernels. But before doing so, we introduce the semiring $\mathbb{S}[\mathbf{x}]$ that can be used for feature extraction.

5.3.1 kProbLog$^{\mathbb{S}[\mathbf{x}]}$: polynomials for feature extraction

kProbLog$^{\mathbb{S}[\mathbf{x}]}$ labels facts and rule heads with polynomials over the semiring S. kProbLog$^{\mathbb{S}[\mathbf{x}]}$ is a particular case of kProbLog$^{\mathbb{S}}$ because polynomials over semirings are semirings in which addition and multiplication are defined as usual.

While polynomials have been used in combination with logic programming for provenance [Green et al., 2007] and sensitivity analysis [Kimmig et al., 2011], we use multivariate polynomials to represent the explicit feature map of a graph kernel.

Definition 24 (Multivariate polynomials over commutative semirings)**.** *A multivariate polynomial $\mathcal{P} \in S[\mathbf{x}]$ can be expressed as:*

$$\mathcal{P}(\mathbf{x}) = \bigoplus_{i=1}^{n} c_i \mathbf{x}^{\mathbf{e}_i} = \bigoplus_{i=1}^{n} c_i \otimes \bigotimes_{t \in T} x_t^{e_{it}} \tag{5.6}$$

where $c_i \in \mathbb{S}$ are the coefficients of the i^{th} monomial and $\mathbf{x}$, $\mathbf{e}$ are vectors of variables and exponents respectively. The vector $\mathbf{x}$ is indexed by ground terms $t \in T$.

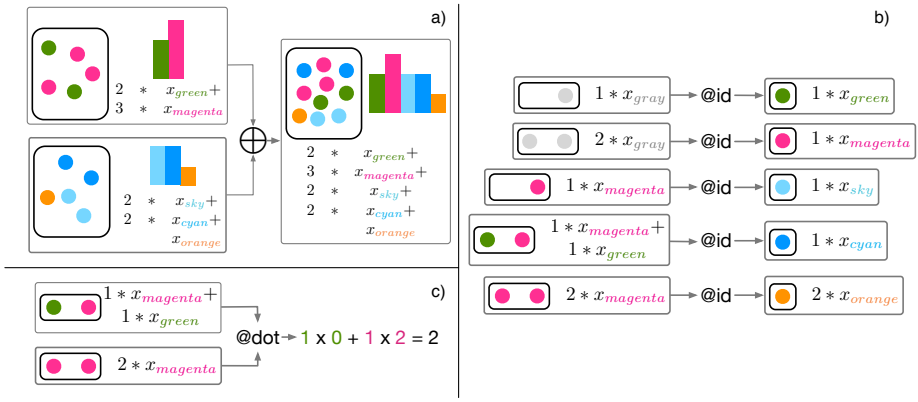


Figure 5.2: Using polynomials for representing multisets: a) multiset union corresponds to sum over polynomials; b) the inner product (kernel) between multisets corresponds to product over polynomials; c) multiset compression via the `@id` meta-function over polynomials.

Weisfeiler-Lehman, propagation and neighborhood-subgraph-pairwise-distance kernel features can all be casted into this representation. These graph kernels propagate messages through the structure of the graphs, these messages can be represented as multisets of terms (elements of the Herbrand universe). Indeed we can represent a multiset μ of terms as a polynomial:

$$\mathcal{P}_\mu(\mathbf{x}) = \sum_{t \in \mu} \sharp t \cdot x_t \tag{5.7}$$

where $\sharp$ counts the number of occurrences of the terms in the multiset μ.

5.3.1.1 Operations for feature extraction

Sum of polynomials The semiring sum $\oplus$ between polynomials is used in kProbLog$^{\mathbb{S}[\mathbf{x}]}$ to sum features or equivalently compute a multiset union operation (see Figure 5.2.a).

Inner product between polynomials The kProbLog `@dot` meta-function corresponds to the inner product on multivariate polynomials over $S[\mathbf{x}]$:

$$\langle \mathcal{P}(\mathbf{x}), \mathcal{Q}(\mathbf{x}) \rangle = \bigoplus_{\substack{(p,\mathbf{e}) \in \mathcal{P} \\ (q,\mathbf{e}) \in \mathcal{Q}}} p \otimes q. \tag{5.8}$$

For each monomial (uniquely identified by the vector of exponents $\mathbf{e}$) that appears in both the polynomials $\mathcal{P}$ and $\mathcal{Q}$, Equation 5.8 computes the product between their coefficients p and q respectively. These products are then summed together to obtain the value of the inner product. Natural choices for the semiring are polynomials over integers $\mathbb{Z}[\mathbf{x}]$ and real numbers $\mathbb{R}[\mathbf{x}]$, these semirings also ensure that the inner-product is positive semidefinite.

Example 10. *The following multivariate polynomials over integers:*

$$\begin{aligned}\mathcal{P}(x_1, x_2, x_3) &= 2x_1 + 3x_1x_2 + x_2x_3^2 \\ \mathcal{Q}(x_1, x_2, x_3) &= 4x_1 + 3x_1x_3 + 3x_2x_3^2\end{aligned} \tag{5.9}$$

can be expressed as two sets of coefficient-exponent pairs $\mathcal{P} = \{(2, [1, 0, 0]), (3, [1, 1, 0]), (1, [0, 1, 2])\}$ *and* $\mathcal{Q} = \{(4, [1, 0, 0]), (3, [1, 0, 1]), (3, [0, 1, 2])\}$ *respectively. The two polynomials have the vectors of exponents* $[1, 0, 0]$ *and* $[0, 1, 2]$ *in common, each contributes to the inner product by* $2 \times 4 = 8$ *and* $1 \times 3 = 3$ *respectively. The value of the inner product between* $\mathcal{P}(x_1, x_2, x_3)$ *and* $\mathcal{Q}(x_1, x_2, x_3)$ *is the sum of such contributions* $8 + 3 = 11$.

In kProbLog, the meta-function `@dot/2`. computes the inner product between two algebraic atoms $\mathcal{P}(\mathbf{x})$`::a` and $\mathcal{Q}(\mathbf{x})$`::b`. An example is shown in Figure 5.2.b where the multisets of terms {{`green`, `magenta`}} and {{`magenta`, `magenta`}} are represented by the following two polynomials:

$$\begin{aligned}\mathcal{P}(x_{green}, x_{magenta}) &= x_{green} + x_{magenta} \\ \mathcal{Q}(x_{green}, x_{magenta}) &= 2x_{magenta}\end{aligned} \tag{5.10}$$

Another useful meta-function in the context of kernel design is `@rbf/3`. It takes as input an atom labeled by a non-negative real value γ and two atoms labeled with the polynomials $\mathcal{P}$ and $\mathcal{Q}$ and it computes the radial basis function kernel $exp\{-\gamma\|\mathcal{P} - \mathcal{Q}\|^2\}$, where $\|\mathcal{P} - \mathcal{Q}\|^2 = \langle\mathcal{P}, \mathcal{P}\rangle + \langle\mathcal{Q}, \mathcal{Q}\rangle - 2\langle\mathcal{P}, \mathcal{Q}\rangle$.

The compression meta-function The `@id/1` meta-function `@id`: $\mathbb{S}[\mathbf{x}] \to \mathbb{S}[\mathbf{x}]$ is injective. `@id/1` transforms a polynomial $\mathcal{P}(\mathbf{x})$ to a new term t and returns the polynomial `@id`$[\mathcal{P}(\mathbf{x})] = 1.0 \cdot x(t)$. This function can be used to compress a multivariate polynomial to a new polynomial in a single variable. We use the `@id` meta-function for polynomial compression as Shervashidze et al. [2011] use the function f to compress multisets of labels. We now show how these functions are used to specify graph kernels.

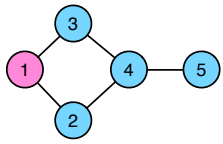
5.3.2 The Weisfeiler-Lehman algorithm

The one-dimensional Weisfeiler-Lehman method is an iterative vertex classification algorithm for the graph isomorphism problem. It begins by coloring vertices with their labels and, at each round, it recolors vertices by a "compressed" version of the multiset of colors at their neighbors. If, at any iteration, two graphs have different sets of vertex colors they cannot be isomorphic. We will use polynomials to represent Weisfeiler-Lehman colors, associating variables with colors and using integer coefficients to encode the number of occurrences of a color in a multiset.

A colored graph $G = (V, E, \ell)$, where V is a set of vertices, $E \subseteq V \times V$ is the set of the edges, and $\ell : V \mapsto \Sigma$ is a function that maps vertices to a color alphabet Σ, can be declared in kProbLog as follows:

```
:- declare(vertex/2, polynomial(int)).
:- declare(edge_asymm/3, boolean).
:- declare(edge/3, polynomial(int)).

1 * x(pink)::vertex(graph_a, 1).
1 * x(blue)::vertex(graph_a, 2).
1 * x(blue)::vertex(graph_a, 3).
1 * x(blue)::vertex(graph_a, 4).
1 * x(blue)::vertex(graph_a, 5).

edge_asymm(graph_a, 1, 2).
edge_asymm(graph_a, 1, 3).
edge_asymm(graph_a, 2, 4).
edge_asymm(graph_a, 3, 4).
edge_asymm(graph_a, 4, 5).
```

```
1.0::edge(Graph, A, B):-
    edge_asymm(Graph, A, B).

1.0::edge(Graph, A, B):-
    edge_asymm(Graph, B, A).
```



where predicate `edge_asymm/3` is implicitly cast to type *integer* and then to type *polynomial over integers* when it appears in the definition of `edge/3`. The Weisfeiler-Lehman color of vertex v at iteration h can be written as:

$$\mathcal{L}^h(v) = \begin{cases} \ell(v) & \text{if } h = 0 \\ f(\mathcal{L}^{h-1}(v), \{\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}\}) & \text{if } h > 0 \end{cases} \tag{5.11}$$

where $\mathcal{N}(v)$ is the set of the vertex neighbors of v, $\{\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}\}$ is the multiset of their colors at step $h-1$, and f is a variadic injective function that maps its arguments to a new color in Σ. [7] Equation 5.11 can be expressed in kProbLog as shown below, where f is implemented via the `@id` meta-function:

```
:- declare(wl_color/3,
    polynomial(int)).
:- declare(wl_color_multiset/3,
    polynomial(int)).

wl_color_multiset(H, Graph, V):-
    edge(Graph, V, W),
    wl_color(H, Graph, W).
```

```
wl_color(0, Graph, V) :-
    vertex(Graph, V).

wl_color(H, Graph, V):-
    1 <= H, H <= MAX_ITER,
    @id[wl_color(H-1, Graph, V),
    wl_color_multiset(H-1, Graph, V)].
```

[7] While mathematically equivalent, the formulation in Equation 5.11 is is slightly different from the one of

The Weisfeiler-Lehman algorithm has been specified as an acyclic program. Indeed, while `wl_color/3` and `wl_color_multiset/3` are mutually recursive `wl_color/3` at step `H` depends on `wl_color/3` and `wl_color_multiset/3` at step `H-1`, therefore is acyclic and we can apply Theorem 1 to verify that it converges.

5.3.3 Graph kernels

In this section we give the declarative specification of some recent graph kernels such as the Weisfeiler-Lehman graph kernel [Shervashidze et al., 2011], propagation kernels [Neumann et al., 2012b] and graph invariant kernels [Orsini et al., 2015a]. These methods have been applied to different domains such as natural language processing [Orsini et al., 2015a], computer vision [Neumann et al., 2012b] and bioinformatics [Shervashidze et al., 2011, Neumann et al., 2012b, Orsini et al., 2015a].

5.3.4 Weisfeiler-Lehman graph kernel and Propagation kernels

The Weisfeiler-Lehman graph kernel is defined using a base kernel [Shervashidze et al., 2011] that computes the inner-product between the histograms of Weisfeiler-Lehman colors of two graphs `Graph` and `GraphPrime`.

$$\phi^{(h)}(G) = \sum_{v \in V(G)} \mathcal{P}_{\text{WL}}^{(h)}(v)$$

```
:- declare(phi/2, real).
phi(H, Graph):-
  wl_color(H, Graph, V).
```

$$k_{\text{BASE}}^{(h)}(G, G') = \langle \phi^{(h)}(G), \phi^{(h)}(G') \rangle$$

```
:- declare(base_kernel/3, real).
base_kernel(H, Graph, GraphPrime):-
  @dot[phi(H, Graph),
       phi(H, GraphPrime)].
```

Where $\mathcal{P}_{\text{WL}}^{(h)}(v)$ is the polynomial that represents the Weisfeiler-Lehman color of vertex v at step h.

The Weisfeiler-Lehman graph kernel [Shervashidze et al., 2011] with `H` iterations is the sum of base kernels computed for consecutive Weisfeiler-Lehman labeling steps $1, \ldots,$`H` on the graphs `Graph` and `GraphPrime`:

Equation 3.4 in chapter 3. While the former uses multisets of vertex colors, the latter uses lexicographically sorted strings of vertex colors.

$$k_{\text{WL}}^{(H)}(G, G') = \sum_{h=0}^{H} k_{\text{BASE}}^{(h)}(G, G')$$

```
:- declare(kernel_wl/3, real).
kernel_wl(0, Graph, GraphPrime):-
  base_kernel(0, Graph, GraphPrime).

kernel_wl(H, Graph, GraphPrime):-
  H > 0, H1 is H - 1,
  kernel_wl(H1, Graph, GraphPrime).

kernel_wl(H, Graph, GraphPrime):-
  H > 0,
  base_kernel(H, Graph, GraphPrime).
```

The above equation can be rewritten using a recursive definition which is closer to the kProbLog specification as follows

$$k_{\text{WL}}^{(H)}(G, G') = \begin{cases} k_{\text{BASE}}^{(0)}(G, G') & \text{if } H = 0 \\ k_{\text{WL}}^{(H-1)}(G, G') + k_{\text{BASE}}^{(H)}(G, G') & \text{if } H > 0. \end{cases} \tag{5.12}$$

Propagation kernels [Neumann et al., 2012b] are a generalization of the Weisfeiler-Lehman graph kernel, that can adopt different label propagation schemas. Neumann et al. [2012b] implement propagation kernels using locality sensitive hashing (LSH). The kProbLog specification is almost identical to the one of the Weisfeiler-Lehman except that the `@id` meta-function is to be replaced with a meta-function that does LSH.

LSH discretizes vectors to integer identifiers so that vectors which are similar have the same integer identifier with high probability.

5.3.4.1 Shortest path Weisfeiler-Lehman graph kernel

The shortest path Weisfeiler-Lehman [Shervashidze et al., 2011] graph kernel is a variant of the Weisfeiler-Lehman graph kernel in which the base kernel counts the number of common occurrences of triplets of the form (a, b, d) between two graphs G and G'. The triplet (a, b, d) represents the occurrence of two vertices v and w at distance d with colors $a = \mathcal{L}(v)$ and $b = \mathcal{L}(w)$. To compactly encode the shortest path variant of the Weisfeiler-Lehman graph kernel we begin by computing all-pairs shortest paths using the tropical semiring:

```
:- declare(distance/3, tropical).
distance(Graph, V, W):-
  edge(Graph, V, W).

distance(Graph, V, W):-
  distance(Graph, V, U), edge(Graph, U, W).
```

We then introduce predicate `triplet_id(Graph, H,V, W)` of type `polynomial (real)` that associates each pair of vertices `V` and `W` of graph `Graph`

with their $\mathtt{H}^{\text{th}}$-iteration Weisfeiler-Lehman color, together with their shortest path distance, d, obtained by calling the `@id/1` meta-function on the distance predicate. Finally, triplet (a, b, d) is represented as the monomial $x_a x_b x_d$ via auxiliary predicate `triplet(Graph, H, V, W)` and compressed to a color by using again the `@id` meta-function:

```
:- declare(triplet/4, polynomial(real)).
:- declare(triplet_id/4, polynomial(real)).

triplet(Graph, H, V, W):-
  wl_color(H, Graph, V),
  wl_color(H, Graph, W),
  @id[distance(Graph, V, U)].

triplet_id(Graph, H, V, W):-
  @id[triplet_id(Graph, H, V, U)].
```

This specification fully employs the expressive power of kProbLog using meta-functions and two distinct semirings that encode distances and vertex colors (the base kernel for this variant of the Weisfeiler-Lehman graph kernel is obtained by replacing predicate `wl_color/3` defined in §5.3.4 with predicate `triplet_id/4` defined above).

5.3.4.2 Graph invariant kernels

Graph invariant kernels are the framework for graph kernels with continuous attributes [Orsini et al., 2015a] that we introduced in Chapter 3. GIKs compute a similarity measure between graphs G and G' matching them at vertex level according to the formula: [8]

$$k(G, G') = \sum_{v \in V(G)} \sum_{v' \in V(G')} w(v, v') k_{\text{ATTR}}(v, v') \tag{5.13}$$

where $w(v, v')$ is the structural weight matrix and $k_{\text{ATTR}}(v, v')$ is a kernel on the continuous attributes of the graphs. The kProbLog specification is parametrized by the logical variable `R`, which is needed for the definition of the structural weight matrix $w(v, v')$.

```
:- declare(gik_radius/3, real).
gik_radius(R, Graph, GraphPrime):-
  w_matrix(R, Graph, V, GraphPrime, VPrime),
  k_attr(Graph, V, GraphPrime, VPrime).
```

where `gik_radius/3`, `w_matrix/5` and `k_attr/4` are algebraic predicates on the real numbers semiring, which is represented with floats for implementation purposes. Assuming that we want to use the RBF with $\gamma = 0.5$ kernel on the vertex attributes we can write:

[8] For the sake of readability, we repeat Eq 3.1 of chapter 3.

```
:- declare(rbf_gamma_const/0, real).
:- declare(k_attr/4, real).
0.5::rbf_gamma_const.
k_attr(Graph, V, GraphPrime, VPrime):-
  @rbf[rbf_gamma_const, attr(Graph, V), attr(GraphPrime, VPrime)].
```

where `attr/2` is an algebraic predicate that associates to the vertex `V` of a `Graph` a polynomial label. To associate to vertex `v_1` of `graph_a` the 4-dimensional feature $[1, 0, 0.5, 1.3]$ we would write:

```
:- declare(attr/2, polynomial(real)).
1.0 * x(1) + 0.5 * x(3) + 1.3 * x(4)::attr(graph_a, v_1).
```

while the meta-function `@rbf/3` takes as input an atom `rbf_gamma_const` labeled with the γ constant and the atoms relative to the vertex attributes.

The structural weight matrix $w(v, v')$ is defined as:

$$w(v, v') = \sum_{g \in \mathcal{R}^{-1}(G)} \sum_{g' \in \mathcal{R}^{-1}(G')} k_{\text{INV}}(v, v') \frac{\delta_m(g, g')}{|V_g||V_{g'}|} \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}. \quad (5.14)$$

The weight $w(v, v')$ measures the structural similarity between vertices and is defined combining an $\mathcal{R}$-decomposition relation, a function $\delta_m(g, g')$ and a kernel on vertex invariants k_{INV} [Orsini et al., 2015a]. In our case the $\mathcal{R}$-decomposition generates `R`-neighborhood subgraphs (as those used in the experiments of Orsini et al. [2015a]).

There are multiple ways to instantiate GIKs, we choose the version called LWL_V, which can achieve very good accuracies most of the time as shown by Orsini et al. [2015a]. LWL_V uses `R`-neighborhood subgraphs $\mathcal{R}$-decomposition relation, computes the kernel on vertex invariants $k_{\text{INV}}(v, v')$ at the pattern level (*local* GIK) and uses $\delta_m(g, g')$ to match subgraphs that have the same number of nodes.

A `R`-neighborhood subgraph of a graph G from a vertex v is the subgraph induced by all the vertices in G whose shortest-path distance from v is less than or equal to `R`.

