KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

A Theory-based Study of Graph Mining

Mostafa Haghir Chehreghani

Supervisor:
Prof. Dr. ir. Maurice Bruynooghe
Dr. ir. Jan Ramon

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

February 2016

A Theory-based Study of Graph Mining

Mostafa Haghir Chehreghani

Examination committee:
Prof. Dr. ir. Jean Berlamont, chair
Prof. Dr. ir. Maurice Bruynooghe, supervisor
Dr. ir. Jan Ramon, supervisor
Prof. Dr. ir. Hendrik Blockeel
Prof. Dr. Bettina Berendt
Prof. Dr. Celine Robardet
(National Institute of Applied Science, Lyon, France)
Prof. Dr. Michalis Vazirgiannis
(LIX, Ecole Polytechnique, Palaiseau, France)

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

February 2016

© 2016 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Mostafa Haghir Chehreghani , Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Acknowledgement

First of all, I would like to thank my supervisors, Prof. Maurice Bruynooghe and Dr. Jan Ramon for their valuable ideas, guidance and support and for many long and detailed discussions on my work. Additionally, I want to thank Prof. Maurice Bruynooghe for reading my dissertation for several times and providing very useful ad detailed feedbacks for me.

I would like to thank the members of my jury: Prof. Bettina Berendt, Prof. Hendrik Blockeel, Prof. Celine Robardet and Prof. Michalis Vazirgiannis for their time and helpful suggestions on the manuscript. Furthermore, I thank Prof. Jean Berlamont for chairing the jury.

I gratefully acknowledge the funding sources that made my PhD work possible. I was funded by the ERC Starting Grant 240186 "MiGraNT: Mining Graphs and Networks: a Theory-based approach".

I had the chance to benefit from the company of many colleagues and friends over the years as a PhD student. My gratitude goes towards the entire ML research group of KU Leuven for creating an enjoyable environment. I am especially thankful to my friends in the MiGraNT team. I am grateful to Dr. Leander Schietgat for helping me in translating the abstract from English to Dutch.

Finally, I would like to thank my family for their love, support and encouragement. I sincerely thank my parents who raised me with love of science and supported me in all my pursuits, and my brothers and my sister for their encouragement and for concern through my PhD years.

Abstract

During the last decade or so, the amount of data that is generated and becomes publicly available is rapidly growing. This makes it impossible to extract useful information from this huge amount of data manually without using automatic tools and algorithms. Data mining has been defined as the process of automatic extraction of useful and previously unknown information from a large dataset using techniques from statistics, artificial intelligence and database management. Furthermore, in many applications, such as Bioinformatics, the world wide web, social and technological and communication networks, data are usually represented with graphs. This makes graph mining practically interesting, while also a challenging research area, due to high computational cost involved in processing graph data. In this dissertation, we investigate two key problems in graph mining: *frequent pattern mining* and *betweenness centrality computation*.

Existing algorithms for finding frequent patterns from large single networks mainly use subgraph isomorphism. However, subgraph isomorphism is expensive to compute: deciding whether one graph is subgraph isomorphic to another graph is NP-complete in terms of the sizes of the graphs. Recently, a few algorithms have used subgraph homomorphism. However, they find very restricted classes of patterns such as trees. The main challenge with pattern mining under subgraph homomorphism is the pattern generation phase. In this work, we go beyond trees and propose an efficient algorithm for mining graph patterns from large networks under homomorphism. We introduce a new class of patterns, called *rooted graphs*, and present an algorithm for complete generation of rooted graphs. We also propose a new data structure for compact representation of all frequent patterns. By performing extensive experiments on several real-world and synthetic large networks, we show the empirical efficiency of our proposed algorithm, called HoPa.

We then present an efficient algorithm for subtree homeomorphism with application to frequent pattern mining. We propose a compact data-structure, called **occ**, that can encode and represent several occurrences of a tree pattern and define efficient join operations on the **occ** data-structure, that help us to count occurrences of tree patterns

according to occurrences of their proper subtrees. Based on the proposed subtree homeomorphism method, we develop an effective pattern mining algorithm, called TPMiner. We evaluate the efficiency of TPMiner on several real-world and synthetic datasets. Our extensive experiments confirm that TPMiner always outperforms well-known existing algorithms, and in several cases the improvement with respect to existing algorithms is significant.

Finally, we propose a randomized algorithm for unbiased estimation of betweenness centrality. We discuss the conditions that a promising sampling technique should satisfy to minimize the approximation error. We then propose a sampling method that fits better with these conditions. By performing extensive experiments on synthetic and real-world networks, we compare our proposed method with existing algorithms and show that our method works with a better accuracy.

Beknopte samenvatting

Tijdens de laatste 10 jaar is de hoeveelheid gegevens die gegenereerd en publiek beschikbaar gemaakt wordt snel aan het groeien. Dat maakt het onmogelijk om nuttige informatie uit deze grote hoeveelheden gegevens te halen zonder gebruik te maken van automatische tools en algoritmes. *Data mining* is het proces dat nuttige en voorheen onbekende informatie op een automatische manier afleidt uit een grote databank, gebruik makende van technieken uit de statistiek, de kunstmatige intelligente en databankenbeheer. Bovendien worden in vele toepassingen zoals bioinformatica, het wereldwijde web en sociale, technologische en communicatienetwerken de gegevens meestal voorgesteld door grafen. Dit maakt *graph mining* praktisch nuttig, terwijl het door de hoge computationele kosten die gepaard gaan met het verwerken van grafen ook een uitdagend onderzoeksdomein vormt. In deze dissertatie onderzoeken we twee belangrijke problemen uit graph mining: het zoeken naar frequente patronen en de berekening van betweenness-centraliteit.

Bestaande algoritmes voor het vinden van frequente patronen in grote netwerken gebruiken vooral subgraaf isomorfisme. Subgraaf isomorfisme is echter heel duur om te berekenen: beslissen of een graaf subgraaf isomorfisch is t.o.v. een andere graaf is NP-compleet in functie van de groottes van de grafen. Onlangs hebben enkele algoritmes het subgraaf homomorfisme toegepast. Ze beperkten zich echter tot heel gelimiteerde klassen van patronen zoals bomen. De grote uitdaging van het zoeken naar patronen m.b.v. subgraaf homomorfisme is het genereren van de patronen. In dit werk gaan we verder dan bomen alleen en stellen we een efficiënt algoritme voor dat graafpatronen kan afleiden uit grote netwerken m.b.v. homomorfisme. We introduceren een nieuwe klasse van patronen, *rooted graphs* genaamd, en presenteren een algoritme voor de volledige generatie van dergelijke patronen. We stellen ook een nieuwe datastructuur voor die alle frequente patronen compact kan representeren. Aan de hand van een uitgebreide experimentele analyse op meerdere bestaande en artificiële grote netwerken tonen we de efficiëntie van ons algoritme HoPa aan.

Vervolgens stellen we een efficiënt algoritme voor deelboom homeomorfisme voor als toepassing op het zoeken naar frequente patronen. We introduceren een compacte

datastructuur, genaamd **occ**, die verschillende voorkomens van een boompatroon kan encoderen en representeren, en we definiëren ook efficiënte join operaties voor de occ datastructuur, die ons helpen om voorkomens van boompatronen te tellen op basis van de voorkomens in hun deelbomen. We hebben een algoritme ontwikkeld voor het zoeken naar patronen op basis van dit deelboom homeomorfisme, dat we TPMiner noemen. We evalueren de efficiëntie van TPMiner op meerdere bestaande en synthetische datasets. Onze uitvoerige experimenten bevestigen dat TPMiner beter presteert dan bekende algoritmes, en in meerdere gevallen is de verbetering significant.