In kProbLog we would write:

```
:- declare(w_matrix/5, real).
w_matrix(R, Graph, V, GraphPrime, VPrime):-
  vertex_in_ball(Graph, R, BallRoot, V),
  vertex_in_ball(GraphPrime, R, BallRootPrime, VPrime),
  delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime),
  @inv[ball_size(R, Graph, BallRoot)],
  @inv[ball_size(R, GraphPrime, BallRootPrime)],
  k_inv(Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime).
```

where:
a) `vertex_in_ball(R, Graph, BallRoot, V)` is a Boolean predicate which is true if `V` is a vertex of `Graph` inside a `R`-neighborhood subgraph rooted in `BallRoot`. `vertex_in_ball/4` encodes both the term $\mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}$ and the pattern generation of the decomposition relation $g \in \mathcal{R}^{-1}(G)$.

```
:- declare(vertex_in_ball/4, bool).
vertex_in_ball(0, Graph, Root, Root):-
  vertex(Graph, Root).

vertex_in_ball(R, Graph, Root, V):-
  R > 0, R1 is R - 1,
  vertex_in_ball(R1, Graph, Root, V).
```

```
vertex_in_ball(R, Graph, Root, V):-
  R > 0, R1 is R - 1,
  edge(Graph, Root, W),
  vertex_in_ball(R1, Graph, W, V).
```

b) `delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime)` matches subgraphs with the same number of vertices

```
:- declare(delta_match/5, real).
:- declare(v_id/3, polynomial(real)).
:- declare(ball_size/3, int).

delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime):-
  @eq[v_id(R, Graph, BallRoot), v_id(R, GraphPrime, BallRootPrime)].

v_id(R, Graph, BallRoot):- @id[ball_size(R, Graph, BallRoot)].

ball_size(R, Graph, BallRoot):- vertex_in_ball(R, Graph, BallRoot, V).
```

c) `@inv[ball_size(Radius, Graph, BallRoot)]` corresponds to the normalization term $1/|V_g|$. `@inv` is the meta-function that computes the multiplicative inverse and `ball_size(Radius, Graph, BallRoot)` is a the float predicate that counts the number of vertices in a `Radius`-neighborhood rooted in `BallRoot`.
d) `k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime)` computes k_{INV} using `H_WL` iterations of the Weisfeiler-Lehman algorithm to obtain vertex features `phi_wl(R, H_WL, Graph, BallRoot, V)` from the `R`-neighborhood subgraphs.

```
:- declare(k_inv/7, real).
:- declare(phi_wl/5, polynomial(real)).
wl_iterations(3). % constant

k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime):-
  wl_iterations(H_WL),
  @dot[phi_wl(R, H_WL, Graph, BallRoot, V),
    phi_wl(R, H_WL, GraphPrime, BallRootPrime, VPrime)].

phi_wl(R, 0, Graph, BallRoot, V):-
  wl_color(R, Graph, BallRoot, 0, V).

phi_wl(R, H, Graph, BallRoot, V):-
  H > 0, wl_color(R, Graph, BallRoot, H, V).
```

```
phi_wl(R, H, Graph, BallRoot, V):-
  H > 0, H1 is H-1,
  phi_wl(R, H1, Graph, BallRoot, V).
```

where `wl_color/5` is defined as `wl_color/3`, but has two additional arguments `R` and `BallRoot` that are needed to restrict the graph connectivity to the `R`-neighborhood subgraph rooted in vertex `BallRoot`.

5.3.4.3 Random walk graph kernels

Vishwanathan et al. [2010] propose generalized random walk kernels. The similarity between a pair of graphs is computed by performing random walks on both graphs and then counting the number of matching walks.

Counting the number of matching random walks between two graphs $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$ is equivalent to counting the number of walks in $G_\times = (V_\times, E_\times)$, where $G_\times$ is the direct product between the graphs G_a and G_b [Vishwanathan et al., 2010].

The direct product graph $G_\times$ is defined in terms of G and G' as follows:

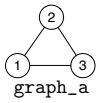
$$\begin{array}{lcl} V_\times & = & \{(v_a, v_b) | v_a \in V_a \wedge v_b \in V_b\} \\ E_\times & = & \{((v_a, u_a), (v_b, u_b)) | (v_a, u_a) \in E_a \wedge (v_b, u_b) \in E_b\} \end{array} \tag{5.15}$$

To encode the product graph in kProbLog, we start from an edge connectivity predicate `edge_asymm/3` such that `edge(Graph, V, U)` is true whenever there is an edge between vertices `V` and `U` in graph `Graph`. In a similar way we define the vertex predicate `vertex/2`.

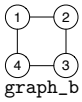
```
:- declare(vertex/2, bool).
:- declare(edge_asymm/3, bool).
:- declare(edge/3, real).
```

Predicate `edge/3` when its first argument is `graph_a` (`graph_b`) represents the adjacency matrix $W_a \in \mathbb{R}^{|V_a| \times |V_a|}$ ($W_b \in \mathbb{R}^{|V_b| \times |V_b|}$) of graph G_a (G_b).

We shall now consider the same example graphs used by Vishwanathan et al. [2010] starting from two graphs G_a and G_b encoded with the kProbLog symbols `graph_a` and `graph_b`.




```
vertex(graph_a, 1).     edge_asymm(graph_a, 1, 2).
vertex(graph_a, 2).     edge_asymm(graph_a, 1, 3).
vertex(graph_a, 3).     edge_asymm(graph_a, 2, 3).
```




```
vertex(graph_b, 1).     edge_asymm(graph_b, 1, 2).
vertex(graph_b, 2).     edge_asymm(graph_b, 2, 3).
vertex(graph_b, 3).     edge_asymm(graph_b, 3, 4).
vertex(graph_b, 4).     edge_asymm(graph_b, 1, 4).
```

we then define `edge/3` as the symmetric closure of `edge_asymm/3`.

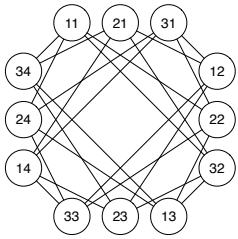
```
:- declare(edge/3, real).
edge(Graph, V, W):- edge_asymm(Graph, V, W).
edge(Graph, V, W):- edge_asymm(Graph, W, V).
```

The kernel definition also includes starting $\mathbf{p}_a \in \mathbb{R}^{|V_a|}$ ($\mathbf{p}_b \in \mathbb{R}^{|V_b|}$) and stopping $\mathbf{q}_a \in \mathbb{R}^{|V_b|}$ ($\mathbf{q}_b \in \mathbb{R}^{|V_b|}$) probabilities associated to the vertices of the graph G_a (G_b), that we shall assume to be uniform.

```
:- declare(prob_start/2, real).
:- declare(prob_stop/2, real).
:- declare(graph_size/1, real).

graph_size(G):- vertex(G, V).

prob_start(G, V):- % uniform initial probability
  vertex(G, V), @inv[graph_size(G)].

prob_stop(G, V):- % uniform stopping probability
  vertex(G, V), @inv[graph_size(G)].
```

The product graph $G_\times$ can be specified following Equation 5.15. When the first argument of predicate `edge/3` is `kron(graph_a, graph_b)` it represents the adjacency matrix $W_\times = W_a \times W_b \in \mathbb{R}^{|V_a||V_b| \times |V_a||V_b|}$ of $G_\times$.

According to Vishwanathan et al. [2010] also the initial (stopping) probabilities $\mathbf{p}_\times$ ($\mathbf{q}_\times$) of the vertices $V_\times$ can be obtained as the Kronecker product between the initial (stopping) probabilities of G_a and G_b, i.e. $\mathbf{p}_\times = \mathbf{p}_a \times \mathbf{p}_b$ ($\mathbf{q}_\times = \mathbf{q}_a \times \mathbf{q}_b$).



kron(graph_a, graph_b)

```
vertex(kron(Ga, Gb), i(Va, Vb)):-
  vertex(Ga, Va), vertex(Gb, Vb).

edge(kron(Ga, Gb), i(Va, Vb), i(Ua, Ub)):-
  edge(Ga, Va, Ua), edge(Gb, Vb, Ub).

prob_start(kron(Ga, Gb), i(Va, Vb)):-
   prob_start(Ga, Va), prob_start(Gb, Vb).

prob_stop(kron(Ga, Gb), i(Va, Vb)):-
   prob_stop(Ga, Va), prob_stop(Gb, Vb).
```

The above definition of Kronecker product differs from the Kronecker product given in §5.2.1, only in the parametrization of the connectivity with graph identifiers.

The generalized random walk kernel [Vishwanathan et al., 2010] is expressed as:

$$k(G, G') = \sum_{k=0}^{\infty} \mu(k) \mathbf{q}_\times^\top W_\times^k \mathbf{p}_\times \tag{5.16}$$

where $W_\times^k$ is the k^{th} power of $W_\times$. The element related to the `i(Va, Vb)`$^{\text{th}}$-row and `i(Ua, Ub)`$^{\text{th}}$-column of $W_\times^k$ represents the similarity between simultaneous length

k random walks [Vishwanathan et al., 2010]. While $\mu(k)$ is a factor that weighs the contribution, that the paths of length k give to the similarity.

Different definitions of the parameter $\mu(k)$ lead to different instances of random walk graph kernels. We specify in kProbLog the geometric variant. The geometric random walk graph kernel between two graphs G and G' is obtained by setting $\mu(k) = \lambda^k$.

$$k(G, G') = \mathbf{q}_{\times}^{\top} \sum_{k=0} \lambda^k W_{\times}^k \mathbf{p}_{\times} = \mathbf{q}_{\times}^{\top} (I - \lambda W_{\times})^{-1} \mathbf{p}_{\times}. \tag{5.17}$$

Vishwanathan et al. [2010] propose different methods to compute such kernel value. For our kProbLog specification we choose fixed-point iterations in which Equation 5.17 is rewritten as:

$$k(G, G') = \mathbf{q}_{\times}^{\top} \mathbf{x} \tag{5.18}$$

$$(I - \lambda W_{\times})\mathbf{x} = \mathbf{p}_{\times}. \tag{5.19}$$

where $\mathbf{x}$ is an unknown and can be solved using the iterative update rule [Vishwanathan et al., 2010]:

$$\mathbf{x}_{t+1} = \mathbf{p}_{\times} + \lambda W_{\times} \mathbf{x}_t. \tag{5.20}$$

until the fixed point is reached. We shall assume that $\lambda = 0.5$ and specify the iterative update of Equation 5.20 as:

```
:- declare(lambda/0, real).
:- declare(x_sol/2, real, destructive).
:- declare(geometric_rw_kernel/2, real).
0.5::lambda.
x_sol(kron(Ga, Gb), i(Va, Vb)):-
  vertex(Ga, Va), vertex(Gb, Vb), randn(0, 0.001).

lambda_w_product_x(GraphKron, I):-
  lambda, edge(GraphKron, I, J), x_sol(GraphKron, J).

x_sol(GraphKron, I):-
  @addition[lambda_w_product_x(GraphKron, I),
  p_product(GraphKron, I)].
```

The geometric random walk graph kernel is then specified according to Equation 5.20 as:

```
geometric_rw_kernel(Ga, Gb):-
  q_product(kron(Ga, Gb), I),
  x_sol(kron(Ga, Gb), I).
```

5.4 kProbLog$^{\epsilon}$: dual numbers for automatic differentiation

The main purpose of this section is to show that is possible to perform automatic differentiation in kProbLog. Automatic differentiation has a fundamental role in

nowadays machine learning. Recent advances in deep learning have lead to a the proliferation of many frameworks for automatic differentiation such as Torch [Collobert et al., 2002], Theano [Bastien et al., 2012] and TensorFlow [Abadi et al., 2015]. It is beyond the scope of this work to develop a complete automatic differentiation tool, however it would be an interesting future work to integrate one of these frameworks in the implementation of the language.

In this section we show how to use the semiring of dual numbers and the gradient semiring (a generalization of dual numbers) [Eisner, 2002, Kimmig et al., 2011] in kProbLog for gradient descent learning.

A dual number $x + \epsilon x'$ is composed by a primal part x and a dual part x' where ϵ is the nilpotent element (i.e. $\epsilon^2 = 0$). For a variable the dual part is equal to one (i.e. $x' = 1$) while for constants it is equal to zero (i.e. $x' = 0$).

Example 11. *Let $f(x)$ and $g(x)$ be two real valued functions over reals with derivatives $f'(x)$ and $g'(x)$ respectively. We have the following rules for the derivative of combined functions:*

1. sum rule $\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$,
2. product rule $\frac{d}{dx}(f(x)g(x)) = f(x)g'(x) + f'(x)g(x)$.

We use dual numbers and represent f and g together with their derivatives as $y = f(x) + \epsilon f'(x)$ and $z = g(x) + \epsilon g'(x)$.

According to the algebra of dual numbers we have that:

1. sum $y + z = f(x) + g(x) + \epsilon(f'(x) + g'(x))$,
2. product $yz = f(x)g(x) + \epsilon(f(x)g'(x) + f'(x)g(x))$.

We observe that the dual part of $y + z$ and yz are the combined-function derivatives that we obtained with the sum rule *and the* product rule *respectively.*

Dual numbers are generalized to gradients by introducing multiple nilpotent elements $\epsilon_1, \ldots, \epsilon_n$ such that $\epsilon_i \epsilon_j = 0,\ \forall i, j$. The gradient number $x + \epsilon_1 x'_1 + \ldots + \epsilon_n x'_n$ combines the primal part x with n partial derivatives $x'_1, \ldots, x'_n$.

In kProbLog we denote the nilpotent element ϵ with the compound term `eps(index_term)`, where the argument `index_term` is some term that is used to index distinct partial derivatives. The meta-function `@grad/2` takes as inputs a dual number y and a nilpotent element ϵ_x and outputs the partial derivative $\frac{\partial y}{\partial x}$.

Example 12. *Differentiation of a quadratic form. Let us assume that we have a quadratic form $f(\mathbf{x}) = \mathbf{x}^\top A\mathbf{x}$ where $A = \left[\begin{smallmatrix} 2 & 1 \\ 6 & 3 \end{smallmatrix}\right]$ and we want to compute its gradient ∇f in $\mathbf{x}_0 = \left[\begin{smallmatrix} 2 \\ 1 \end{smallmatrix}\right]$.*

```
:- declare([x0/1, a/2], float).
:- declare(grad_f/1, float).
:- declare(dim/1, term).
:- declare([x/1, f/0], grad).

2::a(0, 0). 1::a(0, 1).
6::a(1, 0). 3::a(1, 1).

2::x0(0). 1::x0(1).

x(I):- x0(I).
eps(I)::x(I):- range(I, 0, 2).

f :-
  a(I, J), x(I), x(J).

eps(Dim)::dim(Dim):-
  range(Dim, 0, 2).

grad_f(I):-
  range(I, 0, 2),
  @grad[f, dim(I)].

query(grad_f(_)).
```

Where we defined `x(I)` as `x0(I)` $+\epsilon$.

The output of this program is:

```
20.0::grad_f(1).
15.0::grad_f(0).
```

The gradient semiring adds to kProbLog support for automatic differentiation and can naturally be employed for gradient descent learning.

A very natural task to express in kProbLog is matrix factorization [Nickel et al., 2011, Koren et al., 2009, Kim and Candan, 2011].

Example 13. *Koren et al. [2009] propose a basic factorization model. Users and items are mapped to a joint f-dimensional factor space. The interaction between an item and a user is modeled as the inner-product between their representations in the factor space.*

Each user u is associated with a vector $\mathbf{q}_u \in \mathbb{R}^f$ while each item i is associated with a vector $\mathbf{p}_i$ and r_{ui} is the rating given by user u to item i. The goal is to approximate the rating r_{ui} with a score derived by the inner-product between $\mathbf{q}_u$ and $\mathbf{p}_i$. Koren et al. [2009] use the mean squared error between the predicted score and the rating r_{ui} and regularize the factor representations of users and items ($\mathbf{q}_u$ and $\mathbf{p}_i$) with the ℓ_2-norm.

In kProbLog we can represent $\mathbf{q}_u$, $\mathbf{p}_i$ and r_{ui} declaring the predicates:

```
:- declare([q/2 p/2], grad).
:- declare(r/2, real).
```

The rating predicate `r/2` *is initialized according to the available data. While the initial weight of* `q/2` *and* `p/2` *will be a dual number whose primal number is initialized with small random values, to break symmetries and whose dual part ϵ_f is identified by a nilpotent element indexed by the factor index i.e.:*

```
p(Item, Factor):- randn(0, 0.001).
eps(Item, Factor)::p(Item, Factor).

q(User, Factor):- randn(0, 0.001).
eps(User, Factor)::p(User, Factor).
```

The cost predicate `cost/0` *is defined in terms of* `q/2` *and* `p/2` *and then is differentiable.*

```
:- define(lambda/0, real).
:- define([score/2, cost_mse/2, cost_reg_user/0, cost_reg_item/0, cost/0], grad).

1::lambda. % COST HYPERPARAMETER
score(User,Item):-
  q(User, Factor), p(Item, Factor).

cost_mse(User, Item):- % MEAN SQUARED ERROR
  @square[@subtract[r(User, Item), score(User, Item)]].

cost_reg_user:- % L2-REGULARISER
  @square[q(User, Factor)].

cost_reg_item:- % L2-REGULARISER
  @square[p(Item, Factor)].

cost:- cost_mse(User, Item).
cost:- lambda, cost_reg_user.
cost:- lambda, cost_reg_item.
```

In the above program we specified the cost `cost/0` *function to minimize as a sum of the mean squared error* `cost_mse(User, Item)` *and an* ℓ_2*-norm regularizer* `cost_reg(User, Item)` *weighted by a hyper-parameter* `lambda` *that we set to* 1. *by default. We can express* `cost/0` *using mathematical formulae as follows:*

$$cost = \sum_{u,i} \underbrace{(r_{ui} - score_{ui})^2}_{\texttt{cost\_mse(User,Item)}} + \lambda \underbrace{\left(\sum_{u,f} q_{uf}^2 \right)}_{\texttt{cost\_reg\_user}} + \underbrace{\sum_{i,f} p_{if}^2}_{\texttt{cost\_reg\_item}} \Bigg) . \tag{5.21}$$

The optimization of such a cost function can be performed with gradient descent.

5.5 kProbLog$^{D[\mathbb{S}]}$: ProbLog and aProbLog as special cases

We now clarify the relationship between kProbLog and Problog, and we show that the ProbLog implementation using SDDs of [Vlasselaer et al., 2015] can be emulated by kProbLog. ProbLog is a probabilistic programming language that defines a probability distribution over possible worlds (Herbrand interpretations). A ProbLog program consists of a set of definite clauses c_i and a set of facts labeled with probabilities p_i. While a Prolog query can either succeed or fail, ProbLog computes the success probability of a query. The success probability of a query is the sum of the probabilities of all the possible worlds I in which the query q is true, it thus corresponds to the probability that it is true in a randomly chosen possible world.

Figure 5.3: Example of a ProbLog program (on the left) with the enumeration of the possible worlds and their probabilities (on the right).

```
0.5::p(a).
0.6::p(b).
p(c) :- p(a), p(b).
p(d) :- p(a).
p(d) :- p(b).

query(p(_)).
```

Worlds in which `p(c)` is true.

{p(a), p(b)}	0.6×0.5	$= 0.3$
	p(c)	$= 0.3$

Worlds in which `p(d)` is true.

{p(a)}	$0.5 \times (1 - 0.6)$	$= 0.2$
{p(b)}	$(1 - 0.5) \times 0.6$	$= 0.3$
{p(a), p(b)}	0.6×0.5	$= 0.3$
	p(d)	= 0.8

Example 14. *In Figure 5.3 (on the left) we show a ProbLog program in which there are two facts* `p(a)` *and* `p(b)` *with probability labels* 0.5 *and* 0.6 *respectively.* `p(c)` *and* `p(d)` *are defined as the conjunction* `p(a)` $\wedge$ `p(b)` *and the disjunction of* `p(a)` $\vee$ `p(b)` *respectively. On the right we show two tables which compute the probabilities of* `p(c)` *and* `p(d)`*. For* `p(c)` *we have one possible world while for* `p(d)` *there are three possible worlds. For both* `p(c)` *and* `p(d)` *we enumerate the worlds in which they are true and compute their weighted model count.*

To compute the probabilities of queries, ProbLog compiles the logical part of the program into a Boolean circuit and then evaluates this circuit on the probabilities p_i. The circuit is evaluated by replacing disjunctions and conjunctions with sums and products respectively. The compilation process is necessary to cope with the disjoint-sum problem [De Raedt et al., 2007, Kimmig et al., 2011]. For instance, to compute $P(\texttt{p(d)})$ we cannot simply sum up $P(\texttt{p(a)})$ and $P(\texttt{p(b)})$ (two possible explanations/proofs for `p(d)`) as this would lead to a value that is larger than one, but rather we need to compute $P(\texttt{p(a)}) + P(\texttt{p(b)} \wedge \neg\texttt{p(a)})$. The disjoint-sum problem can be solved by representing the Boolean circuit either as a decision diagram (in practice this can be an ordered binary decision diagram (OBDD) [Bryant, 1992] or as a sentential decision diagram (SDD) [Darwiche, 2011]). While the first version of ProbLog [De Raedt et al., 2007] was using OBDDs a more recent work [Vlasselaer et al., 2015] used SDDs.

The key property that makes OBDDs suitable to handle the disjoint-sums problem is *determinism* [Darwiche and Marquis, 2002] which guaranties that conjunctions in OBDDs are mutually exclusive. SDDs are a strict superset of OBDDs which maintains their key property such as determinism, canonicity and polytime composability [Darwiche, 2011].

Algebraic model counting (AMC) generalizes probabilistic model counting to a semiring

S. In kProbLog is possible to employ a semiring $D[\mathbb{S}]$ to specify AMC tasks on an arbitrary commutative semiring $\mathbb{S}$. The semiring of valued decision diagrams $D[\mathbb{S}]$ can be represented using an SDD whose variables are labeled with elements from the commutative semiring $\mathbb{S}$. Valued decision diagrams are similar to PSDD [Kisa et al., 2014], except that values are not necessarily probabilities and they do not necessarily encode probability distributions.

Any ProbLog program can be directly translated into a kProbLog$^{D[\mathbb{R}]}$ program using the semiring of SDDs labeled with probabilities. This is a direct consequence of the fact that the evaluation algorithm of kProbLog generalizes the T_P-compilation with SDDs of Vlasselaer et al. [2015] to arbitrary semirings. If we label kProbLog facts with SDDs we recover the compilation algorithm of Vlasselaer et al. [2015].

Example 15. *We now compare the same program specified in ProbLog with the probability semiring (i.e. ProbLog) and in kProbLog$^{D[\mathbb{R}]}$ with* SDD*s labeled with probabilities:*

ProbLog	*kProbLog$^{D[\mathbb{R}]}$*

`0.5::p(a).`
`0.6::p(b).`
`p(c) :- p(a), p(b).`
`p(d) :- p(a).`
`p(d) :- p(b).`
 	
`query(p(_)).`
	`:- declare(p, sdd(real)).`
	`sdd(0.5, p(a))::p(a).`
	`sdd(0.6, p(b))::p(b).`
	`p(c) :- p(a), p(b).`
	`p(d) :- p(a).`
	`p(d) :- p(b).`
	
	`query(p(_)).`

The semiring values `sdd(0.5, p(a))` *and* `sdd(0.6, p(b))` *represent parametrized* SDD *variables and are in one-to-one correspondence with kProbLog facts.*

The notation `sdd(Value,Atom)::Atom` used for kProbLog is cumbersome and can be replaced by the syntactic sugar `Value::Atom`. In this way the kProbLog$^{D[\mathbb{R}]}$ program becomes syntactically identical to the ProbLog one.

So far we have shown that kProbLog$^{\mathbb{R}}$ can perform probabilistic model counting. This behavior is not enforced by the language as in ProbLog, but is optional (i.e. it is induced by the type declaration `:- declare(p, sdd(real))`).

While kProbLog$^{D[\mathbb{R}]}$ is equivalent to ProbLog, it is also straightforward to represent aProbLog on a semiring S as kProbLog$^{D[\mathbb{S}]}$ using SDDs labeled with semiring values.[9]

[9] Probabilities have *neutral sums* (i.e. for each atom a we have that $p(a) + p(\neg a) = 1$) this property is not verified for semirings in general. This issue is known as the *neutral-sums problem* [Kimmig et al., 2011].

Algebraic model counting is useful for inference tasks and reasoning about possible worlds, but there are some tasks which are nontrivial to express in aProbLog. Examples are linear algebra operations and explicit feature extraction as explained in §5.3.1.

5.6 Experimental evaluation

We now experimentally evaluate kProbLog and show how it can be used as a declarative language for machine learning. The choices that a kProbLog programmer needs to make in order to satisfy a requirement are quite different from the ones that an imperative programmer would do. While an imperative programmer would have to use different data structures to meet the software requirements, a kProbLog user can just specify the requirements with logical rules. For instance, when moving from a directed to an undirected graph, imperative programmers would have to change their data structure, while in kProbLog it suffices to simply add an extra rule to capture the symmetry of undirected graphs.

kProbLog is well suited for prototyping. As we will show below on an example in graph kernels (cf. **E2**), using kProbLog makes it easy to compose existing programs in order to construct new ones that combine the strengths of both. Different graph kernels take into account of different structural aspects. For example, the Weisfeiler-Lehman subtree kernel can capture degree centrality while shortest path kernels do not. On the other hand, shortest path kernels are a natural choice if one wants to capture patterns with distant nodes. While also the Weisfeiler-Lehman subtree patterns can capture distant nodes, the number of iterations required to do so, could lead to diagonal kernels. Since both these kernels can easily be specified in kProbLog (as we will show), it is also straightforward to create a a hybrid graph kernel possessing the strengths of both underlying kernels.

Another powerful construct of kProbLog are the meta-functions. In a machine learning context, meta-functions can be exploited as a flexible and expressive instrument for describing rich families of kernels. In this sense, meta-functions can be interpreted as a powerful generalization of common kernel hyper-parameters, lifting them from simple numbers to functions. We will show in **E1** how meta-functions can be exploited to explore multiple feature spaces against the same logical specification and provide a rich class of feature spaces.

Our experiments address the following questions:
Q1 Can we use meta-functions to explore multiple feature spaces against the same kProbLog specification and increase the classification accuracy?

Kimmig et al. [2011] explain how to overcome the *neutral-sums problem* by modifying the evaluation of a Boolean circuit.

Q2 Can kProbLog produce hybrid kernels that combine the strengths of existing ones?
Q3 Are the results obtained with kProbLog in line with the state of the art?

5.6.1 Datasets

We empirically validate some kProbLog specifications on the following natural language and bioinformatic datasets:

QC [Li and Roth, 2002] is a dataset about question classification and contains 5500 training and 500 test questions from the TREC10 QA competition. Question classifiers are often used to improve the performance of question answering systems. Indeed, they can be used to provide constraints on the answer types and determine answer selection strategies. QC labels questions according to a two-layer taxonomy of answer types. The taxonomy contains 6 coarse classes (ABBREVIATION, ENTITY, DESCRIPTION, HUMAN, LOCATION and NUMERIC VALUE) and 50 fine classes.

Example 16. *Perhaps, the sentence:*

What films featured the character Popeye Doyle ?

is labeled in QC *as* ENTITY *since we expect films in the answer.*

In order to be comparable with the existing literature, we adopted the coarse grained labels as classification targets.
MUTAG [Debnath et al., 1991] is a dataset of 188 mutagenic compounds labeled according to whether or not they have a mutagenic effect on the Gramnegative bacterium Salmonella typhimurium.
BURSI [Kazius et al., 2005] is dataset of 4337 molecular compounds subdivided in two classes (2401 mutagens and 1936 nonmutagens) determined with the Ames in vitro assay.

5.6.2 Experiments

E1 This experiment was designed to provide an answer to **Q1** and, in particular, to illustrate the expressiveness of kProbLog's meta-functions in an NLP context, where a large number of options are typically available to describe the feature space. It also aims to answer **Q3** since question classification is a typical task where good results using graph kernels have been reported in the literature [Li and Roth, 2002, Zhang and

Lee, 2003]. Each sentence in the QC dataset is tokenized. Perhaps, the sentence in Example 16 is tokenized as the sequence:

[“What”, “films”, “featured”, “the”, “character”, “Popeye”, “Doyle”, “?”].

We define a predicate `token_labels/1`. `token_labels/1` is a unary relation that associates to each token `t` an algebraic label which encodes word, lemma and part of speech (POS) tag of token `t`.

We then use a dependency parser [De Marneffe and Manning, 2008] [10] to extract typed dependency relations between tokens and encode them using the predicate `dep_rel/2`. `dep_rel/2` encodes an edge in the graph of the dependency relations of a sentence, the type of the dependency relation is encoded as algebraic label. The dependency relations between the tokens of the sentence in Example 16 are encoded as:

```
x(det)::dep_rel(1, 0).
x(nsubj)::dep_rel(2, 1).
x(dobj)::dep_rel(2, 6).
x(punct)::dep_rel(2, 7).
x(det)::dep_rel(6, 3).
x(compound)::dep_rel(6, 4).
x(compound)::dep_rel(6, 5).
```

We then define a predicate `dep_rel_edge/2` that casts dependency edges `dep_rel(V, W)` to the shortest path semiring and defines shortest paths on the dependency graph with the predicate `spath/2` (see the definition of the shortest path semiring in Appendix §5.9.2).

```
:- declare([spath/2, dep_rel_edge/2], shortest_paths , additive).
dep_rel_edge(V, W):- @cast_to_shortest_path[dep_rel(V, W)].
spath(V, W):- dep_rel_edge(V, W).
spath(V, W):- V != W, dep_rel_edge(V, U), spath(U, W).
```

We used the `@cast_to_shortest_path/1` meta-function to cast `dep_rel/2` to the shortest path semiring predicate `dep_rel_edge/2`.

We extract unigram and shortest path features with the rules:

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[token_labels(V), config].
feature_blocks:- @decorate_paths[spath(V, W), v2labels, config].
```

the meta-functions `@decorate_vertices` and `@decorate_paths` replace the token indices in unigrams and paths with token labels (i.e. words, lemmas, POS tags or no label) according to the information specified by the algebraic label of the `config` atom. The `config` atom also specifies whether or not edge labels should be placed in the decorated shortest paths. Since the algebraic label of the `config` atom encodes a set of configurations, the output of `@decorate_vertices` and `@decorate_paths` is a multiset of feature blocks represented by the semiring

[10] We used the spaCy Python library to extract lemmas, POS tags and dependency relations.

`polynomial(polynomial(real))` that associates a block of features to each possible configuration specified by the algebraic label of `config`.

Finally all the feature blocks are aggregated together into the algebraic label of `final_features` which corresponds to the feature vector of the sentence.

```
:- declare([final_features/0], polynomial(real)).
final_features:- @aggregate_feature_blocks[feature_blocks].
```

We used the above kProbLog specification to extract features for 127 different configurations (i.e. value assignments of the algebraic label of atom `config`). We considered two kinds of structural features: unigrams and shortest paths. Tokens can be labeled with words (w), lemmas (l), POS tags (p) or not labeled at all (_). There are 8 ways of generating features by combining blocks of unigrams labeled with words, lemmas and POS tags. The possible unigram configurations correspond to the power set of $\{w, p, l\}$. Similarly we have 16 possible ways of combining blocks of shortest path features in which the token indices are replaced by an appropriate token label of type $\{w, p, l, \_\}$, for the p and _ types we also include the edge label information (we add the edge label between labels of consecutive tokens). From the Cartesian product of unigram and shortest path configurations we obtain a total of 128 from which we skip the one without feature blocks. We ran classification experiments using a linear SVM classifier with the C parameter set to 10^4. In Table 5.1 we report the accuracy on the training set obtained with 10-fold cross-validation we show the top 16 best configurations.

Table 5.1: List of the 16 configurations that achieve the highest classification accuracy the training set on QC during the model selection.

unigram features	shortest path features	cross-validated accuracy
lpw	lp	84.6%
lw	lpw	84.4%
lpw	lpw	84.4%
lw	lp	84.4%
lpw	$\_w$	84.3%
lw	$\_lw$	84.2%
lw	$\_lpw$	84.1%
lw	$\_w$	84.1%
unigram features	shortest path features	cross-validated accuracy
lw	$\_l$	84.1%
lpw	l	84.1%
lw	l	84.0%
lpw	$\_lw$	83.9%
lpw	$\_l$	83.9%
lw	pw	83.9%
lpw	$\_lpw$	83.9%
lw	$\_lp$	83.8%

We selected from Table 5.1 the configuration that yields the highest accuracy of 84.6% which is the one that uses lpw for the unigram features and lp for the shortest path features. We retrained on the whole training with the selected configuration and obtained a test accuracy of 91.2% on the test set.

We measured the runtime of the feature extraction and we found that none of the 127 runs on QC exceeds 32 seconds. The measurement of the runtime was performed on a 16 cores machine (Intel Xeon CPU E5-2665@2.40GHz and 96GB of RAM).

E2 In this experiment we mainly aim to answer **Q2** and, in particular, to test the ability of kProbLog to hybridize two well known graph kernels in a context (molecule classification) where they are known to perform well. In order to capture the complementary advantages of the Weisfeiler-Lehman subtree and shortest path kernels, mentioned at the beginning of this section, we shall specify a hybrid kernel. We extract histograms of shortest paths and decorate them with Weisfeiler-Lehman labels. This is where we hybridize the two kernels. The reader should not confuse this kernel with the Weisfeiler-Lehman shortest path kernel [Shervashidze et al., 2011] (explained in § 5.3.4.1) which takes as features pairs of Weisfeiler-Lehman labels together with their

shortest path distance.

We encode each molecule in the MUTAG dataset with the `vertex/1` and the `edge_asymm/2` predicates which represent atoms labeled with atom symbols and chemical bonds labeled with their type respectively. Differently the graph of the dependency relations of a sentence, molecules are naturally represented as undirected graphs so we define the predicate `edge/2` as the symmetric closure of `edge_asymm/2`.

```
:- declare([edge/2], polynomial(real)).
edge(V, W):- edge_asymm(V, W).
edge(V, W):- edge_asymm(W, V).
```

We use the `@cast_to_shortest_path/1` meta-function to cast `edge/2` to the shortest path semiring and generate shortest paths.

```
:- declare([spath/2, edge_sp/2], shortest_paths , additive).
edge_sp(V, W):- @cast_to_shortest_path[edge(V, W)].
spath(V, W):- edge_sp(V, W).
spath(V, W):- V != W, dep_rel_edge(V, U), spath(U, W).
```

We generate the Weisfeiler-Lehman labels of the vertices in the graph.

```
:- declare([wl/2, wl_multiset/2], polynomial(real)).
wl(0, V):- @id[vertex(V)].
wl_multiset(H, V):- edge(V, W), wl(H, W).
wl(H, V):- 0 < H, H <= MAX_ITER, @id[wl(H-1, V), wl_multiset(H-1, V))].
```

We create a predicate `v2wl/1` whose atoms `v2wl(H)` associate to the $\mathtt{H}^{\text{th}}$ iteration of the Weisfeiler-Lehman algorithm a dictionary that maps vertices `V` of the graph to their Weisfeiler-Lehman feature at step `H`.

```
:- declare([v2wl/1], polynomial(polynomial(real))).
v2wl(H):- wl(H, V), @poly_var[V].
```

where the meta-function `@poly_var/1` creates a polynomial variable $x_{\mathtt{V}}$ indexed by the term `V`.

We decorate vertices and shortest paths in a molecule with Weisfeiler-Lehman labels.

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[wl(H, V)].
feature_blocks:- @decorate_paths[spath(V, W), v2wl(H)].
```

And this is the step in which the hybridization happens.

Finally we aggregate the resulting feature blocks using a normalize sum normalize schema.

```
:- declare([normalized_features/0], polynomial(real)).
normalized_features:- @block_normalize_sum_normalize(feature_blocks)
```

For BURSI we use the same kProbLog specification of MUTAG, but we impose K_SP_MAX= 2 as maximum path length. So, we updated the shortest path predicate spath/2 to spath/3 as follows:

```
:- declare([spath/3, edge_sp/2], shortest_paths).
edge_sp(V, W):-
    @cast_to_shortest_path[edge(V, W)].
spath(1, V, W):-
    edge_sp(V, W).
spath(K, V, W):-
    V != W, 2 <= K, K <= K_SP_MAX, edge_sp(V, U), spath(K-1, U, W).
```

Consequently the feature_blocks/0 predicate is updated to:

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[wl(H, V)].
feature_blocks:- @decorate_paths[spath(K, V, W), v2wl(H)].
```

For both MUTAG and BURSI we set maximum number of Weisfeiler-Lehman iterations to MAX_ITER= 1 and ran our kProbLog specification. We made classification experiments using 10 fold cross-validation and measured the classification accuracy and area under the ROC curve for MUTAG and BURSI respectively. We repeated 10 times the 10 fold cross-validations and we obtained an average accuracy of 91.1% with a standard deviation of 0.9% for MUTAG and an average area under the roc curve of 0.902 with a standard deviation of 0.001 for BURSI. For both datasets we used a linear SVM classifier with the C parameter set 1. We measured the runtime on the same hardware used in **E1**. The runtime for MUTAG was 32 seconds while BURSI was 5 minutes and 7 seconds.

All the experiments can be reproduced by running the code provided with the kProbLog implementation (see § 5.2.3).

5.6.3 Discussion

We now answer the experimental questions:

A1 In **E1** we explored a parametrized feature space for QC, using different combinations of words, lemmas, POS tags we could list the 16 best parameterizations in Table 5.1. Since the best results are in line with the results reported by [Li and Roth, 2002] and [Zhang and Lee, 2003], we conclude that meta-functions are a valid language construct to parametrize the feature space. The 91.2% of accuracy obtained on QC with the experiments in **E1** is in line with the results reported by [Li and Roth, 2002] and [Zhang and Lee, 2003].

A2 Shervashidze et al. [2011] experimented on MUTAG with 8 different graph kernels

and achieved the highest accuracy (87.3 ± 0.6) with shortest path kernels, while the accuracy obtained with the Weisfeiler-Lehman subtree kernel is 82.1 ± 0.4 (see Table 1 [Shervashidze et al., 2011]). As anticipated in the beginning of this section, the Weisfeiler-Lehman subtree kernel and the shortest path kernel capture different topological aspects. In experiment **E2**, thanks to the declarative nature of kProbLog, we made a hybrid and labeled shortest paths with Weisfeiler-Lehman colors. We experimented with this kernel on MUTAG and obtained an accuracy of $91.1 \pm 0.9\%$, which is significantly higher than the ones individually achieved by the shortest path and Weisfeiler-Lehman subtree kernels. In **E2**, we also experimented on BURSI with the same hybrid kernel and obtained 0.902 ± 0.001 of area under the ROC, this result is line with those reported in Table 1 of [Costa and De Grave, 2010].

A3 The 91.2% of accuracy obtained in **E1** on QC are in line with the ones reported by [Li and Roth, 2002] and [Zhang and Lee, 2003]. The $91.1 \pm 0.9\%$ of accuracy obtained with our hybrid kernel in **E2** on MUTAG is significantly higher than the ones obtained with 8 different graph kernels in [Shervashidze et al., 2011]. Also the 0.902 ± 0.001 area under the ROC curve obtained in **E1** on BURSI is in line with the results reported by Costa and De Grave [2010]. For these reasons, we conclude that kProbLog can be used to specify kernels that work well on real-world application domains. The runtimes measured are reasonable and show that kProbLog is usable in practice. Feature extraction on QC and MUTAG took less than a minute while on BURSI took less than 6 minutes.

5.7 Related work

In the introduction, we claimed that kProbLog can express models for tensor-based operations, for kernels, and for probabilistic programs; we also mentioned approaches such as Dyna and aProbLog. We now discuss related work along these lines.

First, kProbLog is able to combine logic with tensors and can express tasks such as matrix factorization. As such kProbLog is related to Tensor Relational Algebra [Kim and Candan, 2011], which combines tensors with relational algebra and which was successfully employed for tensor decomposition. However, tensor relational algebra does not support recursion and is therefore less expressive than kProbLog.

Secondly, and perhaps most importantly, kProbLog can be used to declaratively specify a wide range of relational and graph kernels and feature extraction problems using polynomial semirings. As such it is related to the kLog system [Frasconi et al., 2014], which has focused on the specification of relational learning problems and provides a framework to map them into graph-based learning problems via a procedure called *graphicalization*. In conjunction with a graph kernel, kLog can construct feature

vectors associated with tuples of objects in relational domains. However, kLog does not provide support for *programming* the kernel itself, it uses a built-in kernel (the NSPDK [Costa and De Grave, 2010]) or defers the kernel specification to external plugins. kLog and kProbLog are therefore complementary languages. Furthermore, by adopting kProbLog in kLog one would obtain a statistical relational learning system in which the kernel could be declaratively specified as well. Also Gärtner et al. [2004] contributed kernels within a typed higher-order logic in which individuals (the examples) are represented as terms and the kernel definitions, specified in a lambda calculus, exploit the syntactic structure of these example representations. While this also yields a declarative language for specifying kernels on structured objects, it does neither involve the use of semirings nor was it applied to other modeling tasks such as those involving probabilistic reasoning.

Finally, kProbLog is an algebraic logic programming system building upon aProbLog [Kimmig et al., 2011] and Dyna [Eisner et al., 2004, Eisner and Filardo, 2011]. The relationships to these languages are quite subtle and more technical. Nevertheless, distinguishing features of kProbLog are that it supports A) multiple semirings, B) meta-functions, C) additive and destructive updates, D) algebraic model counting, and E) its semantics are rooted in logic programming theory (using an adaptation of the T_P-operator [Vlasselaer et al., 2015]).

On the other hand, aProbLog [Kimmig et al., 2011] is a generalization of the probabilistic programming language ProbLog [De Raedt et al., 2007] to semirings. ProbLog and other statistical relational learning formalisms are based on a possible world semantics on weighted model counting. The key contribution of aProbLog is that it generalizes weighted model counting to algebraic model counting [Kimmig et al., 2012] based on commutative semirings instead of the probabilistic semiring. kProbLog extends aProblog in that it supports multiple semirings (A), meta-functions (B) and destructive as well as additive updates (C). Furthermore, kProbLog (in particular the kProbLog$^{D[\mathbb{S}]}$) replicates aProbLog by performing AMC on a semiring S using the semiring of SDDs whose variables are labeled with values which belong to the semiring S. Furthermore, aProbLog was conceived for algebraic reasoning about possible worlds, while kProbLog main design goal was the specification of tensor algebra and feature extraction problems.

A second closely related language is Dyna [Eisner et al., 2004, Eisner and Filardo, 2011], a language that was initially conceived as a semiring weighted extension of Datalog for dynamic programming. Dyna has been developed for quite a while and is a fairly complex language supporting many different extensions of the basic algebraic Datalog. While kProbLog builds upon Dyna's ideas, Dyna does not support meta-functions (B), destructive updates (C), and algebraic model counting (D). W.r.t. D), Dyna has not dealt with the disjoint-sum problem occurring in probabilistic and algebraic logics such as ProbLog and aProbLog. Furthermore, the semantics of Dyna have been specified in a more informal way in [Eisner and Blatz, 2007] using the

definition of a *valuation function* and although [Eisner and Blatz, 2007, Eisner and Filardo, 2011] relate Dyna's semantics to a T_P-operator; Dyna's T_P-operator is not formally defined in these papers (E).

5.8 Conclusions

We proposed kProbLog, a simple algebraic extension of Prolog that can be used for declarative machine learning, most importantly, for kernel programming. Indeed, using polynomials and meta-functions allows to elegantly specify many recent kernels (e.g. the Weisfeiler-Lehman graph kernel, propagation kernels and GIKs) in kProbLog.

We further introduced in the language the semiring of dual numbers so that kProbLog can also express gradient descent learning, while the semiring of dual numbers allowed us to specify matrix factorization. We showed how the semiring of decision diagrams allows to capture aProbLog (and so ProbLog and, hence, probabilistic programming) as a fragment of kProbLog.

All these features make kProbLog a language in which the user can combine rich logical and relational representations with algebraic ones to declaratively specify models for machine learning. Our experimental evaluations showed that kProbLog can be applied to real world datasets, obtaining good statistical performance and runtimes.

5.9 Appendix

5.9.1 Proof of Theorems

of Theorem 1. In line 4 of Algorithm 2 the ground program GROUND(P) is subdivided into n strata, where n is finite and never exceeds the total number of ground atoms in GROUND(P). Strata are visited in sequence (lines 5-26), for each stratum the for loop (lines 14-26) applies the algebraic T_P-operator exactly once for each ground acyclic rule, then the loop at lines 14-26 produces no side effects Algorithm 2 and terminates. The loop on cyclic rules (lines 14-26) does not produce side effects on w, because the loops at lines 16, 18 and 21 are never executed since $CYCLIC$ is empty for acyclic programs. □

of Theorem 2. The proof of Theorem 2 is identical to the one of Theorem 1 except that $CYCLIC$ is not empty for some strata. We just need to prove that when a stratum P_i is a cyclic kProbLog$^{\mathbb{S}_i}$ program on an ω-continuous semirings $\mathbb{S}_i$ the loop at lines 14-26 terminates. Since when there are no meta-functions, lines 15-25 implement an update of the atom weights w according to Eq 5.1 which corresponds to a step of the Kleene iteration in a system of polynomial equations. Because $\mathbb{S}_i$ is ω-continuous the termination of the loop at lines 14-26 is guaranteed by Corollary 1. □

5.9.2 Shortest path semiring

The shortest path semiring is a variant of the tropical semiring that keeps track of the set of shortest paths corresponding to a given shortest path distance.

The elements $a \in \mathcal{S}$ of the shortest path semiring $(\mathcal{S}, \oplus, \otimes, 0_s, 1_s)$, are sets of strings over the vocabulary V of the vertex identifiers. All the strings in a must have the same length $len(a)$

Let $a, b \in \mathcal{S}$ sum and product are defined as follows:

$$a \oplus b = \begin{cases} a & \text{if } len(a) < len(b) \\ b & \text{if } len(a) > len(b) \\ a \cup b & \text{if } len(a) = len(b) \end{cases} \tag{5.22}$$

$$a \otimes b = \{\text{CONCAT}(s_a, s_b) | s_a \in a \wedge s_b \in b\} \tag{5.23}$$

where CONCAT is the string concatenation operator.

Additive and multiplicative identity are the empty set $\emptyset$ and the singleton set $\{\epsilon\}$ containing the empty string ϵ respectively.

Chapter 6

Weisfeiler-Lehman Graph Morphing

6.1 Introduction

Graphs have been successfully exploited to learn from structured domains such as biochemistry, natural language, robotics etc. While there is a considerable amount of literature about learning with graph kernels on structured domains, the problem of graph generation with kernel methods has been much less investigated. We propose a *constructive mining* technique that allows us to synthesize new graphs and can be applied to a number of different domains such as: *de novo* synthesis of small molecules, automatic level generation in video games, floor plan generation in robotics, etc.

Let us consider the task of drug design, we may want to create a new druggable molecule which has some desirable properties such as low toxicity. We can devise a constrained optimization problem that, given a set of input molecules, which are known to be druggable, creates a new molecule that is similar to the input molecules and is not toxic. Molecules can be represented as graphs while the non-toxicity could be encoded as a constraint on the output graph.

This task can be generalized to *graph morphing* which is the act of transforming a graph G_1 into another graph G_2 through a seamless transition. The graphs that are produced during such transition are hybrids that are *close* with respect to some graph distance d from the original graphs G_1 and G_2. Graph morphing is an act of *computational creativity* that is successful when the morphed graphs are novel, i.e. diverse from the original graphs. Naturally, the morphing problem can also be generalized to a dataset

of n graphs $G_1, \ldots, G_n$.

We formulate graph morphing as a constrained optimization problem that searches in the space of colored graphs:

$$G^\star = \operatorname*{argmin}_G \sum_{i=1}^{n} \gamma_i d(G, G_i) \quad \text{s.t.} \quad f(G) = \mathbf{y} \quad (\mathbf{C1}). \tag{6.1}$$

The above equation morphs a graph G by minimizing its γ_i-weighted sum of distances $d(G, G_i)$ from the input graphs G_i.

A human expert could use this optimization problem to interactively synthesize new molecules by manually setting nonnegative values of γ_i and making the morphed graph closer to the molecules G_i that are most relevant for the task.

We call our method Weisfeiler-Lehman graph morphing because we choose to define the distance between two graphs $d(G, G')$ in Eq. 6.1 as the squared ℓ_2-norm in the feature space induced by Weisfeiler-Lehman subtree kernel (WLST) [Shervashidze et al., 2011] i.e.

$$d(G, G') = \frac{1}{2} \|\phi(G) - \phi(G')\|_2^2 \tag{6.2}$$

where $\phi(G)$ and $\phi(G')$ are the Weisfeiler-Lehman features of G and G' respectively.

The choice of using Weisfeiler-Lehman graph kernel features is well motivated by chemical engineering applications since WLST can take into account of the compatibility between the number of incident edges (bonds) and vertex label (atom type). Previous methods tackled graph morphing with *graph edit distance* (GED) and *marginalized kernels between labeled graphs* (MKLG) [Kashima et al., 2003] and could not take into account of this problem [Jiang et al., 2001, Bakır et al., 2004].

The WLST feature space is also convenient because it allows to mathematically decompose the graph morphing of Eq. 6.1 into an *inference* and a *decoding* phase.

Example 17. *In Figure 6.1 we show the steps that we take in order to morph two graphs G_1 and G_2. Weisfeiler-Lehman graph morphing proceeds as follows: ① the Weisfeiler-Lehman features of the original graphs are extracted, ② during the inference phase the Weisfeiler-Lehman features $\phi^\star$ of the morphed graph are computed and finally ③ since inference guarantees that $\phi^\star$ has a preimage, $\phi^\star$ is decoded (i.e. materialized) to a morphed graph $G^\star$.*

While existing methods require to materialize the graph during the morphing, we morph graphs in the WLST feature space using global optimization for the inference phase and materialize the morphed graph only once i.e. during the decoding phase.

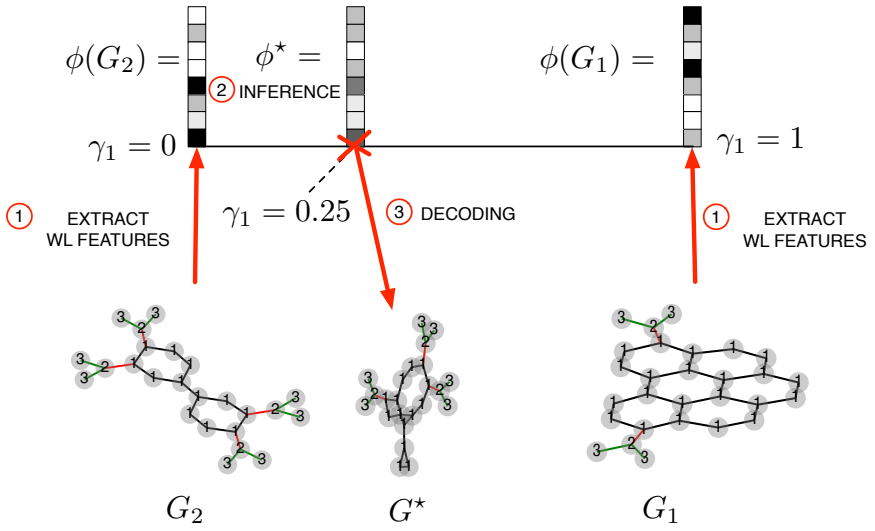


Figure 6.1: We show the Weisfeiler-Lehman graph morphing of two graphs G_1 and G_2 using as coefficients $\gamma_1 = 0.25$ and $\gamma_2 = 1 - \gamma_1$. Vectors of features are pictorially represented as columns of square cells filled with different shades of grey used to represent counts on substructures. The morphing proceeds with the following steps: (1) the Weisfeiler-Lehman features $\phi(G_1)$ and $\phi(G_2)$ of graphs G_1 and G_2 are extracted respectively, (2) the Weisfeiler-Lehman features $\phi^\star$ of the graph to be morphed are computed during the inference, (3) since inference guarantees that $\phi^\star$ has a preimage we can decode $\phi^\star$ to a morphed graph $G^\star$.

The simple and elegant mathematical properties of WLST allow us to compute Eq. 6.1 exactly with off-the-shelf solvers rather than using genetic algorithms or stochastic optimization as done by Jiang et al. [2001] and Kashima et al. [2003] respectively. Most importantly, we can lift graph properties such as *preimage existence* and *simplicity* to linear and quadratic constraints in the WLST feature space respectively. Preimage existence ensures that the image of some colored graph $G^\star$ actually exists (i.e. $\exists G^\star : \phi(G^\star) = \phi^\star$) so that the graph can be materialized by the decoding phase. Simplicity can be used to infer a vector of WLST features that can be decoded to a simple graph.

Moreover, since we express inference as an optimization problem in the feature space we can easily incorporate linear constraints in Eq. 6.1. Continuing our molecule synthesis example, we could constrain the morphed graph to be nontoxic or satisfy certain ADME properties. [1] While handcrafting these properties in the WLST feature

[1] ADME is an abbreviation that designates the pharmacokinetic properties *absorption*, *distribution*, *metabolism* and *excretion*.

space would be difficult task it is possible to learn them from examples. For example we could collect a dataset of molecules with toxicity/non-toxicity labels, extract WLST features and train a linear classifier (e.g. an SVM) that is able to predict nontoxic ones. We can then constrain the morphed graphs to be classified as nontoxic by imposing that their features lie in the positive half-space of the classification hyperplane. Consequently, during the decoding phase we materialize only graphs with the desired properties.

The chapter is organized as follows. In § 6.2 we provide some notation (§ 6.2.1), the necessary background on the Weisfeiler-Lehman algorithm (§ 6.2.2) and subtree kernel (§ 6.2.3). In § 6.3 we introduce the color neighborhood graph kernel (CNGK) (§ 6.3.1). CNGK is a simpler version of the Weisfeiler-Lehman graph kernel that we use as an auxiliary construct to simplify the analysis of the Weisfeiler-Lehman graph kernel preimage problem. In § 6.3.2 we explain how to derive the linear constraints that guarantee that the result of inference can actually be decoded to a graph (see § 6.3.2.1) and the quadratic constraint needed to guarantee that is the image the inferred feature vector is the image of a simple graph (see § 6.3.2.2). The inference problem is then encoded (§ 6.3.3) as a *quadratically constrained quadratic program* (QCQP). The provided results hold for CNGK and are extended to WLST in § 6.4.1 then the preimage problem is encoded in § 6.4.2 as a *constraint satisfaction program* (CSP). We then extend our method in order to handle also colored edges (§ 6.5). In § 6.6 we experiment with our method for graph morphing on some application domains and discuss the results. In § 6.7 we review the contributions that are most closely related to our work. We draw our conclusions in § 6.8.

6.2 Background

We now introduce some notation for graph theory and provide background on graph kernels summarizing the Weisfeiler-Lehman (WL) algorithm and the Weisfeiler-Lehman subtree kernel (WLST).

6.2.1 Graph theory

A *graph* G is a pair (V, E) where V is a set of vertices and $E \subseteq V \times V$ a set of edges. A graph is *undirected* if $(v, v') \in E \Leftrightarrow (v', v) \in E$, in this case we write $\{v, v'\} \in E$. When E is a multiset of edges then G is called *multigraph*. Two edges are *parallel* if they share the same endpoints. A graph is *simple* if it has no *loops* (vertices are not connected to themselves) and a pair of vertices can be connected by at most one edge. A *colored graph* $G = (V, E, \ell)$ is a graph (V, E) with a color function $\ell : V \cup E \to \Sigma$