Tenslotte stellen we een gerandomiseerd algoritme voor onbevooroordeelde benadering van betweenness-centraliteit. We bespreken de voorwaarden aan dewelke een *sampling* techniek moet voldoen om de benaderingsfout zo klein mogelijk te houden. Dan stellen we een sampling techniek voor die beter aan deze voorwaarden voldoet. Aan de hand van uitvoerige experimenten op netwerken vergelijken we onze voorgestelde methode met bestaande algoritmes en tonen we aan dat onze methode een betere accuraatheid heeft.

Contents

Acknowledgement i

Abstract iii

Contents vii

List of Figures xi

List of Tables xvii

1 Introduction 3

1.1 High level description of the studied problems 4

1.2 Contributions 7

1.3 Outline of the dissertation 8

1.4 Connections between different chapters 9

2 Background 11

2.1 Graph theory 11

2.2 Poset and closure operator 15

2.3 Data mining 16

2.4 Betweenness centrality 17

3 Mining Large Single Networks under Subgraph Homomorphism 19

3.1 Introduction . . . 19
3.2 Related work . . . 22
3.2.1 Efficient pattern matching algorithms . . . 23
3.2.2 Pattern mining strategies . . . 24
3.2.3 Single network mining . . . 24
3.2.4 Transactional graph mining . . . 26
3.3 Rooted patterns and their generation . . . 26
3.3.1 Rooted patterns . . . 26
3.3.2 Generating BRETDs . . . 33
3.4 Mining frequent rooted patterns . . . 39
3.5 Condensed representations of frequent patterns . . . 44
3.6 Experimental results . . . 51
3.6.1 Experimental setup . . . 51
3.6.2 Datasets . . . 52
3.6.3 Results . . . 53
Mining frequent rooted trees. . . . 53
Mining frequent rooted graphs. . . . 55
3.6.4 Discussion . . . 61
3.7 Conclusion . . . 61

4 Mining Rooted Ordered Trees under Subtree Homeomorphism 63

4.1 Introduction . . . 63
4.2 Problem statement . . . 67
4.3 Related work . . . 68
4.4 Efficient tree mining under subtree homeomorphism . . . 70
4.4.1 Occurrence trees and their extensions . . . 70

4.4.2 **Occ-list**: an efficient data structure for tree mining under subtree homeomorphism 74

4.4.3 Operations on the **occ-list** data structure 75

4.4.4 Complexity analysis 79

4.4.5 A brief comparison with other vertical frequency counting approaches 82

4.5 TPMiner: an efficient algorithm for finding frequent embedded tree patterns 82

4.5.1 The TPMiner algorithm 82

4.5.2 An optimization technique for candidate generation 84

4.6 Experimental Results 84

4.7 Conclusion 89

5 An Efficient Algorithm for Approximate Betweenness Centrality Computation 91

5.1 Introduction 91

5.2 Related work 94

5.3 Approximate betweenness centrality computation 96

5.4 Sampling methods 98

5.4.1 Optimal sampling 98

5.4.2 A property of promising sampling methods 100

5.4.3 A new sampling technique 100

5.5 Experimental results 103

5.5.1 Datasets 103

5.5.2 Empirical results 104

5.6 Conclusion 109

6 Conclusion and Future Work 113

6.1 Summary of main contributions 113

6.2 Future work 115

Bibliography **117**

Selected Papers **127**

Curriculum **129**

List of Figures

1.1 A graph formed by Wikipedia language versions as vertices and Wikipedia editors as edges (Hale, 2014). 4

1.2 Different matching operators between a pattern and a database graph. 5

1.3 In the graph of this figure, vertices v_2 and v_4 have higher betweenness scores than vertices v_1, v_3 and v_5 and hence, they have more control over communications/information flow in the network. 6

2.1 G is a graph and $T1$, $T2$ and $T3$ are three tree decompositions of it. For example, T_1 has three nodes z_1, z_2 and z_3, where $B(z_1) = \{v_2, v_4\}$, $B(z_2) = \{v_1, v_3\}$, and $B(z_3) = \{v_3, v_4, v_5\}$. $T1$, $T2$ and $T3$ have *width* 2, 2 and 4, respectively. Observe that no tree decomposition with width 1 is possible for the subgraph with vertices v_3, v_4 and v_5, hence the treewidth is 2. 14

2.2 All minimal extensions of a rooted ordered tree, a rooted unordered tree, and a free tree. 16

3.1 A database graph (left) and a pattern that is expressed using homomorphism but not using isomorphism (right). 20

3.2 There are an infinite number of rooted patterns ${P_1}^{X_1}$, ${P_2}^{X_2}$, ..., ${P_n}^{X_n}$, ... that are subgraph homomorphic to the network H and to each other. The tree decomposition above each rooted pattern shows its BRETD. 29

3.3 Figure (a) shows a rooted pattern P^X, (b) a BRETD of it and (c) $Core(P^X)$. 32

3.4 The possible BRETDs for the rooted pattern P^X are $T1$, $T2$ and $T3$. 32

3.5 An example of the extension operator. On the left a pattern (bottom) and a tree decomposition of it (top); on the right an extension of the tree decomposition (top) and the pattern it represents (bottom). 34

3.6 An example of joining two BRETDs T_{P^X} and T_{Q^Y}, that have respectively the underlying rooted patterns P^X and Q^Y. The isomorphism mapping between $R(P^X)$ and $R(Q^Y)$ is depicted by arrows. It maps v_4 to v_7, v_2 to v_8, and v_3 to v_6. In the generated rooted pattern S^Z, v_3 is replaced by v_6 and v_2 is replaced by v_8. . . . 37

3.7 An example of root embedding. The graph on the left is a database graph. The graph on the right is a rooted pattern that has three embeddings in the database graph. The first and second embeddings have the same root embeddings, that is $\{u_0, u_1, u_2\}$. The third one has a different root embedding $\{u_0, u_2, u_1\}$. Therefore, the frequency of the rooted pattern is 2. 41

3.8 Let H be a database graph. $P2$ and $P3$ are the maximal super patterns of $P1$ that have the same frequency as $P1$. As $P1$ has no unique maximal supper pattern, choosing an arbitrary one as its closure violates condition C2 for the other one. As a result, the frequency based closedness data structure is not a closure operator. 45

3.9 Rooted patterns P^X and Q^Y are in the same root embedding equivalence class. H is the database graph and N^Z is generated by a merge of P^X and Q^Y. 45

3.10 Experimental results for mining frequent rooted trees from facebook-mhrw for different values of $minsup$; $maxLevel$ is 3. 54

3.11 Experimental results for mining frequent rooted trees from facebook-mhrw for different values of $maxLevel$; $minsup$ is 64,000. 54

3.12 Experimental results for mining frequent rooted trees from facebook-uniform for different values of $minsup$; $maxLevel$ is 3. 55

3.13 Experimental results for mining frequent rooted trees from facebook-uniform for different values of $maxLevel$; $minsup$ is 4,000. 55

3.14 Experimental results for mining frequent rooted trees from IMDB for different values of $minsup$; $maxLevel$ is 3. 56

3.15 Experimental results for mining frequent rooted trees from IMDB for different values of $maxLevel$; $minsup$ is $150,000$. 56

3.16 Experimental results for mining frequent rooted trees from $BA10^6$ for different values of $minsup$; $maxLevel$ is 3. 57