that maps vertices and edges to a color alphabet Σ. Without loss of generality we assume that a strict ordering relation $<$ is defined between the colors of Σ. [2] Two graphs G and G' are isomorphic $G \simeq G'$ if there is bijection $f : V \to V'$ between their vertices that preserves the edges (i.e. $\{v, u\} \in E \Leftrightarrow \{f(v), f(u)\} \in V'$).

6.2.2 Weisfeiler-Lehman algorithm

The Weisfeiler-Lehman algorithm (WL) [Weisfeiler, 1976] is an isomorphism test for colored graphs that iteratively refines the colors $\mathcal{L} : V \to \Sigma$ of a graph as follows:

$$\mathcal{L}_l(v) = \begin{cases} \ell(v) & \text{if } l = 0, \\ id([\mathcal{L}_{l-1}(v)] \parallel \text{SORTED}([\mathcal{L}_{l-1}(u) | u \in \mathcal{N}(v)])) & \text{if } l > 0, \end{cases} \tag{6.3}$$

where ID : $\Sigma^* \to \Sigma$ is an injective function that maps a string on the color alphabet Σ to a color. The function ID takes as argument a string of colors which derives from the $\parallel$-concatenation of $[\mathcal{L}_{l-1}(v)]$ (where $\mathcal{L}_{l-1}(v)$ is the color of v at the previous iteration) and the lexicographically sorted string SORTED$([\mathcal{L}_{l-1}(u) | u \in \mathcal{N}(v)])$ of the colors $\mathcal{L}_{l-1}(u)$ of the neighbors $u \in \mathcal{N}(v)$ of v at the previous iteration. If at some iteration l the WL colors of two graphs G and G' are not identical, then G and G' are not isomorphic. The reverse does not hold, indeed two non-isomorphic graphs could have the same WL colors for all the iterations.

6.2.3 Weisfeiler-Lehman graph kernel

The Weisfeiler-Lehman subtree kernel (WLST) [Shervashidze et al., 2011] represents a colored graph $G = (V, E, \ell)$ as the Weisfeiler-Lehman graph sequence $\{G^l_{\text{WL}}\}^L_{l=0} = \{(V, E, \mathcal{L}_l)\}^L_{l=0}$ and computes the similarity between two graphs G and G' as a sum of base kernels $k_l(G, G')$ as follows:

$$k_{\text{WL}}(G, G') = \sum_{l=0}^{L} k_l(G, G'). \tag{6.4}$$

The base kernel $k_l(G, G')$ is computed on the multisets of WL colors of G and G' at iteration l:

$$k_l(G, G') = \sum_{\substack{v \in V \\ v' \in V'}} [\![\mathcal{L}_l(v) = \mathcal{L}_l(v')]\!] = \sum_{\substack{g \in \mathcal{R}_t^{-1}(G) \\ g' \in \mathcal{R}_t^{-1}(G')}} [\![g \simeq g']\!]. \tag{6.5}$$

The similarity $k_l(G, G')$ is a convolution kernel on discrete data structures which uses a decomposition relation $\mathcal{R}_l(g, G)$. A pattern $g \in \mathcal{R}_l^{-1}(G)$ is a subtree of height l rooted

[2] For example we could associate to the colors in Σ distinct integers. If i and j are distinct integers

in a vertex v of G. WLST exploits the fact that for rooted subtrees, the compressed labels $\mathcal{L}_l(v)$ and $\mathcal{L}_l(v')$ are equivalent to an isomorphism certificate for g and g' respectively (i.e. $\mathcal{L}_l(v) = \mathcal{L}_l(v') \Leftrightarrow g \simeq g'$).

6.3 Graph morphing in the feature space

In order to formulate the preimage problem in the WLST feature space we first define the *color neighborhood graph kernel* (CNGK) as an auxiliary construct. We first formulate the preimage problem in the CNGK feature space and then we generalize the results provided in this section to WLST features in § 6.4.1.

6.3.1 Color neighborhood graph kernel

Definition 25. *A* color neighborhood *is a pair* $(c, \mathcal{I}) \in \Sigma \times \mathbb{N}^{\Sigma}$ *where* $c \in \Sigma$ *is the root color and* $\mathcal{I} \in \mathbb{N}^{\Sigma}$ *is the multiset of neighbor colors.*

Differently from MKLG patterns, color neighborhood patterns can capture the fact that the number of incident edges must agree to the vertex labels. Indeed, a color neighborhood pattern stores both the color/label of a vertex and the colors/labels of its neighbors with their respective cardinalities.

Definition 26. *A* color neighborhood embedding g^v *in a colored graph* $G = (V, E, \ell)$ *is a subtree* $g^v = (V^v, E^v, \ell)$ *of* G *rooted in* $v \in V$ *with height* 1 *(i.e.* $V^v = \{v\}$, $E^v = \{\{v, u\} : u \in \mathcal{N}(v)\}$*).*

Definition 27. *The* color neighborhood graph kernel *(*CNGK*) computes the similarity measure* $k_{\text{CN}}(G, G')$ *between two colored graphs* G *and* G' *as the inner-product between their histograms of color features (*$\phi_{\text{CN}}(G)$ *and* $\phi_{\text{CN}}(G')$ *respectively) and is defined as follows:*

$$k_{\text{CN}}(G, G') = \langle \phi_{\text{CN}}(G), \phi_{\text{CN}}(G') \rangle = \sum_{i=1}^{\eta} \phi_{\text{CN}_i}(G) \phi_{\text{CN}_i}(G') \tag{6.6}$$

where $\phi_{\text{CN}} : \mathcal{G} \to \mathbb{N}^{\eta}$ *is a feature map that returns an* η*-dimensional vector of counts. The components of* $\phi_{\text{CN}}(G)$ *are* $\phi_{\text{CN}_i}(G) = |\{v \in V(G) | \text{CN}(v, G) = i\}|, \forall i = 1, \ldots, \eta$ *where* η *is the number of distinct color neighborhoods in the input graphs* G *and* $\text{CN} : V \times \mathcal{G} \to \{1, \ldots, \eta\}$ *is a function that associates to a vertex* $v \in V(G)$ *the index of its color neighborhood.*

associated to c and c' respectively, we have that $c < c'$ iff $i < j$, $c' < c$ iff $j < i$ and $c = c'$ iff $i = j$.

We reformulate the graph morphing problem of Eq. 6.1 in the feature space using a graph kernel map $\phi : \mathcal{G} \to \mathbb{N}^\eta$. The input graphs $\{G_i\}_{i=1}^n$ are mapped to the vectors $\mathbf{h}_1, \ldots, \mathbf{h}_n$ (i.e. $\mathbf{h}_i = \phi(G_i)\ \forall i = 1, \ldots, n$). The optimized graph G is represented as a vector of variables $\mathbf{h} \in \mathbb{N}^\eta$ on which we must enforce the sufficient conditions for the existence of the preimage (i.e. $\exists G : \mathbf{h} = \phi(G)$). We rewrite Eq. 6.1 as a *constrained optimization problem* in the feature space $\mathbb{N}^\eta$:

$$\begin{aligned} \mathbf{h}^\star = \operatorname*{argmin}_{\mathbf{h} \in \mathbb{N}^\eta} & \sum_{i=1}^{n} \frac{\gamma_i}{2} \|\mathbf{h} - \mathbf{h}_i\|_2^2 \\ \text{s.t.} \quad & W\mathbf{h} \geq \mathbf{0} \qquad (\mathbf{C1}) \\ & \exists G \in \mathcal{G} : \mathbf{h} = \phi(G) \quad (\mathbf{C2}) \end{aligned} \tag{6.7}$$

where we made the assumption that the distance of graph G from a given input graph G_i can be expressed in the feature space as the squared Euclidean distance $d(G, G_i) = \frac{1}{2}\|\phi(G) - \phi(G_i)\|_2^2$.[3]

For the classifiers represented by $f(G)$ we choose the linear model $f(G) = \text{SIGN}(W\phi(G))$ where W is a weight vector that can be trained on the dataset $\mathcal{D}$ (perhaps with linear SVMs, logistic regression etc.) and it is constant in Eq. 6.7. Because we want to generate examples that are predicted as positive by all the classifiers we use the symbol $\geq$. The constraint **C2** is not present in Eq. 6.1 and was added to Eq. 6.7 to guarantee that the solution $\mathbf{h}^\star$ can be decoded to a graph. In § 6.3.3 we explain how to encode **C2** with linear and quadratic constraints.

6.3.2 Preimage existence

We discuss the sufficient and necessary conditions that a histogram of color neighborhoods $\mathbf{h}$ must satisfy to guarantee the existence of a preimage (i.e. a graph G such that $\mathbf{h} = \phi_{\text{CN}}(G)$).

Assumption 1. *The simplifying assumption **S1** holds when among the input graphs $\{G_i\}_{i=1}^n$ there is not a graph G_i with monochromatic edges (i.e. edges $\{v, u\} \in E$ whose endpoints have the same color $\ell(v) = \ell(u)$).*[4]

The application of a *monochromatic-edge breaking* procedure on the input graphs $\{G_i\}_{i=1}^n$ ensures that assumption **S1** holds. The reason for assumption **S1** will be made clear after the definition of Eq. 6.9.

[3] Bakır et al. [2004] formulated this optimization problem in the feature space of MKLG.

[4] Because the color neighborhoods are induced from the input graphs $\{G_i\}_{i=1}^n$, if **S1** holds then for all the color neighborhoods $(c, \mathcal{I})$ we have that $c \notin \mathcal{I}$.

Given a colored graph $G = (V, E, \ell)$, every monochromatic edge $\{v, v'\} \in E$ such that $\ell(v) = \ell(v')$ is replaced with the edges $\{v, u\}$ and $\{u, v'\}$, where $u \notin V$ is a new vertex of color $\ell(u) = \blacksquare$. The special color $\blacksquare$ is used as placeholder. This transformation preserves all the information in the graph and can be reverted in a successive step, once the preimage is computed. From now on we shall assume that **S1** holds without loss of generality.

A histogram of color neighborhoods $\mathbf{h}$ has a preimage graph G iff the following properties hold:

- **countable** (color neighborhoods): $\mathbf{h} \in \mathbb{N}^\eta$ is a vector of counts,
- **zero-sum** (colored edges): the number of (c, c')-colored edges *generated* by color neighborhoods $(c, \mathcal{I})$ such that $c' \in \mathcal{I}$ must be equal to the number of (c', c)-colored edges *generated* by color neighborhoods $(c', \mathcal{I}')$ such that $c \in \mathcal{I}'$ (§ 6.3.2.1),
- **pigeonhole** (vertex pairs and edges): the number of $\{c, c'\}$-colored edges with $c \neq c'$ that is *generated* by combining color neighborhoods rooted in c and c' must be less then or equal to the number of *possible* $\{c, c'\}$-edges (§ 6.3.2.2).[5]

6.3.2.1 Zero-sum property and the color incidence matrix

The zero-sum property is encoded as the linear constraint on the histogram of color neighborhoods $\mathbf{h}$

$$K\mathbf{h} = \mathbf{0}_f \tag{6.8}$$

where $K \in \mathbb{Z}^{f \times \eta}$ is a matrix in which each $j^{th}-$column is a color incidence vector $\mathbf{k}_j \in \mathbb{Z}^f$.

We now define color incidence vectors $\mathbf{k}_j \in \mathbb{Z}^f$ so that they are into one-to-one correspondence color neighborhoods $(c_j, \mathcal{I}_j)$.

The $(i, j)^{th}$-element of K is associated to a sorted color pair (c_i, c_i') (i.e. $c_i < c_i'$) and is defined as follows:

$$k_{ij} = \begin{cases} & \text{card}(c_i', \mathcal{I}_j) & \text{if } c_i = c_j \wedge c_i' \in \mathcal{I}_j, \\ - & \text{card}(c_i, \mathcal{I}_j) & \text{if } c_i' = c_j \wedge c_i \in \mathcal{I}_j, \\ & 0 & \text{otherwise.} \end{cases} \tag{6.9}$$

where $\text{card}(c_i', \mathcal{I}_j)$ ($\text{card}(c_i, \mathcal{I}_j)$) is the cardinality of color c_i' (c_i) in the interface $\mathcal{I}_j$ (i.e. the number of colored edges $\{c_i, c_j\}$ ($\{c_i', c_j\}$) in the color neighborhood $(c_j, \mathcal{I}_j)$).

[5] The number of *possible* $\{c, c'\}$-edges is equal to the product of the count of color neighborhoods $(c, \mathcal{I})$

When **S1** holds the three branches of Eq. 6.9 are mutually exclusive. By contradiction if **S1** does not hold we can have that $c_i = c'_i$, in this case it is impossible to determine the sign of k_{ij} except for the trivial case in which $k_{ij} = 0$.

The color incidence matrix can be decomposed as $K = K^+ - K^-$ where $K^+ = \max(K, 0)$ and $K^- = -\min(K, 0)$.[6] We can use K^+ to compute the histogram $\mathbf{f}$ of the colored edges which is $\mathbf{f} = K^+\mathbf{h}$. The value of the i^{th}-component of $\mathbf{f}$ is the number of edges $\{v, v'\}$ with color $\{\ell(v), \ell(v')\} = \{c_i, c_{i'}\}$:

$$\sharp\{c_i, c_{i'}\} = \sum_{j=1}^{\eta} k_{ij}^+ h_j = \sum_{j=1}^{\eta} k_{ij}^- h_j.^7 \tag{6.10}$$

Example 18. *We consider the graph G represented as $\circ - \bullet - \circ - \bullet - \circ$ with color alphabet $\Sigma = \{\circ, \bullet\}$ and strict ordering relation $\circ < \bullet$. In G there are $\eta = 3$ kinds of color neighborhoods $(\circ, \{\{\bullet\}\})$, $(\circ, \{\{\bullet, \bullet\}\})$ and $(\bullet, \{\{\circ, \circ\}\})$ and $f = 1$ colored edges $\{(\circ, \bullet)\}$.*

The color incidence matrix K and the color histogram $\mathbf{h}$ of G are:

$$\begin{array}{rccccc} & (\circ,\{\{\bullet\}\}) & (\circ,\{\{\bullet,\bullet\}\}) & (\bullet,\{\{\circ,\circ\}\}) & & \\ K = & [\quad 1 & 2 & -2 &] & (\circ,\bullet) \\ \mathbf{h}^\top = & [\quad 2 & 1 & 2 &] & G \end{array} \tag{6.11}$$

Indeed, because $\mathbf{h}$ is the color neighborhood feature of graph G it satisfies $K\mathbf{h} = 0$.

We only have one type of colored edge (i.e. $(\circ, \bullet)$) and its count is $K^+\mathbf{h} = [4]$ (because $K^+ = [1 \quad 2 \quad 0]$).

6.3.2.2 Pigeonhole principle and the colored edges

The pigeonhole property is satisfied when we guarantee that the number of generated colored edges is less than or equal to the number of possible colored edges. While in section § 6.3.2.1 we computed the number of colored edges *generated* by a histogram of color neighborhoods $\mathbf{h}$ using the color incidence matrix K, in this section we define the sign incidence matrix S that we use to determine the number of *possible* colored edges induced by a histogram of color neighborhood $\mathbf{h}$. We use S to encode the pigeonhole property for colored graphs.

The sign incidence matrix $\text{SIGN}(K)$ is a matrix $S \in \mathbb{S}^{f \times \eta}$ where the j^{th}-column is a sign incidence vector $\mathbf{s}_j \in \mathbb{S}^f$ associated to a color neighborhood $(c_j, \mathcal{I}_j)$ and

such that $c' \in \mathcal{I}$ and the count of the color neighborhoods $(c', \mathcal{I}')$ such that $c \in \mathcal{I}'$

[6] Where $\max$ and $\min$ are applied element-wise.

[7] Notice that: $K\mathbf{h} = 0 \Leftrightarrow K^+\mathbf{h} - K^-\mathbf{h} = \mathbf{0}_f \Leftrightarrow K^+\mathbf{h} = K^-\mathbf{h}$.

$\mathbb{S} = \{-1, 0, 1\}$. The i^{th}-element s_{ij} of the sign incidence vector $\mathbf{s}_j$ is associated to the color pair (c_i, c'_i) such that $c_i < c'_i$ and is defined as follows:

$$s_{ij} = \begin{cases} 1 & \text{if } c_i = c_j \wedge c'_i \in \mathcal{I}_j, \\ -1 & \text{if } c'_i = c_j \wedge c_i \in \mathcal{I}_j, \\ 0 & \text{otherwise.} \end{cases} \tag{6.12}$$

The element s_{ij} is positive (negative) when color c_i (c'_i) is the root of $(c_j, \mathcal{I}_j)$.

As we did for the color incidence matrix K, we decompose S as $S = S^+ - S^-$ where $S^+ = \max(S, 0)$ and $S^- = -\min(S, 0)$.

Given a colored graph (V, E, ℓ) the total number of vertices $v \in V$ with of color $\ell(v) = c_i$ ($\ell(v) = c'_i$) that can be connected with at least a vertex of color c'_i (c_i) is:

$$\sharp(c_i \to c'_i) = \sum_{j=1}^{\eta} s^+_{ij} h_j \quad \left(\sharp(c'_i \to c_i) = \sum_{j=1}^{\eta} s^-_{ij} h_j \right) \tag{6.13}$$

where i is the index of the sorted colored edge (c_i, c'_i) ($c_i \neq c'_i$ because of **S1**) and s^+_{ij} (s^-_{ij}) are the elements of the positive (negative) sign matrix S^+ (S^-).

Theorem 3. *Let (V, E, ℓ) be a colored graph with histogram of color neighborhoods $\mathbf{h} \in \mathbb{N}^\eta$ and color incidence matrix K then (V, E, ℓ) is simple if and only if*

$$\sharp\{c_i, c_{i'}\} \leq \sharp(c_i \to c_{i'}) \sharp(c_{i'} \to c_i), \ \forall i = 1, \ldots, f. \tag{6.14}$$

Equivalently in matrix form we can write:

$$K^+ \mathbf{h} \leq (S^+ \mathbf{h}) \odot (S^- \mathbf{h}). \tag{6.15}$$

Proof. For any given colored edge $\{c_i, c_{i'}\}$ such that $c_i \neq c_{i'}$ the number of generated edges is $\sharp\{c_i, c_{i'}\}$ while the number of the possible edges is $\sharp(c_i \to c_{i'}) \sharp(c_{i'} \to c_i)$. By contradiction if the number of generated edges is greater than the number of possible edges then some edges are parallel and the graph is not simple. □

Example 19 (Example 18 continued)**.** *We consider the same color incidence matrix K of Eq. 6.11 and choose a different histogram of color neighborhoods $\hat{\mathbf{h}}^\top = [0 \quad 1 \quad 1]$. The histogram of color neighborhoods $\hat{\mathbf{h}}$ satisfies $K\hat{\mathbf{h}} = \mathbf{0}_f$ and involves the color neighborhoods $(\circ, \{\{\bullet, \bullet\}\})$, $(\bullet, \{\{\circ, \circ\}\})$. The only possible structure $\hat{G}$ for $\hat{\mathbf{h}}$ has two nodes v, u and a parallel edge which connects them. $\hat{G}$ is the multigraph $\bullet = \circ$. The color incidence matrix K combined with the histogram of color neighborhoods $\hat{\mathbf{h}}$ violate the pigeonhole principle applied to generated versus possible colored edges. If we substitute $K^+ = \begin{bmatrix} 1 & 2 & 0 \end{bmatrix}$, $S^+ = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$, $S^- = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$, and $\hat{\mathbf{h}}$ in Eq. 6.15 we have:*

$$\underbrace{\begin{bmatrix} 1 & 2 & 0 \end{bmatrix}}_{K^+} \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}}_{\hat{\mathbf{h}}} \leq \underbrace{\begin{bmatrix} 1 & 1 & 0 \end{bmatrix}}_{S^+} \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}}_{\hat{\mathbf{h}}} \odot \underbrace{\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}}_{S^-} \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}}_{\hat{\mathbf{h}}} \tag{6.16}$$

$$[2] \leq [1] \odot [1] = [1]. \tag{6.17}$$

As expected the pigeonhole principle on the colored edges is violated indeed $\hat{G}$ is not simple.

6.3.3 The inference optimization problem

We reformulate the optimization problem of Eq. 6.7 as an integer nonconvex *quadratically constrained quadratic program* (QCQP):

$$\begin{aligned} \mathbf{h}^\star = \underset{\mathbf{h}\in\mathbb{N}^\eta}{\operatorname{argmin}} & \sum_{i=0}^{n} \frac{\gamma_i}{2} \|\mathbf{h} - \mathbf{h}_i\|_2^2 \\ \text{s.t.} \quad & W\mathbf{h} \geq \mathbf{0}_q \qquad (\mathbf{C1}) \\ & K\mathbf{h} = \mathbf{0}_f \qquad (\mathbf{C2.1}) \\ & K^+\mathbf{h} \leq (S^+\mathbf{h}) \odot (S^-\mathbf{h}) \qquad (\mathbf{C2.2}) \end{aligned} \tag{6.18}$$

where we encoded the **countable** property as a domain constraint $\mathbf{h} \in \mathbb{N}^\eta$ and the **zero-sum** and **pigeonhole** properties were encoded in **C2** as **C2.1** and **C2.2** respectively. **C2.1** and **C2.2** are the linear (Eq. 6.8) and nonconvex quadratic (Eq. 6.15) constraints explained in § 6.3.2.1 and § 6.3.2.2 respectively. The non-convexity of the formulation arises from the pigeonhole constraint **C2.2**.

To the best of our knowledge QCQP programs are hard to solve and they are usually treated as NP-hard problems [d'Aspremont and Boyd, 2003].

6.3.3.1 A particular case that reduces to convex integer QP

There is a particular case in which the pigeonhole principle always holds which is when K is a sign matrix (i.e. $K = \text{sign}(K)$). When K is a sign matrix constraint **C2.2** can be removed and the optimization problem of Eq. 6.18 becomes an integer convex *quadratic program* (QP).

Corollary 2 (of Theorem 3)**.** *Let (V, E, ℓ) be a colored graph with histogram of color neighborhoods $\mathbf{h} \in \mathbb{N}^\eta$ and color incidence matrix K, if K is a sign matrix then (V, E, ℓ) is always simple.*

Proof. If K is a sign matrix we have that $K = \text{SIGN}(K) = S$. We apply $K^+ = S^+$ and $S^+\mathbf{h} = S^-\mathbf{h}$ to Eq. 6.15 and obtain $S^+\mathbf{h} \leq (S^+\mathbf{h}) \odot (S^+\mathbf{h})$.[8] If $x_i = \sum_{j=1}^{\eta} S_{ij}^+ h_j$ we need to show that $x_i \leq x_i^2, \ \forall i = 1, \ldots, f$ or equivalently that $x_i \in \mathbb{R} \setminus (0, 1), \ \forall i = 1, \ldots, f$. The interval $(0, 1)$ does not contain integers. By contradiction if $x_i \in (0, 1)$ some of the terms $S_{ij}h_j, \ j = 1, \ldots \eta$ must be non-integer and this can not be, because $S_{ij}h_j$ is a product of two integers. □

The color incidence matrix K is a sign matrix whenever all the color neighborhoods $(c, \mathcal{I})$ induced from the input graphs $\{G_i\}_{i=1}^n$ have an interface $\mathcal{I}$ in which all the colors are distinct.

This result has two implications a) it reduces the complexity of inference from NP-hard to NP-complete (as mixed-integer quadratic programming was found to be NP-complete by Del Pia et al. [2014]) and b) it increases the number of the solvers which are able to tackle this kind of problems.

6.4 Weisfeiler-Lehman Subtree Kernel Preimage Problem

So far we solved the preimage existence problem for CNGK, in this section we will generalize this result to WLST and show how to generate preimage graphs from a given graph image $\mathbf{h}$.

6.4.1 Generalization to Weisfeiler-Lehman subtree features

The Weisfeiler-Lehman graph $G_{\text{WL}}^l = (V, E, \mathcal{L}_l)$ is the recolored version of the colored graph $G = (V, E, \ell)$ where l is the number of iterations of the WL algorithm. The color neighborhoods of G_{WL}^{l-1} are in one-to-one correspondence with the colors of G_{WL}^l, indeed we have $\mathcal{L}_l(v) = id([c_i] \parallel \text{SORTED}(\mathcal{I}_i)), \;\; \forall v \in V : i = \text{CN}(v, G_{\text{WL}}^{l-1})$. This equation can be verified by comparing Eq. 6.3 and Definition 25. We used $\text{CN}(v, G_{\text{WL}}^{l-1})$ to identify the index i of the color neighborhood $(c_i, \mathcal{I}_i)$ of vertex v in the colored graph G_{WL}^{l-1}. Given the WL color $\mathcal{L}_L(v)$ of a vertex v at iteration T we can reconstruct all the WL colors $\{\mathcal{L}_l(v)\}_{l=0}^{L-1}$ of vertex v in the previous iterations.[9] The WLST vertex feature $\phi_{\text{WL}}^L(v)$ at iteration L is actually the histogram of the WL colors $\{\mathcal{L}_l(v)\}_{l=0}^{L}$ of v. In Eq. 6.19 we reformulate the inference problem of Eq. 6.18 in the WLST feature space representing the colored graphs G, G_i with histograms of WLST features ϕ and ϕ_i respectively instead of histograms of color neighborhoods $\mathbf{h}$ and $\mathbf{h}_i$ respectively. To map WLST histograms of features ϕ obtained with L iterations to CNGK histograms of color neighborhoods we need to induce the color neighborhoods on colored graphs recolored with $L-1$ iterations of the WL algorithm. In this case we can transform the features by using matrix multiplication $\phi = \chi_{\text{WL}}\mathbf{h}$ where $\chi_{\text{WL}} \in \mathbb{R}^{p \times \eta}$ is a matrix whose i^{th} row corresponds to the WLST vertex feature at iteration L associated (it is a

[8] Notice that $(K^+ = S^+) \wedge (K^- = S^-) \wedge (K^+\mathbf{h} = K^-\mathbf{h}) \Rightarrow (S^+\mathbf{h} = S^-\mathbf{h})$.

[9] Indeed, the WL algorithm in Eq. 6.3 invokes ID function which is injective and its argument contains a

one-to-one correspondence) to the color neighborhood $(c_i, \mathcal{I}_i)$.

$$\begin{aligned} \mathbf{h}^\star = \operatorname*{argmin}_{\mathbf{h}\in\mathbb{N}^\eta} & \sum_{i=1}^{n} \frac{\gamma_i}{2}\|\phi - \phi_i\|_2^2 \\ \text{s.t.} \quad & W\mathbf{h} \geq \mathbf{0}_q \qquad (\mathbf{C1}) \\ & K\mathbf{h} = \mathbf{0}_f \qquad (\mathbf{C2.1}) \\ & K^+\mathbf{h} \leq (S^+\mathbf{h}) \odot (S^-\mathbf{h}) \quad (\mathbf{C2.2}) \end{aligned} \tag{6.19}$$

Substituting $\phi = \chi_{\text{WL}}\mathbf{h}$, $\phi_i = \chi_{\text{WL}}\mathbf{h}_i$, $W_{\text{WL}} = W\chi_{\text{WL}}$ and $C_{\text{WL}} = \chi_{\text{WL}}^\top\chi_{\text{WL}}$ in Eq. 6.19 we obtain:

$$\begin{aligned} \mathbf{h}^\star = \operatorname*{argmin}_{\mathbf{h}\in\mathbb{N}^\eta} & \sum_{i=1}^{n} \frac{\gamma_i}{2}(\mathbf{h} - \mathbf{h}_i)^\top C_{\text{WL}}(\mathbf{h} - \mathbf{h}_i) \\ \text{s.t.} \quad & W\mathbf{h} \geq \mathbf{0}_q \qquad (\mathbf{C1}) \\ & K\mathbf{h} = \mathbf{0}_f \qquad (\mathbf{C2.1}) \\ & K^+\mathbf{h} \leq (S^+\mathbf{h}) \odot (S^-\mathbf{h}) \quad (\mathbf{C2.2}) \end{aligned} \tag{6.20}$$

which is the graph generation problem in the WLST feature space.

6.4.2 Preimage problem

The WLST preimage problem for an image vector $\mathbf{h}$ can be expressed as a CSP. The number of nodes $|V|$ in the preimage graph $G = (V, E, \ell)$ is $|V| = \|\mathbf{h}\|_1$ and there are exactly h_i vertices with color neighborhood $(c_i, \mathcal{I}_i)$. The connectivity of G is represented as the adjacency matrix $\mathbf{A} \in \{0,1\}^{|V|\times|V|}$ whose components $\mathbf{a}_{ij}$ are binary variables. The number of CSP variables is much smaller than $|V|^2$ because: a) $\mathbf{A}$ must be symmetric and b) the vertex pairs $v, v' \in V$ can be connected by an edge only if their color neighborhoods ($(c, \mathcal{I})$, $(c', \mathcal{I}')$ respectively) are compatible (i.e. $\text{COMP}(v, v') = c \in \mathcal{I}' \wedge c' \in \mathcal{I}$). The CSP formulation has the linear constraint

$$\sum_{\substack{v_j \in V\,: \\ \text{COMP}(v_i, v_j) \wedge c_j = \hat{c}}} \mathbf{a}_{ij} \quad = \quad \text{card}(\hat{c}, \mathcal{I}_i), \quad \forall v_i \in V, \forall \hat{c} \in \Sigma \tag{6.21}$$

that is reminiscent of the zero-sum constraint. While the latter is enforced on the histogram of the color neighborhoods $\mathbf{h}$ (i.e. counts on vertices) the former is enforced on the of the vertices v_i (i.e. on the materialized graph). For all vertices $v_i \in V$ the number of compatible neighbors v_j (whose color is $\ell(v_j) = \hat{c}$) must be equal to the number $\text{card}(\hat{c}, \mathcal{I}_i)$ of neighbors of color $\hat{c}$ in the interface $\mathcal{I}_i$.

A current limitation of our method is that we do not provide theoretical results to enforce the generation of *connected* graphs. While this is beyond the scope of the

string that starts with the WL color $\mathcal{L}_{l-1}(v)$ at the previous iteration.

Table 6.1: Comparison between WLST graph morphing and the median graph problem on the median word problem [Jiang et al., 2001].

Word	Input words	Median graph (from [Jiang et al., 2001]) set	Median graph (from [Jiang et al., 2001]) generalized	Morphing with WLST (our method) $L=1$	Morphing with WLST (our method) $L=2$	Morphing with WLST (our method) $L=3$
graph (2.34, **3.22**, **4.07**)	ccaph crcchb gcph gbaph grbph grbh gcabcb grach craph grah	grach (**2.25**, 3.51, 4.47)	graph (2.34, **3.22**, **4.07**)	gcaph (**2.25**)	graph (**3.22**)	gcph (4.19)
matching (3.29, 4.37, 5.43)	matdhidgd datchidg mbtcdig matcding matdhidd ddcding mtchibg mbbchdng matchin baddbbbg	matcding (2.86, 4.13, 5.20)	matching (3.29, 4.37, 5.43)	matchidg (**2.61**)	matchidg (**3.91**)	matchidg (**5.05**)
median (2.61, 3.63, 4.66)	mbbiand medban bciab cedcanb cedian medicn mbdianc mbdiab cbdianc mccan	cedian (2.57, 3.61, 4.64)	mbdian (**2.28**, **3.35**, **4.33**)	mbdian (**2.28**)	mbdian (**3.35**)	mbdian (**4.33**)
genetic (3.37, 4.43, 5.43)	genatbb gbaaaib gebetic genatbc genaabb genaaica enbtib genetbb enbtic gebetica	genatbb (**2.32**, **3.53**, **4.51**)	genatib (2.36, 3.67, 4.79)	genatbb (**2.32**)	genatbb (**3.53**)	genatbb (**4.51**)
search (3.21, 4.19, 5.16)	sedrdh sddbbh seabdh sdardh seddc ddardh sebrchd sedrch bddrch dearch	sedrch (2.35, **3.43**, **4.36**)	sedrch (2.35, **3.43**, **4.36**)	sedrdh (**2.30**)	sedrdh (**3.43**)	sedrch (**4.36**)

chapter, in our experiments (§ 6.6) we will show that this is not a practical limitation. Indeed we could always generate connected graphs.

6.5 Graph morphing with colored edges

So far we did not consider edge colors for the sake of simplicity. However, edge colors are useful for the representation of some domains such as: molecular compounds (to represent the type of chemical bonds between atoms), parse trees (in which the symbols generated by nonterminal nodes are ordered using edge labels), entity-relationship diagrams etc. This part is technical and can be skipped during a first read of the chapter.

Edge colors if present must be part of the propagation process of the Weisfeiler-Lehman algorithm in which the second branch of Eq. 6.3 (i.e. $l > 0$) is modified to:

$$id([\mathcal{L}_{l-1}(v)] \parallel \text{SORTED}([[\ell(\{v,u\}),\ \mathcal{L}_{l-1}(u)] | u \in \mathcal{N}(v)])) \tag{6.22}$$

In the above equation vertex v not only receives as propagated colors the WL colors $\mathcal{L}^{l-1}(u)$ of its neighbors $u \in \mathcal{N}(v)$, but also the color $\ell(\{v,u\})$ of the edge $\{v,u\}$ that propagated the vertex color $\mathcal{L}^{l-1}(u)$.

The introduction of edge colors also affect the color incidence matrix K whose columns are in one-to-one correspondence with color triples (c, c', c_e) (where $c < c'$ and c_e is an edge color) instead of sorted vertex-color pairs (c, c'). The interface $\mathcal{I}$ of a color neighborhood is represented as a multiset of edge/vertex color pairs (c_e, c_v) instead of color pairs.

The color incidence matrix K is a sign matrix if the interface $\mathcal{I}$ of every color neighborhood $(c, \mathcal{I})$ contains edge/vertex color pairs (c_e, c_v) that are all distinct. Therefore the introduction of colored edges allows to solve the inference problem

of Eq. 6.19 as a convex integer QP program [10] on a family of graphs that is more general than the one proposed in § 6.3.3 (in which the vertex colors in the interface $\mathcal{I}$ of the color neighborhoods $(c, \mathcal{I})$ had to be all distinct).

The update of the preimage problem to colored edges requires the introduction of a colored edge tensor $\mathbf{E} \in \{0,\ 1\}^{|V| \times |V| \times |\Sigma|}$ with elements

$$e_{ijk} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \wedge \ell(\{v_i, v_j\}) = c_k, \\ 0 & \text{otherwise.} \end{cases} \tag{6.23}$$

The constraint of Eq. 6.21 must be updated to:

$$\sum_{\substack{v_j \in V\ : \\ \text{COMP}(v_i, v_j, c_k) \wedge c_j = \hat{c}}} \mathbf{e}_{ijk} \quad = \quad \text{card}((c_k, \hat{c}), \mathcal{I}_i), \quad \forall v_i \in V, \forall \hat{c}, c_k \in \Sigma \tag{6.24}$$

where the predicate $\text{COMP}(v_i, v_j, c_k)$ is defined as $c \in \mathcal{I}' \wedge c' \in \mathcal{I} \wedge c_k = \ell(\{v_i, v_j\})$ and $\text{card}((c_k, \hat{c}), \mathcal{I}_i)$ is the cardinality of the edge/vertex color pair $(c_k, \hat{c})$ in the interface $\mathcal{I}$. Because an edge can have at most one color, we also enforce the constraint:

$$\mathbf{a}_{ij} = \sum_{c_k \in \Sigma} \mathbf{e}_{ijk}, \forall v_i, v_j \in V \tag{6.25}$$

where $\mathbf{a}_{ij}$ is the ij^{th}-element of the adjacency matrix $\mathbf{A}$ [11] of the graph that we are materializing. Trivially when $\mathbf{a}_{ij} = 0$ the edge $\{v_i, v_j\}$ is not present and $\mathbf{e}_{ijk} = 0$ for all the colors $c_k \in \Sigma$ (i.e. an edge that does not exists has no color).

6.6 Experimental evaluation

We experiment with our method on the *median word problem* [Jiang et al., 2001] and *graph interpolation* [Bakır et al., 2004].

The median word problem models a word "$w_1 w_2 \dots w_n$" as the sequence $w_1 \rightarrow w_2 \dots \rightarrow w_n$ (which is a directed graph). Jiang et al. [2001] choose 5 words (see the *Word* column in Table 6.1) and attempt to reconstruct each of them from 10 distorted instances (see the *Input words* column in Table 6.1) using *set graph median* and *generalized graph median* as denoisers. The aim of this artificial problem is to mimic the denoising of words obtained with different *optical character recognition* (OCR) softwares. Jiang et al. [2001] point out that the degree of distortion that they applied to their synthetic data is substantially higher than the one obtained in real-world OCR applications, so the problem is harder.

[10] Allowing to remove constraint **C2.2** from Eq. 6.19.

[11] See the adjacency matrix in § 6.4.2.

Bakır et al. [2004] interpolate graphs, randomly selecting two molecules G_1, G_2 from the MUTAG dataset, set $\bar{\phi} = \alpha\phi(G_1) + (1-\alpha)\phi(G_2)$ (for $\alpha = 0, 0.2, \ldots, 1$), and show the preimage graph $G^\star$ whose squared Euclidean distance of $\phi(G^\star)$ from $\bar{\phi}$ is minimal. The method is evaluated by showing that the for each value of α the feature $\phi(G^\star)$ of the reconstructed graph $G^\star$ is closer (w.r.t. the Euclidean norm) to $\bar{\phi}$ than any other graph G_i in the MUTAG dataset.

6.6.1 Experimental questions

We answer to the following experimental questions.

Q1 Is our method able to recover the original word in the median word reconstruction task [Jiang et al., 2001] ?

Q2 Is our method capable of reconstructing the input graphs G_i when $\gamma_i = 1$ and $\gamma_j = 0, \forall i \neq j$ from the MUTAG dataset?

Q3 Can we morph graphs from the MUTAG dataset such that their reconstruction error is lower than any other graph in the dataset? Are the morphed graphs novel?

Q4 What is the effect of the L parameter of the WLST pattern on graph morphing?

Q5 Given that we are working on a hard problem, is the runtime of our method a limitation for its future uses?

6.6.2 Experiments

We perform some experiments in order to answer to our experimental questions.

E1 We use our method to solve the median word problem. The directed graph $w_1 \rightarrow w_2 \ldots \rightarrow w_n$ (which represents the word "$w_1 w_2 \ldots w_n$") is encoded as an undirected graph by replacing each arch $w_i \rightarrow w_{i+1}$ with a node a_i connected to its predecessor by an edge of color 1 and to its successor by an edge of color 2. The directed graph $w_1 \rightarrow w_2 \ldots \rightarrow w_n$ becomes the undirected graph $G_{\mathbf{w}} = w_1 \overset{1}{-} a_1 \overset{2}{-} w_2 \overset{1}{-} \ldots \overset{1}{-} a_{n-1} \overset{2}{-} w_n$.

This representation is used to encode the noisy input words shown in Table 6.1. Our goal is to reconstruct the graph corresponding to the original word (first column of Table 6.1). We run our method in the WLST feature space with parameter $L = 1, 2, 3$ and report our results in Table 6.1. Below each word $\mathbf{w}$ we report (between parenthesis) the 3 values of the reconstruction error $\|\phi(G_{\mathbf{w}}) - \bar{\phi}\|_2$ for $L = 1, 2, 3$ except for the words obtained with our method that are annotated with just 1 value (i.e. the reconstruction error corresponding to the value of L used to generate them). The

runtime of the inference and decoding problems (see Eq. 6.20 and Eqs 6.24, 6.25 respectively) executed to generate the results in Table 6.1 is reported in Table 6.2.

Table 6.2: Runtime of the WLST graph morphing applied to the median word problem [Jiang et al., 2001] (see also Table 6.1).

Word	Morphing with WLST (our method)		
	$L=1$	$L=2$	$L=3$
graph	gcaph inf: 311ms dec: 9ms	graph inf: 104ms dec: 10ms	gcph inf: 82ms dec: 12ms
matching	matchidg inf: 7” dec: 10ms	matchidg inf: 483ms dec: 16ms	matchidg inf: 175ms dec: 19ms
median	mbdian inf: 2” dec: 8ms	mbdian inf: 93ms dec: 18ms	mbdian inf: 160ms dec: 14ms
genetic	genatbb inf: 270ms dec: 9ms	genatbb inf: 88ms dec: 10ms	genatbb inf: 79ms dec: 12ms
search	sedrdh inf: 638ms dec: 7ms	sedrdh inf: 129ms dec: 10ms	sedrch inf: 96ms dec: 15ms

E2 We select from MUTAG [12] two graphs G_1 and G_2 (Figures 6.3, 6.3 respectively) so that they have the same number of vertices of those used by Bakır et al. [2004] (28 and 24 respectively) and run our implementation to generate the following figures:

E2.1 Figure 6.4 we show the morphing results for $\alpha = 0, 0.25, \ldots, 1$ (setting $\gamma_1 = \alpha$ and $\gamma_2 = 1 - \alpha$) obtained with WLST features at resolution $L = 1, \ldots, 7$. Each morphed graph in Figure 6.4 is accompanied by the runtimes for the inference and decoding problems.

E2.2 Figure 6.2 we show the distance $\|\phi(G^\star) - \bar{\phi}\|_2$ of the reconstructed graph $G^\star$ from $\bar{\phi}$ (in blue) versus the distance $D_n = \min_{i=1,\ldots,n} \|\phi(G_i) - \bar{\phi}\|_2$ of the closest graph G_i in the MUTAG dataset from $\bar{\phi}$ (in red) for $\alpha = 0, 0.05, \ldots, 1$ and $L = 1, \ldots, 7$. The smallest is the reconstruction error $\|\phi(G^\star) - \bar{\phi}\|_2$ the more the $G^\star$ graph resembles the input graphs $\{G_i\}_{i=1}^n$.

When D_n is not equal to the reconstruction error $\|\phi(G^\star) - \bar{\phi}\|_2$, the morphed graph $G^\star$ is novel (i.e. is not an element of the MUTAG dataset).

[12] http://mlcb.is.tuebingen.mpg.de/Mitarbeiter/Nino/WL/

Figure 6.2: Distance $\|\phi(G^\star) - \bar{\phi}\|_2$ of the reconstructed graph $G^\star$ from $\bar{\phi}$ (in blue) vs. distance $D_n = \min_{i=1,\ldots,n} \|\phi(G_i) - \bar{\phi}\|_2$ of the closest graph G_i in the dataset from $\bar{\phi}$ (in red). Where $\bar{\phi} = \gamma_1 \phi(G_1) + (1 - \gamma_1)\phi(G_2)$ and $\gamma_1 \in [0, 1]$. In each subplot the distances are normalized dividing by the maximum reconstruction error.

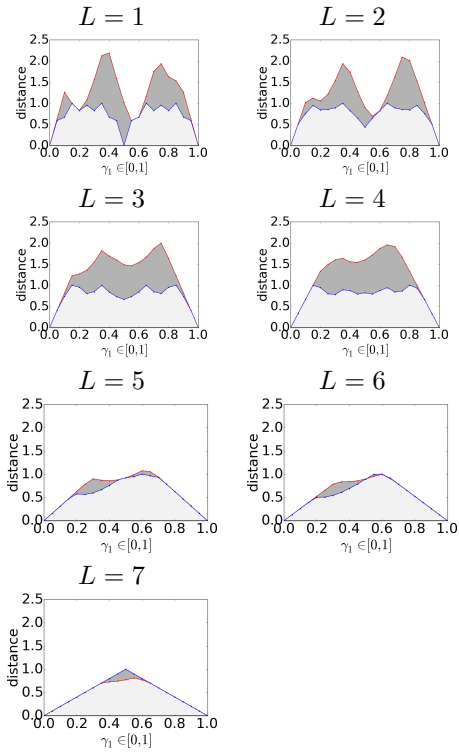
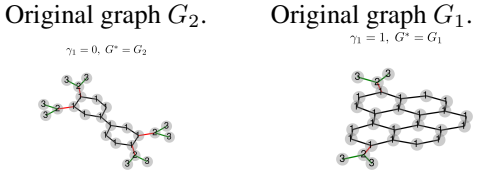


Figure 6.3: Graph morphing with WLST patterns between the MUTAG molecules $G1$ and $G2$. We use $L = 1, \ldots, 7$ iterations of WLST and $\gamma_1 = 0, 0.25, \ldots, 1$.



We implemented our method using the CHOCO 3 solver[13] for both the *inference* and *decoding* the problem (Eq. 6.20 and Eqs 6.24, 6.25 respectively). The implementation of the decoding step also uses the CHOCO-GRAPH[14] extension of CHOCO to graph variables that allows to enforce connectivity. We chose the domain of the histogram of color neighborhood variables in $\mathbf{h} \in \mathbb{N}^{\eta}$ to be lower bounded by 0 and upper bounded by the vector $\mathbf{h}_{ub}$. Where $\mathbf{h}_{ub} = \max_{i=1,\ldots,n:\gamma_i>0} \mathbf{h}_i$ is the element-wise maximum among the histograms of color neighborhoods $\{\mathbf{h}_i\}_{i=1}^n$ of the input graphs $\{G_i\}_{i=1}^n$ with positive coefficient $\gamma_i > 0$. We measured the runtime of our experiments on a MacBook Pro with a 2,9 GHz Intel Core i7 processor and 8 GB of ram.

6.6.3 Discussion

We discuss the experimental results and answer to the experimental questions **Q1-5**.

A1 We could reconstruct the original word from the input words in Table 6.1 for 1 of the 5 original words (i.e. graph). Our system almost always found reconstructed words in the WLST feature space with a reconstruction error which is lower than the one of the original words (the reconstruction errors which are minimal are in bold typeface in Table 6.1). The only exception that we have is the WLST morphing in which we attempt to reconstruct the word graph with the parameter $L = 3$. The reconstructed word gcph has a reconstruction error (4.19) which is higher than the cost (4.07) of the original word graph. This happens because (among the color neighborhood patterns induced from the input words for the word graph), not all the necessary color neighborhoods needed to construct the word graph are present. This can be avoided choosing smaller values of L which generate less and smaller patterns.

A2 From Figure 6.2 we see that when $\gamma_1 = 1$ and $\gamma_2 = 0$ our method generates a histogram of color neighborhoods $\mathbf{h}^\star$ whose corresponding feature $\phi^\star$ has 0 reconstruction error (i.e. $\|\phi^\star - \bar{\phi}\|_2 = 0$). The decoding problem was implemented to materialize graphs $G^\star$ with a given histogram of color neighborhoods $\mathbf{h}^\star$ however there might be multiple graphs which satisfy such condition. In Figure 6.4 we observe that when $L = 6, 7$ and $\gamma_1 = 0$ we obtain a graph $G^\star_{\gamma_1=0}$ isomorphic to G_2, while for $\gamma_1 = 1$ the returned graph $G^\star_{\gamma_1=1}$ is not isomorphic to G_1, but indistinguishable for the WLST patterns. Indeed both $G^\star_{\gamma_1=1}$ have G_1 the same histogram of color neighborhoods (because of the 0 reconstruction error shown in Figure 6.2).

A3 In Figure 6.2 we show that for $L \in [1, 5]$ we can morph graph pairs from the MUTAG dataset with a reconstruction error that is less than or equal to the one of any other graph in the MUTAG dataset. When $L = 6, 7$ the patterns tend to be too large. For $L = 6$ we have only one case (i.e. $\gamma_1 = 0.55$) where the reconstruction error is

[13] http://choco-solver.org/

[14] https://github.com/chocoteam/choco-graph/releases/tag/choco-graph-3.2.1

marginally higher than the best graph found in the MUTAG dataset (see Figure 6.2). While for $L = 7$ the system can only generate the graphs with the same histogram of color neighborhoods of the original graphs, as the size of the WLST patterns is too large. For each plot in Figure 6.2 we can observe that whenever the red line does not touch the blue line the morphed graph is novel (i.e. it is not present in the MUTAG dataset).