3.17 Experimental results for mining frequent rooted trees from $BA10^6$ for different values of $maxLevel$; $minsup$ is $30,000$. 57

3.18 Experimental results for mining frequent rooted trees from $BA10^7$ for different values of $minsup$; $maxLevel$ is 3. 58

3.19 Experimental results for mining frequent rooted trees from $BA10^7$ for different values of $maxLevel$; $minsup$ is $350,000$. 58

3.20 Experimental results over the facebook-mhrw dataset for $treewidth = 1, 2, 3$; $minsup$ is $1,500$ and $maxLevel$ is 4. 58

3.21 Experimental results over the facebook-uniform dataset for $treewidth = 1, 2, 3$; $minsup$ is 10 and $maxLevel$ is 4. 59

3.22 Experimental results over the $BA10^6$ dataset for $treewidth = 1, 2$; $minsup$ is $30,000$ and $maxLevel$ is 3. 59

4.1 Under subtree homeomorphism, the tree T can be classified as $C1$ while it does not match the model under subtree isomorphism and subtree homomorphism. 64

4.2 In the database tree T and for $minsup = 2$, while $P1$ is infrequent, it has two frequent supertrees $P2$ and $P3$. 67

4.3 From left to right, a database tree T, a pattern P and three occurrence trees $\mathcal{OT}(\varphi_1)$, $\mathcal{OT}(\varphi_2)$, and $\mathcal{OT}(\varphi_3)$. Labels are inside vertices, preorder numbers are next to vertices. The occurrence trees are represented by showing the occurrences of the pattern vertices in bold. Their edges are the images of the edges in the pattern; for example, $\mathcal{OT}(\varphi_3)$ refers to the tree formed by vertices 0, 4 and 5, with 0 as the root, and with $(0, 4)$ and $(0, 5)$ as edges. 71

4.4 A database tree is shown in 4.4a, together with four rightmost path extensions of different occurrence trees in that database tree. However, only 4.4b is itself an occurrence tree in the database tree; 4.4c, 4.4d and 4.4e violate conditions (ii), (iii) and (i) of Proposition 9, respectively. 71

4.5 An example of **occ-list**. $T0$ and $T1$ are two database trees and minimum-support is equal to 2. The figure presents the **occ-list**s of some frequent 1-tree patterns, frequent 2-tree patterns, frequent 3-tree patterns and frequent 4-tree patterns. 76

4.6 Details of the relationship between **occ-list**(P'), **occ-list**(v) and **occ-list**(P) for the database trees of Figure 4.5. The entries oc'_1, ov_3 and oc_1 satisfy the properties of Proposition 12. Also the tuples (oc'_2, ov_4, oc_2) and (oc'_2, ov_5, oc_3) satisfy the properties. The proposition is exploited in Algorithm 6 below. Its $leaf_join$ operation uses **occ-list**(P') and **occ-list**(v) to compute **occ-list**(P). . . 77

4.7 Details of the relationship between **occ-list**(P'), **occ-list**(v) and **occ-list**(P) for the database trees of Figure 4.5. The entries $\{oc'_1, oc'_2\}$, ov_1 and oc satisfy the properties of Proposition 13. As $c = 0$, rightmost paths of the occurrence trees represented by oc'_1 and oc'_2 share the first vertex, that is vertex 0 of $T0$; and rightmost paths of the occurrence trees represented by oc share the first and second vertices, that are vertices 0 and 3 of $T0$. The proposition is exploited in Algorithm 6 below. Its $inner_join$ operation uses **occ-list**(P') and **occ-list**(v) to compute **occ-list**(P). 79

4.8 Figure (a) shows a tree pattern P and (b) a database tree T. The number of occurrences of P in T is exponential in terms of n and k. In this case, the size of **occ-list** is linear, however, the size of the data-structures generated by the other algorithms is exponential. . . . 80

4.9 Figure (a) shows a tree pattern P and (b) a database tree T. The number of occurrences of P in T is exponential in terms of n and k. In this case, the size of the **occ-list** and the size of the data-structures generated by the other algorithms for P are exponential. 80

4.10 Comparison over CSLOGS32241. 85

4.11 Comparison over Prions. 86

4.12 Comparison over NASA. 86

4.13 Comparison over synthetic datasets. 87

5.1 Figure (a) shows a toy road network where betweenness score of vertex 1 is 6 and it is desired to reduce this score. Three new configurations are suggested, that are depicted in Figures (b), (c) and (d). Betweenness scores of vertex 1 in the networks of Figures (b), (c), and (d) are 3, 3 and 2, respectively. 93

5.2 A graph (left) and its SPD rooted at vertex 1 (right). 102

5.3 A comparison between approximation errors of uniform sampling and distance-based sampling for 57 different vertices in the BA10^3 dataset. 107

5.4 A comparison between approximation errors of uniform sampling and distance-based sampling for 155 different vertices in the BA10^4 dataset. 107

5.5 A comparison between approximation errors of uniform sampling and distance-based sampling for 89 different vertices in the Wiki-Vote dataset. 108

5.6 A comparison between approximation errors of uniform sampling and distance-based sampling for 118 different vertices in the Email-Enron dataset. 108

5.7 A comparison between approximation errors of uniform sampling and distance-based sampling for 28 different vertices in the dblp0305 dataset. 109

5.8 A comparison between approximation errors of uniform sampling and distance-based sampling for 9 different vertices in the dblp0507 dataset. 110

5.9 A comparison between approximation errors of uniform sampling and distance-based sampling for different vertices in the CA-CondMat dataset. 110

5.10 A comparison between approximation errors of uniform sampling and distance-based sampling for different vertices in the CA-HepTh dataset. 110

5.11 In sparse graphs, distance-based sampling is closer to optimal sampling. The graph in the left side shows an SPD in a dense graph, and the graph in the right side shows an SPD in a sparse graph. 111

List of Tables

3.1 Summary of real-world networks. 53

3.2 Patterns statistics over facebook-mhrw ($minsup = 1,500, maxLevel = 4$ and $treewidth = 3$). 60

3.3 Patterns statistics over facebook-uniform ($minsup = 10, maxLevel = 4$ and $treewidth = 3$). 60

4.1 Summary of real-world datasets. 85

5.1 Summary of real-world networks. 104

5.2 Comparing average approximation error and average running time of uniform sampling, distance-based sampling, and the Brandes exact method, for various single vertices in different datasets. 106

List of symbols

$\cong^i$	Graph (tree) isomorphism
$\preceq^i$	Subgraph (subtree) isomorphism
$\preceq^h$	Subgraph (subtree) homeomorphism
$\cong^{hm}$	Graph (tree) homomorphism
$\preceq^{hm}$	Subgraph (subtree) homomorphism
P^X	A rooted graph (a graph P rooted at vertices $X \subseteq V(P)$)
ρ_{ext}	The extension operator
ρ_{join}	The join operator
ρ_{mrg}	The merge operator
$RE(P^X, H)$	The set of root embeddings of a rooted pattern P^X in a database graph H
$\equiv_{RE}$	Root embedding equivalence class
σ_{RE}	Our proposed closure operator for single network mining under homomorphism
$RExtend(T, v, u)$	Rightmost path extension of tree T with the new vertex v attached at the existing vertex u
$\mathcal{OT}$	Occurrence tree
$rdep_T(x)$	The length of the path from the root of tree T to vertex x
$x.scope$	The scope of vertex x in a database tree
Occ-list(P)	**Occ-list** of pattern P, used to efficiently count frequency of P under subtree homomorphism
$BC(v)$	Betweenness centrality of vertex v
σ_{st}	The number of shortest paths between vertices s and t
$\sigma_{st}(v)$	The number of shortest paths between vertices s and t that also pass through v
$\delta_{s\bullet}(v)$	The dependency score of vertex s on vertex v

Chapter 1

Introduction

During the last years, the amount of data that is generated and becomes publicly available is rapidly increasing. This makes it impossible to extract useful and interesting information from this data manually by hand and without using automatic tools and algorithms. To satisfy this requirement, the field of data mining emerged that deals with automatic extraction of useful and previously unknown information from a large dataset using techniques from statistics, artificial intelligence and database management (Han and Kamber, 2000). Furthermore, in many applications, such as Bioinformatics, the world wide web, social and technological and communication networks, data are usually represented with graphs. Informally speaking, a graph is a set of vertices (nodes) joined by a set of edges. Examples of using graphs for modeling data include:

- A road network, where vertices are intersections of roads and (undirected) edges are roads connecting these intersections.
- The link structure of a website, where vertices represent web pages and (directed) edges represent links from one page to another.
- A social network, such as facebook, where vertices are individuals and (undirected) edges represent friendship relations between them.

As an example of representing a (small and real-world) network with a graph, consider Figure 1.1, where vertices of the graph represent different language versions of Wikipedia during July 2013 and a directed edge between two vertices (versions) shows the existence of at least one user primarily editing one language who has edited another language as well (Hale, 2014).

Figure 1.1: A graph formed by Wikipedia language versions as vertices and Wikipedia editors as edges (Hale, 2014).

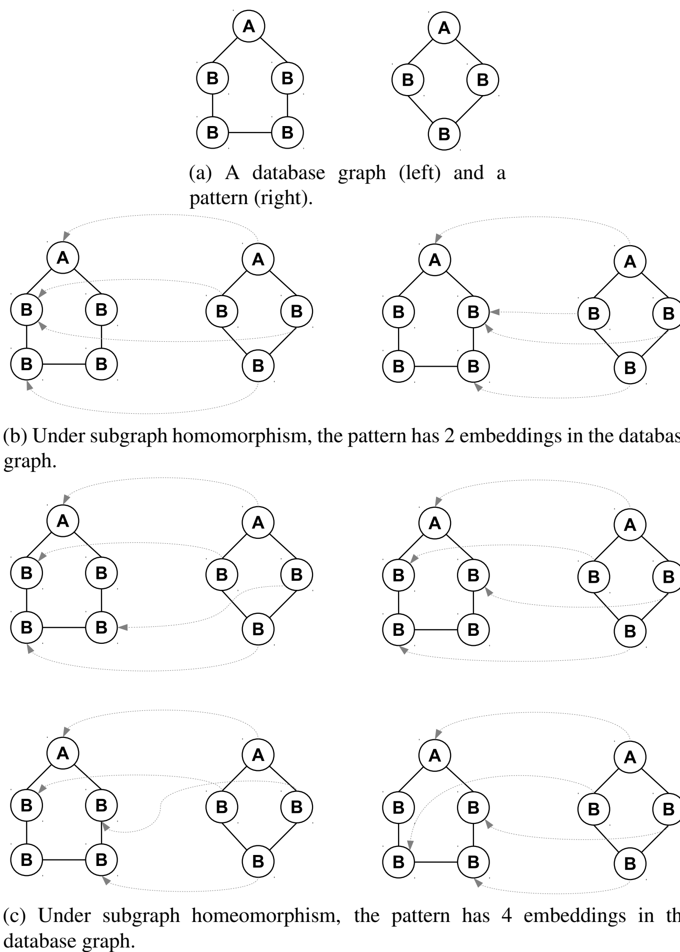
1.1 High level description of the studied problems

The huge amount of available graph data makes graph mining an interesting and important research challenge. The concern of this dissertation is graph mining and in particular, the following problems: *frequent pattern mining* and *betweenness centrality computation*.

Frequent pattern mining A key task in graph mining is frequent pattern mining. A graph (pattern) is frequent in a database DB iff its *frequency* in DB, under a matching operator, is more than a user-defined threshold *minimum support*. The *frequency* of a pattern in a database is defined as e.g., the number of occurrences/embeddings/instances of the pattern in the database. The *matching operator* defines how the pattern is matched onto a graph in the database, called a *database graph*. There are three widely-used matching operators: *subgraph isomorphism*, *subgraph homomorphism* and *subgraph homeomorphism*. *Subgraph isomorphism* is an injective mapping of the vertices such that an edge in the pattern is mapped onto an edge in the database graph. *Subgraph homeomorphism* is also an injective mapping, however, an edge in the pattern is mapped onto a path in the database graph. Finally, subgraph homomorphism is a mapping where an edge in the pattern is mapped onto an edge in the database graph. In this mapping, two vertices in the pattern might be mapped onto the same vertex in the database graph.

For example, in Figure 1.2, 1.2a presents a database graph (left) and a pattern (right).

Under subgraph isomorphism the pattern has no embedding in the database graph, however, under subgraph homomorphism and subgraph homeomorphism, it has 2 and 4 embeddings, respectively. If the user-defined threshold *minimum support* is 3, the pattern will not be frequent under subgraph isomorphism and subgraph homomorphism, however, it will be frequent under subgraph homeomorphism.



(a) A database graph (left) and a pattern (right).

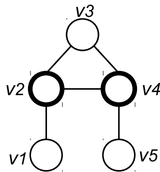
(b) Under subgraph homomorphism, the pattern has 2 embeddings in the database graph.

(c) Under subgraph homeomorphism, the pattern has 4 embeddings in the database graph.

Figure 1.2: Different matching operators between a pattern and a database graph.

In the frequent pattern mining problem, given a database of graphs and an integer minimum support, the goal is to find all frequent patterns in the database.

Figure 1.3: In the graph of this figure, vertices v_2 and v_4 have higher betweenness scores than vertices v_1, v_3 and v_5 and hence, they have more control over communications/information flow in the network.



Betweenness centrality computation. An essential index in determining the centrality or importance of a vertex in a graph is *betweenness centrality*. Betweenness score of a vertex v is defined as the ratio (or the number) of shortest paths from all vertices to all others in the graph that pass through v. With the assumption that in a graph (network) communications are done through shortest paths, betweenness of a vertex indicates its influence on the communications/information flow/transfers. For example, in Figure 1.3:

- There is no shortest path passing through vertex v_1.
- The following shortest paths pass through vertex v_2:
  - $v_1 - v_2 - v_3$,
  - $v_1 - v_2 - v_4$, and
  - $v_1 - v_2 - v_4 - v_5$.
- There is no shortest path passing through vertex v_3.
- The following shortest paths pass through vertex v_4:
  - $v_2 - v_4 - v_5$,
  - $v_3 - v_4 - v_5$, and
  - $v_1 - v_2 - v_4 - v_5$.
- There is no shortest path that passes through vertex v_5.

Therefore, vertices v_2 and v_4 have more influence than vertices v_1, v_3 and v_5 on the information flow in the graph and hence, they are more important.

In the betweenness centrality computation problem, given a graph (and one or more vertices of it), the goal is to compute betweenness centrality of the (given) vertices of the graph.

1.2 Contributions

In this dissertation, we have made a number of novel and important contributions to the field of graph mining. A summary of our key contributions is listed below.