A4 In Figure 6.4 for $\gamma_1 = 0.75, 1.0$ we notice that in correspondence of $L = 1, 2$ our method tends to generate chains of nodes, while increasing the value of L it starts to mimic the aromatic (hexagonal) rings which are present in molecule G_1. When the value of L increases our method is able to capture more structural information.

A5 Inference is treated as an NP-hard problem, however our experiments had an acceptable runtime. For the median word problem (see Table 6.2) with $L = 1$ the inference optimization always terminates in at most 7 seconds (for the word matching) while for $L = 2, 3$ it always terminates in at most 483 milliseconds which is a much better runtime. For the graph morphing problem (see Figure 6.4) the inference optimization problem always terminates in less than 184 milliseconds. The runtime of the decoding step is always below 19 ms (see Table 6.2) for and 387 ms (see Figure 6.4) for the median word and the molecule morphing tasks respectively. These time measurement are quite promising for future applications of our method.

6.7 Related works

Jiang et al. [2001] optimize with genetic algorithms the same cost function of Eq. 6.1 but define the dissimilarity $d(G, G_i)$ between the optimized graph G and the input graph G_i with GED, which itself is NP-complete to compute [Jiang et al., 2001]. The chromosome representation used by Jiang et al. [2001] includes G and also all the mappings between G and the input graphs G_i. As for efficiency, each mapping contains the edit operations that are needed to transform G into G_i.

Bakır et al. [2004] define the distance function $d(G, G_i)$ in Eq. 6.1 as the squared Euclidean distance between G and G_i in the feature space induced by MKLG [Kashima et al., 2003]. The decomposition relation employed by MKLG generates labeled paths produced by random walks on graphs [Kashima et al., 2003]. Since they choose the feature space induced by MKLG their method entirely discards the information about the number of nodes in the graph [Bakır et al., 2004]. For this reason the graph generation is decomposed in three steps which are: a) estimation of the number of vertices, b) estimation of the vertex set and c) optimization of the cost function by sampling the adjacency matrix of G.

In recent work, Costa [2016] has combined the graph kernel NSPDK and graph grammars in order to machine learn a probability distribution on graph data and sample

graphs from such distribution. First a probability distribution on graphs is estimated in the NSPDK feature space and then graphs are sampled with Metropolis–Hastings. In order to generate a random walk in the sample space it uses the production rules of a graph grammar. At each step of the random walk a new graph is generated and its NSPDK features are extracted in oder to compute its acceptance ratio.

Finally, a number of works [Ishida et al., 2008, Shimizu et al., 2011, Akutsu et al., 2012] consider the problem of reconstruction of tree-like molecules from histograms of path frequencies.

Our method differs from the existing ones because it uses histograms of counts on WLST patterns, it retains the information about the number of nodes in the graph and can take into account of the fact that the number of edges must agree with the vertex labels. In this regard it differs from Jiang et al. [2001] and Bakır et al. [2004] because the former does not take into account of these problems while the latter is invariant with respect to this information as it uses MKLG features.

Furthermore, our method solves the graph generation problem (inference phase) in the feature space induced by WLST without the necessity to materialize the graph (decoding phase) in the intermediate steps. For this reason, it can use global optimization techniques such as QP/QCQP. For example, we can constrain a morphed graph to lie in the positive half-space of a hyperplane learnt with an SVM. In this regard it is different from the work of Costa [2016] since that method was never extended neither to morph graphs nor to enforce additional constraints.

We finally summarize the principal strengths of our method in Table 6.3.

6.7.1 Demi-degree subsequences

In this section we report some graph-theoretical results on demi-degree subsequences of which our method is reminiscent.

Berge and Minieka [1973] explain how to derive the existence of a p-graph [15] for a given *demi-degree sequence*.

For a directed multigraph $G = (V, E)$ the outer (inner) demi-degree $d^+_G(v)$ ($d^-_G(v)$) of a vertex $v \in V$ is defined as the number of arcs having v as their initial (terminal) endpoint:

$$d^+_G(v) = \sum_{u \in V} m^+_G(v, u) \quad \left(d^-_G(v) = \sum_{u \in V} m^-_G(v, u) \right) \tag{6.26}$$

where $m^+_G(v, u)$ is the multiplicity of the arc (v, u) in E and $m^-_G(v, u) = m^+_G(u, v)$ [Berge and Minieka, 1973].

[15] A p-graph is a directed graph in which each arc $(v, u) \in E$ can appear at most p times.

Table 6.3: Qualitative comparison of the Weisfeiler-Lehman graph morphing with other methods.

	Jiang et al. [2001]	Bakir et al. [2004]	Costa [2016]	our method
similarity/distance	GED	MKLG	NSPDK	WLST
morphing	✓	✓	✗	✓
constraints	✗	✗	✗	linear/ quadratic
vertex label / edge number problem	✗	✗	✓	✓
avoid graph materialization	✗	✗	✗	✓
optimization	stochastic	genetic	MCMC	global (QP/QCQP)

The definition of the color incidence matrix K (see Eq. 6.9) and its decomposition as the difference between K^+ and K^- (see 6.3.2.1) is reminiscent of the concept of *demi-degree* [Berge and Minieka, 1973].

However, the positive (negative) part K^+ (K^-) of the color incidence matrix K differs from the demi-degree $d_G^+(v)$ ($d_G^-(v)$) for the following reasons:

a) K is applied at the level of the induced patterns (color neighborhoods) rather than the graph itself,

b) K is applied to colored undirected graphs, while $d_G^+(v)$, $d_G^-(v)$ are defined for directed graphs,

c) K^+ (K^-) uses a strict ordering relation among vertex colors to determine the sign of its elements, while $d_G^+(v)$ ($d_G^-(v)$) is defined using the direction of the arcs having v as their initial (terminal) point.

To the best of our knowledge, we are the first to develop an exact method that could generate graphs starting from histograms of subtree patterns (in particular WLST patterns).

6.7.2 Color-lifted inference

The reader familiar with lifted inference in *probabilistic graphical models* (PGM) may have noticed some connections with our inference method. The WL algorithm is also known in literature as color passing and naive vertex classification. Kersting et al. [2009] propose to use color passing to group nodes and potentials of a PGM into supernodes and superpotentials thus deriving a compressed PGM. Instead of performing inference on the PGM, a lifted inference algorithm is applied on the compressed PGM. The method is advantageous when the PGM has a lot of symmetries and the compressed version is much smaller. Our inference optimization problem of Eq. 6.20 when applied to graphs colored with $L-1$ iterations of the WL coloring algorithm optimizes on a vector $\mathbf{h}$ of η variables, where η is the number of distinct colors obtained with L iterations of the WL coloring algorithm on the input graphs. The more the symmetries shared by the input graphs, the smaller is the size η of the vector of optimized variables $\mathbf{h} \in \mathbb{N}^{\eta}$ in Eq. 6.20.

6.8 Conclusions and future works

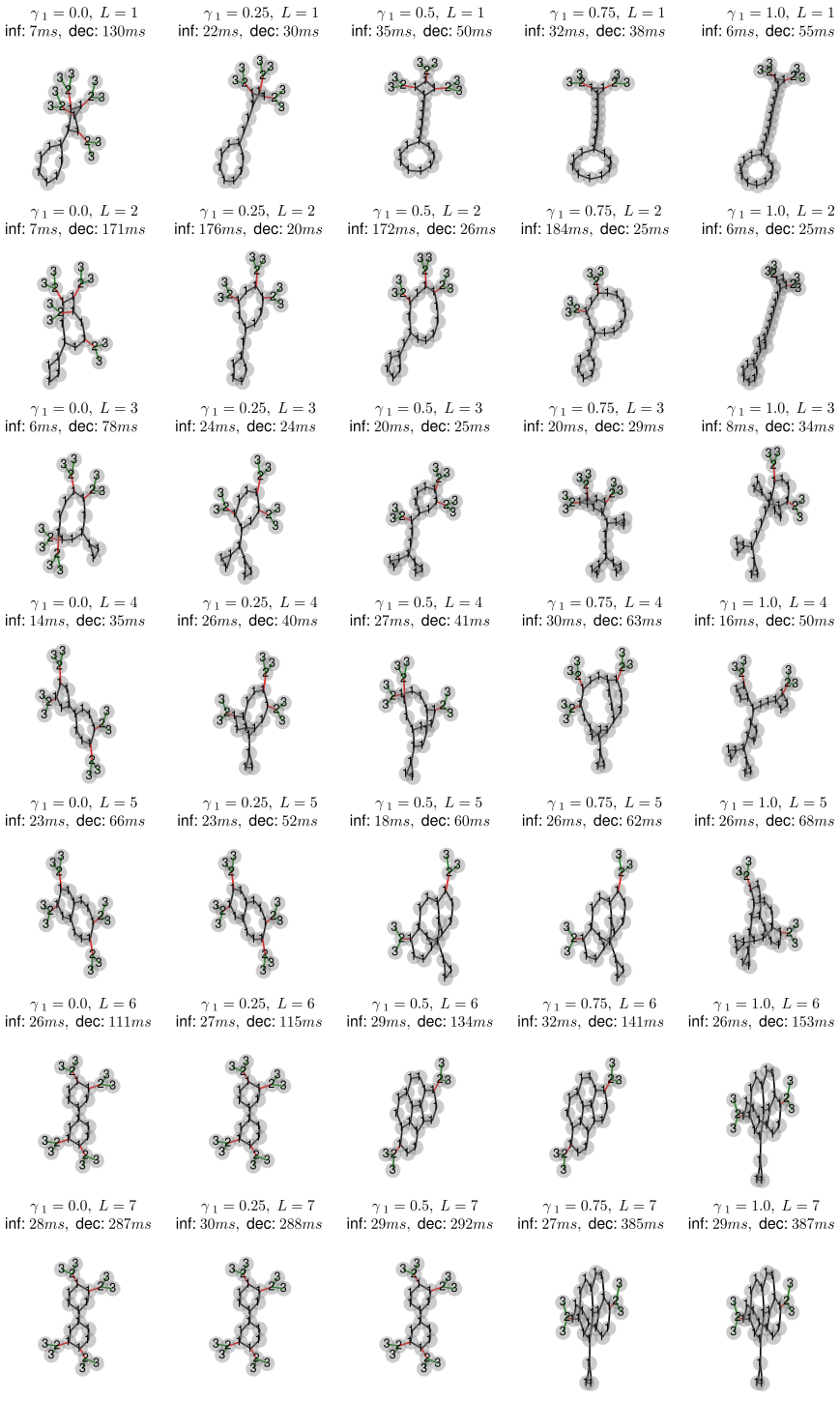
We proposed a method for graph morphing that solves the preimage problem for the WLST kernel. The WLST patterns are interesting for chemical applications as they capture information about the compatibility between the number of incident edges and vertex colors. This is also one of the reasons because they can outperform random walk kernels on datasets of chemical compounds [Shervashidze et al., 2011].

We provided theoretical insights on the WLST kernel that allowed us to lift some graph properties (i.e. preimage existence and simplicity) in the WLST feature space and decompose the graph morphing problem into an inference and a decoding step. This decomposition was advantageous because we could formulate these steps using global optimization with off-the-shelf solvers. Furthermore, our method can support learnt constraints. Indeed, constraining a morphed graph to lie in the positive half-space of a hyperplane learnt with an SVM is trivial because QP and QCQP naturally support linear constraints.

In our experimental evaluation we used WLST patterns to morph new graphs starting from the molecules in the MUTAG dataset and obtained an excellent runtime. For opportune sizes ($L = 1, \ldots, 5$) of the WLST patterns we could morph graphs with a reconstruction error which is lower than the one of the other molecules in the MUTAG dataset, showing that we can morph graphs that are new.

For this reasons our method is very promising for future applications in drug design.

Figure 6.4: γ_1-interpolated graphs with $L = 1, \ldots, 7$ iterations of WLST.



Chapter 7

Conclusions and future work

Often real world data can not be cast into rectangular tables and more natural representations are graphs or logic. In this thesis we addressed some challenges that arise when learning with graphs and logic.

Challenges in learning from structured data arise also in hybrid domains, in which there is the need to merge continuous and discrete structure. Some domains, such as social networks, present a very irregular degree distribution and are particularly challenging for graph kernels and neural networks on graphs that have traditionally been employed in bio-/chemo-informatics and natural language processing. Furthermore, new typologies of datasets may present new challenges which motivated us to design a declarative language for specifying machine learning problems on structured data.

7.1 Learning in hybrid continuous/discrete domains

We addressed the challenge of hybrid continuous/discrete domains and proposed the GIK framework that allowed us to upgrade existing graph kernels to vectors of continuous values. The GIK reformulation of well known graph kernels allowed us to obtain more insights in the exploration of graph kernels with continuous attributes. The underlying idea was to employ vertex invariants for soft subgraph matching. Several graph-kernel instances were then empirically evaluated on a number of new and existing benchmark datasets. The results showed that some combinations of graph and vertex invariants with continuous attributes lead to excellent performance.

7.2 Neural networks for social networks domains

We proposed SAEN, a neural network framework that substantially upgrade and extend the Haussler's convolution relations to hierarchical decompositions. With hierarchical decompositions we introduced a novel notion of depth in the context of learning with structured data, leveraging the nested part-of-parts relation.

Within the context of the SAEN framework we introduced a simple hierarchical decomposition EGNN that uses ego-graphs and experimentally showed that it works particularly well on large graphs with skewed node degree distribution, such as those that naturally occur in social network data. Our approach is also effective for learning with smaller graphs, such as those occurring in bio-/chemo-informatics, although in these cases the performance of SAEN does not exceed the state-of-the-art established by other methods.

Often hierarchical decompositions present symmetries (i.e. repeated substructures) and it is possible to take advantage of them. Another contribution of the SAEN framework is the domain compression algorithm that reuses theoretical results from color lifted inference to compress hierarchical decompositions and greatly reduce the runtime. An empirical evaluation of domain compression showed a great reduction of the memory usage and a considerable speedup the of the training time.

7.3 kProbLog a language for declarative machine learning

The kProbLog language is one possible valid answer to the need for a language for machine learning. kProbLog is a simple algebraic extension of Prolog that can be used for declarative machine learning, most importantly, for kernel programming. We introduced in the language polynomials and meta-functions in order to elegantly specify many recent kernels (e.g. the Weisfeiler-Lehman graph kernel, propagation kernels and GIKs) in kProbLog.

Furthermore, kProbLog can also express gradient descent learning, and probabilistic programming problems. To cover these two aspects we introduced in the language the semiring of dual numbers and the one of the semiring of decision diagrams. The former allowed us to specify matrix factorization problems, the latter to capture aProbLog (and so ProbLog and, hence, probabilistic programming) as a fragment of kProbLog.

All these features make kProbLog a language in which the user can combine rich logical and relational representations with algebraic ones to declaratively specify models for machine learning. Our experimental evaluations showed that kProbLog can be applied

to real world datasets, obtaining good statistical performance and runtimes.

7.4 Future work

Often when comparing the advantages of neural networks over kernel methods we state that neural networks can learn the features while in kernel methods the features need to be handcrafted by a human. However, that statement is only true when data are represented in rectangular tables but false in the relational setting.

Learning on relational datasets requires a technique that allows to adapt the input data to the learning methods. Both kernel methods and neural networks either require to handcraft this adaptation or rely on default choices. For example graph kernels and SAEN require to handcraft $\mathcal{R}$-convolutions and $\mathcal{H}$-decompositions respectively, while the use of convolution windows in CNNs and the unfolding of the computation over the input DAG in RNNs are default choices.

The recent deep-learning hype has attracted a lot of researchers toward neural networks including learning theoreticians. While deep learning is not yet as well understood as convex optimization in kernel methods, its theoretical foundation are being consolidated [Kawaguchi, 2016]. Recent works on deep learning provide better insights on how to structure network architectures and how to train them [Kingma and Ba, 2014, Andrychowicz et al., 2016]. Moreover, the recent proliferation of software frameworks such as Theano and TensorFlow made code reuse easier and improved the reproducibility of the experimental results.

However, human experts are still needed to handcraft architectures that capture the structure in relational data and while existing tools for automated machine learning mostly focus on hyper-parameter optimization [Snoek et al., 2012, Bergstra et al., 2013], we believe that the long-term research goal should be in the direction of program/architecture induction.

In order to accomplish this we will need both a neural knowledge compilation for logic programs and inductive logic programming techniques to learn algebraic logic programs.

7.4.1 Neural knowledge compilation for kProbLog programs

The integration of SAEN inside kProbLog would only be a first step towards a more ambitious project of unifying logic programming and neural networks.

Indeed, ProbLog uses logic circuits (BDDs, SDDs and DNNF) as compilation technology for probabilistic inference, while the compilation technology for kernel design in

kProbLog consists of polynomials and some meta-functions.

An interesting future research direction will be to create a compilation technology that allows us to map logic programs to neural networks. A main technical challenge is to find an architecture able to handle cyclic computations.

However, neither SAEN nor recursive neural networks can handle cyclic computation graphs, in particular SAEN unfolds a feedforward neural network on a hierarchical decomposition and a recursive neural network work on directed acyclic graphs. Sum-product networks [Poon and Domingos, 2011] can be seen as directed acyclic graphs of mixture models and in this sense can not handle cyclic computations.

Differently from the above mentioned architectures, kProbLog$^{\mathbb{S}}$ can handle algebraic cyclic programs and when the semiring is ω-continuous (e.g. the probability semiring) it can be guaranteed to approximate the correct solution in a finite number of steps. Indeed, the problem of the evaluation of this kind of cyclic programs boils down to solving the fixed point computation of a polynomial system of equations, and this can efficiently be accomplished with the Newton method over ω-continuous semirings (cf. Corollary 1 § 5.2.1 and [Esparza et al., 2010, 2014]). The algebraic labels of kProbLog$^{\mathbb{R}}$ programs could then be learnt by gradient descent perhaps using dual numbers to implement automatic differentiation.

Since kProbLog programs can be stratified and the compilation of each stratum can be delegated to different backends, when strata are not cyclic we could also incorporate recursive neural networks and SAEN.

7.4.2 kProbLog program induction

If the coefficients of the polynomials are associated to first order kProbLog$^{\mathbb{R}}$, we could produce a considerable number of rules and then prune them by imposing sparsity on their associated coefficients. We believe that this approach would lead to a scalable inductive logic programming method by gradient descent.

In principle this method could be applied to ProbLog as well. However, ProbLog requires a knowledge compilation step in which arithmetic circuits can grow exponentially thus limiting the number of rules in the search space. Since kProbLog$^{\mathbb{R}}$ has a different semantics and does not deal with the disjoint-sum problem, the scalability of the evaluation would only depend on the size of the ground program and not on the knowledge compilation step, which is not needed by kProbLog$^{\mathbb{R}}$.

However, pruning rules by gradient descent would only solve half of the problem of inductive logic programming in kProbLog because a rule generation technique in that language is still missing. Of course we could adapt existing inductive logic programming techniques for that.

The elegant meta-interpretative learning (MIL) framework recently proposed by Muggleton et al. [2015] has been reported to outperform by a factor of $100 - 1000$ state-of-the-art inductive logic programming systems and it is definitely an interesting direction for the induction of kProbLog programs. However, the adaptation of MIL to kProbLog would require some thinking because algebraic labels and meta-functions bring additional challenges to the problem of the induction of kProbLog programs.

Another recent work for program synthesis is Microsoft PROSE [Polozov and Gulwani, 2015].[1] PROSE allows to synthesize programs from examples by combining domain-specific operators according to the production rules specified by a domain-specific language (DSL). That system efficiently searches in the space of the possible programs because the search is guided by the production rules of the DSL and it uses witness functions to invert the semantics of the domain-specific operators. Witness functions allow the user to specify heuristics to invert the semantics of the domain-specific operators. These heuristics are needed because otherwise the problem of inverting the semantics of the operators might have infinite solutions. Comparing to kProbLog, meta-functions are the analogous PROSE domain-specific operators and a ground kProbLog program would be the equivalent of a PROSE DSL. Currently, kProbLog does not support witness functions nor program synthesis and similar constructs would have to be introduced in the language. However, an inductive extension of kProbLog in this sense would be a more natural starting point than MIL. Indeed, PROSE can already employ operators while in MIL there is not yet the equivalent of a meta-functions.

7.4.3 Constructive machine learning