Single network mining under subgraph homomorphism. Existing algorithms for finding frequent patterns from large single networks mainly use subgraph isomorphism. However, subgraph isomorphism is expensive to compute: deciding whether a graph G_1 is subgraph isomorphic to another graph G_2 is NP-complete in terms of $|V(G_1)|$ and $|V(G_2)|$, even if G_1 is a simple graph such as a path. Recently, a few algorithms have used subgraph homomorphism. However, they find very restricted classes of patterns such as trees. The main challenge with pattern mining under subgraph homomorphism is the pattern generation phase; in particular, a larger graph might be subgraph homomorphic to a smaller graph. In this work, we go beyond tree patterns and address the aforementioned problems for graph patterns. We introduce a new class of patterns, called rooted patterns, and present an algorithm for complete generation of rooted patterns. We also propose a new notion of closedness for compact representation of all frequent patterns and investigate its properties. We then present HoPa, an efficient algorithm for finding frequent rooted patterns from large single networks under subgraph homomorphism. HoPa uses the querying system developed by Fannes et al. (2015) for frequency counting. Finally, by performing extensive experiments over several real-world and synthetic large networks, we show the empirical efficiency of HoPa. This chapter is recent unpublished work in close collaboration with my supervisors. A very preliminary version has been presented at

- M. H. Chehreghani, J. Ramon and T. Fannes: *Mining large networks under homomorphism*, In Dutch-Belgian Database Day (DBDBD), Rotterdam, the Netherlands, 29 November 2013.

Mining rooted ordered trees under subtree homeomorphism. The crucial step in frequent tree pattern mining is frequency counting, which involves a matching operator to find occurrences (instances) of a tree pattern in a given collection of trees. A widely used matching operator for tree-structured data is *subtree homeomorphism*, where an edge in the tree pattern is mapped onto an ancestor-descendant relationship in the given tree. Tree patterns that are frequent under subtree homeomorphism are usually called *embedded patterns*. In this work, we present an efficient algorithm for subtree homeomorphism with application to frequent pattern mining. We propose a compact data-structure, called **occ**, which can encode and represent several occurrences of a tree pattern. We then define efficient join operations on the **occ** data-structure, which help us to count occurrences of tree patterns according to

occurrences of their proper subtrees. Based on the proposed subtree homeomorphism method, we develop an effective pattern mining algorithm, called TPMiner. We evaluate the efficiency of TPMiner on several real-world and synthetic datasets. Our extensive experiments confirm that TPMiner always outperforms well-known existing algorithms, and in several cases the improvement with respect to existing algorithms is significant. This chapter was previously published as:

- M. H. Chehreghani and M. Bruynooghe: *Mining rooted ordered trees under subtree homeomorphism*, Data Mining and Knowledge Discovery journal (DMKD), in press, DOI: 10.1007/s10618-015-0439-5. This paper will also be presented in the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECML-PKDD), Riva del Garda, Italy, September 19-23, 2016.

Approximate betweenness centrality computation. In this work, we propose a randomized algorithm for unbiased estimation of betweenness scores of vertices of a graph. The proposed framework can be adapted with various sampling techniques and give algorithms with different characteristics. We discuss the conditions that a promising sampling technique should satisfy to minimize the approximation error and propose a sampling method that fits better with the conditions. We perform extensive experiments on synthetic networks as well as networks from real-world and show that compared to existing inexact and exact algorithms, our method works with higher accuracy or gives significant speedups. This chapter was published in the following papers:

- M. H. Chehreghani: *An efficient algorithm for approximate betweenness centrality computation*, Computer Journal (Comp. J.), Oxford University press, 57(9), 1371-1382, 2014.
- M. H. Chehreghani: *An efficient algorithm for approximate betweenness centrality computation*, In 22nd ACM International Conference on Information and Knowledge Management (CIKM), 2013.

1.3 Outline of the dissertation

The rest of this dissertation consists of 5 chapters. Chapter 2 presents basic definitions, Chapters 3-5 describe the key contributions of the dissertation, and Chapter 6 presents conclusion and future work.

- In Chapter 2, we review the necessary background on graph theory and data mining.

- In Chapter 3, we introduce rooted patterns, a new notion of closedness for compact representation of frequent rooted patterns, and the HoPa algorithm for finding frequent rooted patterns from large single networks under subgraph homomorphism.
- In Chapter 4, we introduce a new algorithm for subtree homeomorphism with application to frequent pattern discovery, and present the TPMiner algorithm for finding frequent patterns from rooted ordered trees and show its high efficiency compared to existing algorithms.
- In Chapter 5, we propose a randomized algorithm for unbiased estimation of betweenness scores and discuss the conditions that a promising sampling technique should satisfy to minimize the approximation error and propose a sampling method that fits better with the conditions.
- Finally in Chapter 6, we summarize our contributions and present our conclusions and suggestions for future work.

1.4 Connections between different chapters

Two important types of statistics frequently computed for graphs and networks are: *i*) statistics related to the patterns and motifs that frequently occur in the graph, and *ii*) statistics that measure the (relative) importance of a vertex or a set of vertices in the graph. In this PhD dissertation, we try to provide a coverage of both types. For the first type, we investigate two different classes of graphs and for each one, we investigate frequency counting under a proper matching operator (Chapters 3 and 4). For the second type, we study centrality of a vertex under the widely used notion of betweenness centrality (Chapter 5).

The interesting research question arising in the end of this dissertation is whether there is any algorithmic connection between these two types of statistics. In other words, can having statistics of one type improve the efficiency of computing statistics of the other type? In the end of this dissertation, we present this as a conjecture and open problem for future research.

Chapter 2

Background

In this chapter, we introduce basic definitions and concepts widely used in the dissertation. First in Section 2.1, we review the necessary background in graph theory. Then in Section 2.2, we present a brief description of poset and closure operator and in Section 2.3, fundamentals and basic definitions from data mining. Finally in Section 2.4, we present the necessary background in betweenness centrality computation.

2.1 Graph theory

We assume the reader is familiar with the basic concepts in graph theory. The interested reader can refer to e.g., Diestel (2010). Let G be a graph. We refer to its vertices and edges by $V(G)$ and $E(G)$, respectively. We assume graphs are connected, simple (i.e., there is at most one edge between two vertices and there are no reflexive edges) and finite and have labels on both edges and vertices that are selected from a vertex alphabet Σ_V and an edge alphabet Σ_E, respectively. This is without loss of generality as we can assume all vertices/edges have the same label in case of an unlabeled graph. Unless explicitly mentioned, we suppose graphs are undirected. For an $x \in V(G) \cup E(G), \lambda_G(x)$ gives the label of x in G. If G is clear from the context, we drop it and write $\lambda(x)$. Two graphs $G_1 = (V_1, E_1, \lambda_1)$ and $G_2 = (V_2, E_2, \lambda_2)$ (either both are directed or both are undirected) are identical, written as $G_1 = G_2$, if $V_1 = V_2$, $E_1 = E_2$, and $\forall x \in V_1 \cup E_1 : \lambda_1(x) = \lambda_2(x)$. G_1 is a subgraph of G_2, denoted by $G_1 \preceq G_2$, iff $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, and $\forall x \in V_1 \cup E_1 : \lambda_1(x) = \lambda_2(x)$. For a set $S \subseteq V(G)$, the graph induced by S, denoted by $G[S]$, is the subgraph G' of G, where $V(G') = S$ and $E(G') = \{\{u, v\} \in E(G) \mid u \in S \wedge v \in S\}$. The *size* of G is defined as the number of vertices of G.

A *path* from a vertex v_0 to a vertex v_n in a graph $G = (V, E, \lambda)$ is a sequence of vertices such that $\forall i$, $0 \leq i \leq n-1$, $\{v_i, v_{i+1}\} \in E(G)$. The *length* of a path is defined as its number of edges (number of vertices minus 1). A *cycle* is a path with $v_0 = v_n$.

Two graphs G_1 and G_2 are *isomorphic*, denoted by $G_1 \cong^i G_2$, if there is a bijection $\varphi : V(G_1) \rightarrow V(G_2)$ satisfying: i) $\{u, v\} \in E(G_1)$ if and only if $\{\varphi(u), \varphi(v)\} \in E(G_2)$ for every $u, v \in V(G_1)$, ii) $\lambda_{G_1}(u) = \lambda_{G_2}(\varphi(u))$ for every $u \in V(G_1)$, and iii) $\lambda_{G_1}(\{u, v\}) = \lambda_{G_2}(\{\varphi(u), \varphi(v)\})$ for every $\{u, v\} \in E(G_1)$. The mapping φ is called a graph isomorphism from G_1 to G_2. We say G_1 is subgraph isomorphic to G_2, denoted by $G_1 \preceq^i G_2$, if G_1 is isomorphic to a subgraph of G_2. A *graph automorphism* of a graph G is a graph isomorphism from G to itself.

A *graph homomorphism* from a graph G_1 to a graph G_2, is a surjective mapping $\varphi : V(G_1) \rightarrow V(G_2)$ satisfying: i) if $\{u, v\} \in E(G_1)$, then $\{\varphi(u), \varphi(v)\} \in E(G_2)$ for every $u, v \in V(G_1)$, ii) $\lambda_{G_1}(u) = \lambda_{G_2}(\varphi(u))$ for every $u \in V(G_1)$, and iii) $\lambda_{G_1}(\{u, v\}) = \lambda_{G_2}(\{\varphi(u), \varphi(v)\})$ for every $\{u, v\} \in E(G_1)$. G_1 is *homomorphic* to G_2, denoted by $G_1 \cong^{hm} G_2$, iff there exists a graph homomorphism from G_1 to G_2. G_1 is *subgraph homomorphic* to G_2, denoted by $G_1 \preceq^{hm} G_2$, iff there exists a graph homomorphism from G_1 to a subgraph of G_2.

A *subgraph homeomorphism* from a graph G_1 to a graph G_2 is a pair of injective mappings φ_1 and φ_2, the first one from vertices of G_1 to vertices of G_2 and the second one from edges of G_1 to simple paths of G_2, that satisfy: i) for every $u, v \in V(G_1)$, if $\{u, v\} \in E(G_1)$, then there exists a simple path P between $\varphi_1(u)$ and $\varphi_1(v)$ in G_2 such that $\varphi_2(\{u, v\}) = P$, and ii) $\lambda_{G_1}(u) = \lambda_{G_2}(\varphi_1(u))$, for every $u \in V(G_1)$. If the mappings of the edges of G_1 are edge-disjoint paths of G_2, the subgraph homeomorphism is called an *edge-disjoint subgraph homeomorphism* (LaPaugh and Rivest, 1980).

Trees An undirected graph not containing any cycles is called a *forest* and a connected forest is called a (*free*) *tree*. A *rooted tree* is a directed acyclic graph (DAG) in which: i) there is a distinguished vertex, called *root*, that has no incoming edges, ii) every other vertex has exactly one incoming edge, and iii) there is an unique path from the root to any other vertex. In a rooted tree T, u is the *parent* of v (v is the *child* of u) if $(u, v) \in E(T)$. The transitive closure of the parent-child relation is called the *ancestor-descendant* relation. A *rooted ordered tree* is a rooted tree such that there is an order over the children of every vertex. In the rest of this paragraph about trees and also throughout Chapter 4, we refer to *rooted ordered trees* simply as *trees*.

Preorder traversal of a tree T is defined recursively as follows: first, visit $root(T)$; and then for every child c of $root(T)$ from left to right, perform a preorder traversal on the subtree rooted at c. The position of a vertex in the list of visited vertices during

a preorder traversal is called its *preorder number*. We use $p(v)$ to refer to the preorder number of vertex v. The *rightmost path* of T is the path from $root(T)$ to the last vertex of T visited in the preorder traversal. Two distinct vertices u and v are *relatives* if u is neither an ancestor nor a descendant of v. With $p(u) < p(v)$, u is a left relative of v, otherwise, it is a right relative.

A tree T is a *rightmost path extension* of a tree T' iff there exist vertices u and v such that: (i) $\{v\} = V(T) \setminus V(T')$, (ii) $\{(u,v)\} = E(T) \setminus E(T')$, (iii) u is on the rightmost path of T', and (iv) in T, v is a right relative of all children of u. We say that T is the rightmost path extension of T' with v attached at u and we denote T as $RExtend(T', v, u)$.

For trees, more specific definitions of morphisms are typically used. Let P and T be two trees. P is *subtree isomorphic* to T (denoted $P \preceq^i T$) iff there is a mapping $\varphi : V(P) \rightarrow V(T)$ such that:

- $\forall v \in V(P) : \lambda_P(v) = \lambda_T(\varphi(v))$,
- $\forall u, v \in V(P) : (u,v) \in E(P) \Rightarrow (\varphi(u), \varphi(v)) \in E(T)$, and
- $\forall u, v \in V(P) : p(u) < p(v) \Leftrightarrow p(\varphi(u)) < p(\varphi(v))$.

P is *isomorphic* to T (denoted $P \cong_i T$) iff $P \preceq_i T$ and $|V(P)| = |V(T)|$.

P is *subtree homomorphic* to T (denoted $P \preceq^{hm} T$) iff there is a mapping $\varphi : V(P) \rightarrow V(T)$ such that:

- $\forall v \in V(P) : \lambda_P(v) = \lambda_T(\varphi(v))$,
- $\forall u, v \in V(P) : (u,v) \in E(P) \Rightarrow (\varphi(u), \varphi(v)) \in E(T)$, and
- $\forall u, v \in V(P) : p(u) < p(v) \Leftrightarrow p(\varphi(u)) \leq p(\varphi(v))$.

Note that, under subtree homomorphism, successive children of a vertex in P can be mapped onto the same vertex in T.

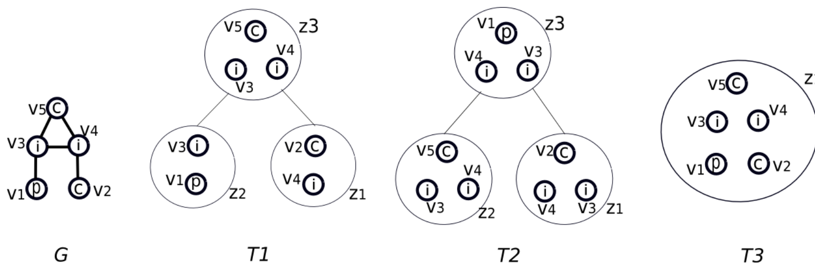
P is *subtree homeomorphic* to T (denoted $P \preceq^h T$) iff there is a mapping $\varphi : V(P) \rightarrow V(T)$ such that:

- $\forall v \in V(P) : \lambda_P(v) = \lambda_T(\varphi(v))$,
- $\forall u, v \in V(P) : u$ is the parent of v in P iff $\varphi(u)$ is an ancestor of $\varphi(v)$ in T, and
- $\forall u, v \in V(P) : p(u) < p(v) \Leftrightarrow p(\varphi(u)) < p(\varphi(v))$.