The efforts spent in GIKs, SAEN and kProbLog are principally made in the direction of providing methods that can learn on structured data by decomposing inputs in substructures that will then be used as features. Nevertheless, also the problem of inverting the feature extraction process (i.e. solving the preimage problem) can have interesting applications to constructive learning tasks. Fascinated by the mathematical beauty and simplicity of the Weisfeiler-Lehman subtree kernel (WLST) we started to work on Weisfeiler-Lehman graph morphing, investigated the properties of WLST and found a formulation that allows us to solve the preimage problem in such feature space. Furthermore, we could implement the Weisfeiler-Lehman graph morphing using off-the-shelf optimization software. While we could show how it can be used to morph molecular graphs and to decode noisy words, in our future work we plan to further consolidate its use in these application domains and extend it for new ones such as natural language, floor plan generation and level generation in video games. A possible extension of the method could be integrated in the kLog [Frasconi et al., 2014] language, indeed the use of logic as description language would give a uniform representation

[1] The Microsoft PROSE framework was formerly known as FlashMeta.

formalism for different application domains. The choice of kLog is well motivated by a peculiar operation performed by the language called *graphicalization* which draws an equivalence between logical interpretations and entity-relationship diagrams (which are bipartite graphs). In this sense, if we know how to construct new graphs we can also construct new logical interpretations. More work will be probably required to find convincing evaluation methods for the outputs generated by this system, perhaps this may also involve domain experts for manual evaluation or the use of simulators.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Tatsuya Akutsu, Daiji Fukagawa, Jesper Jansson, and Kunihiko Sadakane. Inferring a graph from path frequency. *Discrete Applied Mathematics*, 160(10):1416–1428, 2012.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

L. Babai, D. Yu. Grigoryev, and D. M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 310–324, New York, NY, USA, 1982. ACM. ISBN 0-89791-070-2. doi: 10.1145/800070.802206. URL `http://doi.acm.org/10.1145/800070.802206`.

Gökhan H Bakır, Alexander Zien, and Koji Tsuda. Learning to find graph pre-images. In *Pattern Recognition*, pages 253–261. Springer, 2004.

P Baldi and G Pollastri. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *J Mach Learn Res*, 4(Sep):575–602, 2003.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Claude Berge and Edward Minieka. *Graphs and hypergraphs*, volume 7. North-Holland publishing company Amsterdam, 1973.

James Bergstra, Daniel Yamins, and David D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

Anna Maria Bianucci, Alessio Micheli, Alessandro Sperduti, and Antonina Starita. Application of cascade correlation networks for structures to chemistry. *Applied Intelligence*, 12(1-2):117–147, 2000.

Christopher M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

Karsten M. Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*, pages 74–81, 2005. doi: 10.1109/ICDM.2005.132. URL http://dx.doi.org/10.1109/ICDM.2005.132.

Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alexander J. Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. pages 47–56, 2005. doi: 10.1093/bioinformatics/bti1007. URL http://dx.doi.org/10.1093/bioinformatics/bti1007.

Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989. doi: 10.1109/69.43410. URL http://dx.doi.org/10.1109/69.43410.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, IDIAP, 2002.

Fabrizio Costa. Learning an efficient constructive sampler for graphs. *Artificial Intelligence*, pages –, 2016. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/j.artint.2016.01.006.

Fabrizio Costa and Kurt De Grave. Fast neighborhood subgraph pairwise distance kernel. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 255–262, 2010. URL http://www.icml2010.org/papers/347.pdf.

D. Croce, A. Moschitti, and R. Basili. Structured lexical similarity via convolution kernels on dependency trees. In *Proc. of the Conference on Empirical Methods in Natural Language Processing*, pages 1034–1046. Association for Computational Linguistics, 2011.

Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-143. URL `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-143`.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17(1):229–264, 2002.

Alexandre d'Aspremont and Stephen Boyd. Relaxations and randomized methods for nonconvex qcqps. 2003.

M. C. De Marneffe, B. MacCartney, C. D. Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proc. of LREC*, volume 6, pages 449–454, 2006.

Marie-Catherine De Marneffe and Christopher D Manning. The stanford typed dependencies representation. In *Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation*, pages 1–8. Association for Computational Linguistics, 2008.

Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007. URL `http://ijcai.org/Proceedings/07/Papers/396.pdf`.

Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 10(2):1–189, 2016.

Asim Kumar Debnath, Rosa L Lopez de Compadre, Gargi Debnath, Alan J Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991.

Alberto Del Pia, Santanu S Dey, and Marco Molinaro. Mixed-integer quadratic programming is in np. *arXiv preprint arXiv:1407.4798*, 2014.

Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Trans. Knowl. Data Eng.*, 17(8):1036–1050, 2005. doi: 10.1109/TKDE.2005.127. URL http://dx.doi.org/10.1109/TKDE.2005.127.

P. D. Dobson and A. J. Doig. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of molecular biology*, 330(4):771–783, 2003.

Manfred Droste and Werner Kuich. *Semirings and formal power series*. Springer, 2009.

Jason Eisner. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th annual meeting on Association for Computational Linguistics*, pages 1–8. Association for Computational Linguistics, 2002.

Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proc. of Formal Grammar*, pages 45–85, 2007.

Jason Eisner and Nathaniel W. Filardo. Dyna: Extending datalog for modern ai. In *Datalog Reloaded*, pages 181–220. Springer, 2011.

Jason Eisner, Eric Goldlust, and Noah A. Smith. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL), Companion Volume*, pages 218–221, Barcelona, July 2004.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM (JACM)*, 57(6):33, 2010.

Javier Esparza, Michael Luttenberger, and Maximilian Schlund. Fpsolve: A generic solver for fixpoint equations over semirings. In *Implementation and Application of Automata - 19th International Conference, CIAA 2014, Giessen, Germany, July 30 - August 2, 2014. Proceedings*, pages 1–15, 2014. doi: 10.1007/978-3-319-08846-4_1. URL http://dx.doi.org/10.1007/978-3-319-08846-4_1.

Aasa Feragen, Niklas Kasenburg, Jens Petersen, Marleen de Bruijne, and Karsten M. Borgwardt. Scalable kernels for graphs with continuous attributes (see also erratum). In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pages 216–224, 2013.

Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE transactions on Neural Networks*, 9(5): 768–786, 1998.

Paolo Frasconi, Fabrizio Costa, Luc De Raedt, and Kurt De Grave. klog: A language for logical and relational learning with kernels. *Artif. Intell.*, 217:117–143, 2014. doi: 10.1016/j.artint.2014.08.003. URL `http://dx.doi.org/10.1016/j.artint.2014.08.003`.

A d'Avila Garcez, Tarek R Besold, Luc de Raedt, Peter Földiak, Pascal Hitzler, Thomas Icard, Kai-Uwe Kühnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. Neural-symbolic learning and reasoning: contributions and challenges. In *Proceedings of the AAAI Spring Symposium on Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches, Stanford*, 2015.

Artur SD'Avila Garcez, Luis C Lamb, and Dov M Gabbay. *Neural-symbolic cognitive reasoning*. Springer Science & Business Media, 2008.

Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer, 2003.

Thomas Gärtner, John W Lloyd, and Peter A Flach. Kernels and distances for structured data. *Machine Learning*, 57(3):205–232, 2004.

Lise Getoor and Ben Taskar, editors. *Introduction to statistical relational learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 2007. ISBN 978-0-262-07288-5.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

C Goller and A Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2007.

D. Haussler. Convolution kernels on discrete structures. Technical report, Technical report, Department of Computer Science, University of California at Santa Cruz, 1999.

Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA,*

August 22-25, 2004, pages 158–167, 2004. doi: 10.1145/1014052.1014072. URL http://doi.acm.org/10.1145/1014052.1014072.

Yusuke Ishida, Liang Zhao, Hiroshi Nagamochi, and Tatsuya Akutsu. Improved algorithms for enumerating tree-like chemical graphs with given path frequency. *Genome Informatics*, 21:53–64, 2008.

Xiaoyi Jiang, Andreas Münger, and Horst Bunke. An median graphs: properties, algorithms, and applications. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(10):1144–1151, 2001.

Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *ICML*, volume 3, pages 321–328, 2003.

Kenji Kawaguchi. Deep learning without poor local minima. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 586–594. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6112-deep-learning-without-poor-local-minima.pdf.

Jeroen Kazius, Ross McGuire, and Roberta Bursi. Derivation and validation of toxicophores for mutagenicity prediction. *Journal of medicinal chemistry*, 48(1): 312–320, 2005.

Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting belief propagation. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 277–284. AUAI Press, 2009.

Mijung Kim and Kasim Selçuk Candan. Approximate tensor decomposition within a tensor-relational algebraic framework. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 1737–1742, 2011. doi: 10.1145/2063576.2063827. URL http://doi.acm.org/10.1145/2063576.2063827.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3685.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *CoRR*, abs/1211.4475, 2012. URL http://arxiv.org/abs/1211.4475.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2014.

Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, pages 30–37, 2009.

N. Kriege and P. Mutzel. Subgraph matching kernels for attributed graphs. In *Proc. of the 29th ICML*, 2012.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Werner Kuich. Semirings and formal power series: Their relevance to formal languages and automata. In *Handbook of formal languages*, pages 609–677. Springer, 1997.

Niels Landwehr, Andrea Passerini, Luc De Raedt, and Paolo Frasconi. kfoil: Learning simple relational kernels. In *Proc. AAAI Vol. 6*, pages 389–394, 2006.

Daniel J Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 1977.

Xin Li and Dan Roth. Learning question classifiers. In *Proc. of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

Pierre Mahé and Jean-Philippe Vert. Graph kernels based on tree patterns for molecules. *Machine Learning*, 75(1):3–35, 2009. doi: 10.1007/s10994-008-5086-2. URL `http://dx.doi.org/10.1007/s10994-008-5086-2`.

Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert. Extensions of marginalized graph kernels. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, 2004. doi: 10.1145/1015330.1015446. URL `http://doi.acm.org/10.1145/1015330.1015446`.

Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi: 10.1016/j.jsc.2013.09.003. URL `http://dx.doi.org/10.1016/j.jsc.2013.09.003`.

Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in NIPS*, pages 3111–3119, 2013.

Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1352–1359, 2005. URL http://ijcai.org/Proceedings/05/Papers/1546.pdf.

M. Miller and J. Širán. Moore graphs and beyond: A survey of the degree/diameter problem. *Electronic Journal of Combinatorics*, 61:1–63, 2005.

Martin Mladenov, Babak Ahmadi, and Kristian Kersting. Lifted linear programming. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2012, La Palma, Canary Islands, April 21-23, 2012*, pages 788–797, 2012. URL http://jmlr.csail.mit.edu/proceedings/papers/v22/mladenov12.html.

Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Machine Learning*, 86(1):3–23, 2012. doi: 10.1007/s10994-011-5259-2. URL http://dx.doi.org/10.1007/s10994-011-5259-2.

Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015. ISSN 1573-0565. doi: 10.1007/s10994-014-5471-y. URL http://dx.doi.org/10.1007/s10994-014-5471-y.

Marion Neumann, Roman Garnett, Plinio Moreno, Novi Patricia, and Kristian Kersting. Propagation kernels for partially labeled graphs. In *ICML–2012 Workshop on Mining and Learning with Graphs (MLG–2012), Edinburgh, UK*, 2012a.

Marion Neumann, Novi Patricia, Roman Garnett, and Kristian Kersting. Efficient graph kernels by randomization. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I*, pages 378–393. 2012b. doi: 10.1007/978-3-642-33460-3_30. URL http://dx.doi.org/10.1007/978-3-642-33460-3_30.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 809–816, 2011.

M Niepert, M Ahmed, and K Kutzkov. Learning convolutional neural networks for graphs. *arXiv preprint arXiv:1605.05273*, 2016.

Francesco Orsini, Paolo Frasconi, and Luc De Raedt. Graph invariant kernels. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3756–3762, 2015a. URL http://ijcai.org/Abstract/15/528.

Francesco Orsini, Paolo Frasconi, and Luc De Raedt. kprolog: an algebraic prolog for kernel programming. In *25th International Conference on Inductive Logic Programming*. Springer, 2015b.

Francesco Orsini, Daniele Baracchi, and Paolo Frasconi. Shift aggregate extract networks. *arXiv preprint arXiv:1703.05537*, 2017.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 689–690. IEEE, 2011.

J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3): 239–266, 1990.

Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural Networks*, 18(8):1093–1110, 2005. 00251.

J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. In *First International Workshop on Mining Graphs, Trees and Sequences*, pages 65–74. Citeseer, 2003.

Matthew Richardson and Pedro M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006. doi: 10.1007/s10994-006-5833-1. URL http://dx.doi.org/10.1007/s10994-006-5833-1.

Kaspar Riesen and Horst Bunke. Reducing the dimensionality of dissimilarity space embedding graph kernels. *Engineering Applications of Artificial Intelligence*, 22(1): 48–56, 2009.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Frank Rosenblatt. Principles of neurodynamics. 1962.

Claude Sammut. The origins of inductive logic programming: A prehistoric tale. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 127–147. J. Stefan Institute, 1993.

Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 715–729, 1995.

Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1330–1339, 1997. URL http://ijcai.org/Proceedings/97-2/Papers/078.pdf.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *Neural Networks, IEEE Transactions on*, 20(1):61–80, 2009. 00073.

Bernhard Schoelkopf, Jason Weston, Eleazar Eskin, Christina Leslie, and William Stafford Noble. A kernel approach for learning from almost orthogonal patterns. In *European Conference on Machine Learning*, pages 511–528. Springer, 2002.

Bernhard Scholkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

Ida Schomburg, Antje Chang, Christian Ebeling, Marion Gremse, Christian Heldt, Gregor Huhn, and Dietmar Schomburg. Brenda, the enzyme database: updates and major new developments. *Nucleic Acids Research*, 32(Database-Issue):431–433, 2004. doi: 10.1093/nar/gkh081. URL http://dx.doi.org/10.1093/nar/gkh081.

Rahul Kumar Sevakula and Nishchal K. Verma. Support vector machine for large databases as classifier. In *Swarm, Evolutionary, and Memetic Computing - Third International Conference, SEMCCO 2012, Bhubaneswar, India, December 20-22, 2012. Proceedings*, pages 303–313. 2012. doi: 10.1007/978-3-642-35380-2_36. URL http://dx.doi.org/10.1007/978-3-642-35380-2_36.

Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814. ACM, 2007.

N Shervashidze, SVN Vishwanathan, T Petri, K Mehlhorn, and K M Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS-09*, volume 5, pages 488–495, 2009.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011. URL http://dl.acm.org/citation.cfm?id=2078187.

Masaaki Shimizu, Hiroshi Nagamochi, and Tatsuya Akutsu. Enumerating tree-like chemical graphs with given upper and lower bounds on path frequencies. *BMC bioinformatics*, 12(14):1, 2011.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

R Socher, C C Lin, C Manning, and A Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proc. of the ICML-11*, pages 129–136, 2011.

A Sperduti and A Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

Hannu Toivonen, Ashwin Srinivasan, Ross D King, Stefan Kramer, and Christoph Helma. Statistical evaluation of the predictive toxicology challenge 2000–2001. *Bioinformatics*, 19(10):1183–1193, 2003.

Shinji Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(5):695–703, 1988. doi: 10.1109/34.6778. URL http://dx.doi.org/10.1109/34.6778.

Wim Van Laer and Luc De Raedt. How to upgrade propositional learners to first order logic: A case study. In *Machine Learning and Its Applications*, pages 102–126. Springer, 2001.

Vladimir Vapnik. Pattern recognition using generalized portrait method. *Automation and remote control*, 24:774–780, 1963.

M. Verbeke, P. Frasconi, V. Van Asch, R. Morante, W. Daelemans, and L. De Raedt. Kernel-based logical and relational learning with klog for hedge cue detection. In *Inductive Logic Programming*, pages 347–357. Springer, 2012.

S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.

Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with tp-compilation. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1852–1858, 2015. URL http://ijcai.org/Abstract/15/263.

A Vullo and P Frasconi. Disulfide connectivity prediction using recursive neural networks and evolutionary information. *Bioinformatics*, 20(5):653–659, 2004.

Nikil Wale, Ian A Watson, and George Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14(3):347–375, 2008.

Boris Weisfeiler. *On Construction and Identification of Graphs*. Lecture Notes in Mathematics 558. Springer-Verlag Berlin Heidelberg, 1 edition, 1976. ISBN 978-3-540-08051-0,978-3-540-37539-5.

J Whaley, D Avots, M Carbin, and M S Lam. Using datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems*. Springer, 2005.

P Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proc. of KDD-15*, pages 1365–1374, 2015.

Dell Zhang and Wee Sun Lee. Question classification using support vector machines. In *SIGIR 2003: Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, July 28 - August 1, 2003, Toronto, Canada*, pages 26–32, 2003. doi: 10.1145/860435.860443. URL `http://doi.acm.org/10.1145/860435.860443`.

List of publications

- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. Graph Invariant Kernels. In: International Joint Conference on Artificial Intelligence. 2015. p. 3756-3762.
- Orsini, Francesco; Baracchi Daniele; Frasconi, Paolo. Shift Aggregate Extract Networks. Preprint arXiv:1703.05537 [cs.LG] 2017. [2]
- Orsini, Francesco; Frasconi, Paolo; De Raedt, Luc. kProbLog: An Algebraic Prolog for Kernel Programming. In: International Conference on Inductive Logic Programming. Springer International Publishing, 2015. p. 152-165. **Best Student Paper, Machine Learning Journal Award.** (An extended version of this work is currently under revision at the Machine Learning Journal)

[2] https://arxiv.org/pdf/1703.05537.pdf

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIEVE TALEN EN ARTIFICIELE INTELLIGENTIE
Celestijnenlaan 200A box 2402
B-3001 Leuven
francesco.orsini@kuleuven.be
https://dtai.cs.kuleuven.be/