Note that a tree P under a subtree morphism can have several mappings to another tree T. In the context of tree mining, when the matching operator is subtree homeomorphism, we refer to every mapping as an *occurrence* (or *embedding*) of P in T. An occurrence (embedding) of a vertex v is an occurrence (embedding) of the tree consisting of the single vertex v. The number of occurrences of P in T is denoted by $NumOcc(P,T)$.

Tree decomposition A tree decomposition of a graph G is defined as a pair $(T, \mathcal{B})$ with T a rooted tree and $\mathcal{B} = (B(z))_{z \in V(T)}$ a family of subsets of $V(G)$ satisfying: i) $\cup_{z \in V(T)} B(z) = V(G)$, ii) for every $\{u, v\} \in E(G)$, there is a $z \in V(T)$ such that $u, v \in B(z)$, and iii) $B(z_1) \cap B(z_3) \subseteq B(z_2)$ for every $z_1, z_2, z_3 \in V(T)$ such that z_2 is on the path connecting z_1 with z_3 in T. The set $B(z)$ associated with a node z of T is called the bag of z. The *width* of the tree decomposition is defined as $max_{z \in V(T)} |B(z)| - 1$ and the *treewidth* of G is the minimum width over all tree decompositions of G (Diestel, 2005; Halin, 1976). We use $treewidth(G)$ to refer to the treewidth of G. The *height* of the tree decomposition, denoted by $height(T)$, is defined as the size of its longest path starting from the root. As a common convention, we call the vertices of G "vertices" and those of $TD(G)$ "nodes" (with the associated "bags"), the edges of G "edges" and those of $TD(G)$ "branches". To distinguish between the vertices of a graph and those of its tree decomposition, we use $v_1, v_2, \ldots$ for the former and $z_1, z_2, \ldots$ for the latter. Given an integer tw, it is NP-complete in terms of tw and $|V(G)|$, to decide if G is a tw-bounded-treewidth graph (Arnborg et al., 1987). For example, Figure 2.1 shows three different tree decompositions of the same graph. For a node z in a tree decomposition, $V_T(z)$ denotes the union of the vertices in the bags of the nodes of the subtree of the tree decomposition rooted at z.

Figure 2.1: G is a graph and $T1$, $T2$ and $T3$ are three tree decompositions of it. For example, T_1 has three nodes z_1, z_2 and z_3, where $B(z_1) = \{v_2, v_4\}$, $B(z_2) = \{v_1, v_3\}$, and $B(z_3) = \{v_3, v_4, v_5\}$. $T1$, $T2$ and $T3$ have *width* 2, 2 and 4, respectively. Observe that no tree decomposition with width 1 is possible for the subgraph with vertices v_3, v_4 and v_5, hence the treewidth is 2.



2.2 Poset and closure operator

Poset. A *partial order* is a binary relation $\leq$ over a set S which is *reflexive*, *antisymmetric* and *transitive*. In other words, for all $a, b, c \in S$ the followings hold:

- $a \leq a$ (reflexivity);
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry);
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

S is called a *partially ordered set* (or a *poset*, in short) (Simovici and Djeraba, 2014). We may refer to it as $(S, \leq)$ to emphasize the binary relation $\leq$.

Closure operator. Let $(S, \leq)$ be a poset. A mapping $\sigma : S \to S$ is called a *closure operator* iff it satisfies the following conditions for all $s_1, s_2 \in S$:

- (C1.) $s_1 \leq \sigma(s_1)$ (*extensivity*),
- (C2.) $s_1 \leq s_2 \Rightarrow \sigma(s_1) \leq \sigma(s_2)$ (*monotonicity*),
- (C3.) $\sigma(\sigma(s_1)) = \sigma(s_1)$ (*idempotence*).

An element $s \in S$ is *σ-closed* if $\sigma(s) = s$.

Another, slightly weaker operator of interest is the *pseudo closure* operator. A mapping $\sigma : S \to 2^S$ is called *pseudo closure operator*[1] iff it satisfies the following conditions for all $s_1, s_2 \in S$:

- (P1.) For all $s \in \sigma(s_1)$, $s_1 \leq s$, (*extensivity*),
- (P2.) If $s_1 \leq s_2$, then there do not exist different elements $p_2 \in \sigma(s_2)$ and $p_1 \in \sigma(s_1)$ such that $p_2 < p_1$, (*pseudo-monotonicity*),
- (P3.) For all $s \in \sigma(s_1)$, $|\sigma(s)| = 1$ and $\sigma(s) = \{s\}$ (*idempotence*).

An element $s \in S$ is *σ-pseudo closed* if $\sigma(s)$ is singleton and $\sigma(s) = \{s\}$.

The pseudo closure operator is a relaxation of the closure operator in two aspects. First, instead of the *monotonicity* property, it has the *pseudo-monotonicity* property. Second, in the pseudo closure operator an element may have more than one closure. The following proposition follows directly from the definitions.

[1] Note that the notion of *pseudo closure operator* introduced here is different from the notion of *pseudo closure operator* used by Pasquier et al. (1999).

Proposition 1. *A closure operator is a pseudo-closure operator, too.*

2.3 Data mining

Extension (refinement) operator. Let $\mathcal{G}$ be a graph class and $\preceq$ the subgraph relation on $\mathcal{G}$. Given the poset $(\mathcal{G}, \preceq)$, an *extension (refinement) operator* is a function $\rho : \mathcal{G} \rightarrow 2^{\mathcal{G}}$ such that for every $P \in \mathcal{G}$ and every $P' \in \rho(P)$ the followings hold: i) $P \prec P'$, i.e., P' is an extension of P, and ii) there does not exist $Q \in \mathcal{G}$ such that $P \prec Q \prec P'$, i.e., P' is a minimal extension (Raedt and Dehaspe, 1997). In the literature, in some cases the minimality condition on the extensions is dropped, especially when homomorphism is considered (Nienhuys-Cheng and De Wolf, 1997).

Given an extension operator ρ, we define $\rho^0(P) = \{P\}$ and for an integer $i > 0$, $\rho^i(P) = \cup_{P' \in \rho(P)} \rho^{i-1}(P')$. Moreover, $\rho^*(P) = \cup_{i=1}^{\infty} \rho^i(P)$. We denote the empty graph by $\perp$, i.e., for every graph P, $\perp \prec P$. The extension operator ρ is complete iff for every $P, Q \in \mathcal{G}$ such that $P \prec Q$, there is a $P' \in \rho^*(P)$ such that $P' \cong^i Q$. It is optimal iff for every $P_1, P_2 \in \mathcal{G}$ such that $\rho^*(P_1) \cap \rho^*(P_2) \neq \emptyset$, we have $P_2 \in \rho^*(P_1)$ or $P_1 \in \rho^*(P_2)$.

Figure 2.2 presents all minimal extensions of a rooted ordered tree, a rooted unordered tree, and a free tree.

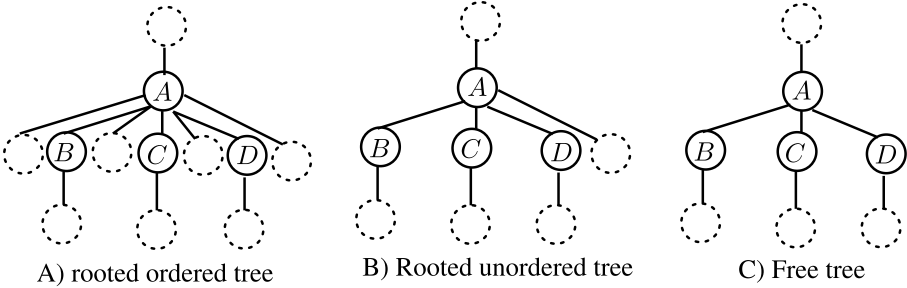


Figure 2.2: All minimal extensions of a rooted ordered tree, a rooted unordered tree, and a free tree.

Pattern mining. A *pattern mining setting* is a triple $(L_d, L_p, \leq)$ where :

- L_d is a class of graphs, called the *database language*,
- L_p is a subclass of L_d, called the *pattern language*, and

- $\leq$ is a matching operator between two graphs $P \in L_p$ and $D \in L_d$.

A database for the pattern mining setting $(L_d, L_p, \leq)$ is a multiset of graphs of L_d (Moens et al., 2014). There are two main settings for pattern mining: *transactional pattern mining* where the database is a collection of graphs, and *single network mining* where the database is one large graph.

An *interestingness predicate* ι for $(L_d, L_p, \leq)$ maps every pair (D, P) to either true or false, where $D \in L_d$ and $P \in L_p$. The input of the problem of ι-interesting pattern mining for $(L_d, L_p, \leq)$ is a database $D \in L_d$ and the task is to list all elements $P \in L_p$ for which $\iota(D, P)$ is true (Moens et al., 2014). A widely used interestingness predicate is the *frequency (support)* measure. In transactional pattern mining, this measure can be defined e.g., as follows. Let $(L_d, L_p, \leq)$ be a pattern mining setting, $D \in L_d$ and $P \in L_p$. *Frequency (support)* of P in D, denoted with $freq(D, P)$, is defined as follows (Moens et al., 2014):

$$freq(D, P) = |\{T \in D \mid P \leq T\}| \tag{2.1}$$

P is frequent in D with respect to a user-defined threshold $minsup$ if $freq(D, P) \geq minsup$. In single network mining, defining an appropriate frequency measure is not always straight forward. We briefly discuss it in Chapter 3.

The input of the problem of frequent pattern mining for the pattern mining setting $(L_d, L_p, \leq)$ is a database $D \in L_d$ and a user-defined threshold $minsup$, and the task is to list all patterns $P \in L_p$ for which $freq(D, P) \geq minsup$ (Moens et al., 2014).

Closed patterns. There are two widely used pseudo-closure operators in the context of data mining: the frequency based closure operator, denoted by σ_F (Chi et al., 2005b), and the image based closure operator, denoted by σ_I (Garriga et al., 2007). A pattern P is σ_F-closed iff it does not have any super pattern P' such that the frequency of P is equal to the frequency of P' (Chi et al., 2005b). A pattern P is σ_I-closed iff it does not have any super pattern P' such that every image of P can be extended to an image of P' (Garriga et al., 2007).

2.4 Betweenness centrality

Let G be a simple and connected graph. A *shortest path* (which is also called a *geodesic path*) between two vertices $u, v \in V(G)$ is a path whose size is minimum, among all paths between u and v. For two vertices $u, v \in V(G)$, we use $d(u, v)$ to denote the number of edges of a shortest path connecting u to v.

For $s, t \in V(G)$, σ_{st} denotes the number of shortest paths between s and t and $\sigma_{st}(v)$ the number of shortest paths between s and t that also pass through v. We have

$$\sigma_s(v) = \sum_{t \in V(G) \setminus \{s,v\}} \sigma_{st}(v)$$

Betweenness centrality of a vertex v is defined as:

$$BC(v) = \sum_{s,t \in V(G) \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.2}$$

In Equation 2.2, undefined terms $\frac{0}{0}$ are treated as 0.

A widely used notion in counting the number of shortest paths in a graph is the directed acyclic graph (DAG) that contains all shortest paths starting from a vertex s (see e.g., (Brandes, 2001)). In this dissertation, we refer to it as the *shortest-path-DAG*, or *SPD* in short, rooted at s. For every vertex s in graph G, the *SPD* rooted at s is unique, and it can be computed in $O(|E(G)|)$ time for unweighted graphs and in $O(|E(G)| + |V(G)| \log(|V(G)|))$ time for weighted graphs (with positive weights) (Brandes, 2001) .

Brandes (2001) introduced the notion of the *dependency score* of a vertex $s \in V(G)$ on a vertex $v \in V(G) \setminus \{s\}$. It is defined as:

$$\delta_{s\bullet}(v) = \sum_{t \in V(G) \setminus \{v,s\}} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.3}$$

We have:

$$BC(v) = \sum_{s \in V(G) \setminus \{v\}} \delta_{s\bullet}(v) \tag{2.4}$$

Brandes (2001) showed that dependency scores of a source vertex on other vertices in the (unweighted) network can be computed using the following recursive relation:

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)) \tag{2.5}$$

where $P_s(w)$ is:

$$\{u \in V(G) : \{u, w\} \in E(G) \wedge d(s, w) = d(s, u) + 1\}$$

In other words, $P_s(w)$ contains the set of all parents (predecessors) of w in the SPD rooted at s.

As mentioned by Brandes (2001), given the SPD rooted at s, dependency scores of s on all other vertices in the graph can be computed in $O(|E(G)|)$ time.

Chapter 3

Mining Large Single Networks under Subgraph Homomorphism

3.1 Introduction

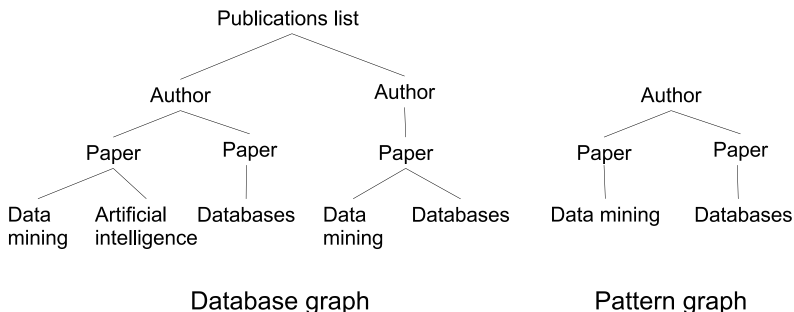
The problem of finding frequent patterns from a database of graphs or from a single network has several important applications in different areas such as the World Wide Web, bioinformatics, chemo-informatics and social and information networks. It is also a fundamental problem in many other data mining tasks such as association rule mining, classification and clustering. The focus of this chapter is the single network mining setting.

Existing algorithms for finding frequent patterns from a large single network mainly use *subgraph isomorphism* (Ghazizadeh and Chawathe, 2002; Cook et al., 1995; Kuramochi and Karypis, 2004, 2005; Vanetik et al., 2002). Subgraph isomorphism is an *injective* mapping, i.e., two vertices in the pattern graph may not be mapped onto the same vertex in the network at the same time. However, subgraph isomorphism is expensive to compute: deciding whether one graph, even if it is as simple as a path, is subgraph isomorphic to another graph is NP-complete in terms of network size and pattern size. Therefore, in massive networks consisting of millions of vertices subgraph isomorphism is intractable. For this reason, we argue that it may be interesting to use *subgraph homomorphism* since for patterns of bounded-treewidth complexity of this matching operator only depends polynomially on the network size

and pattern size. Subgraph homomorphism is not an injective mapping: two vertices in the pattern graph might be mapped onto the same vertex of the network.

Besides the efficiency concern, pattern mining under homomorphism has also an intrinsic value as there are patterns that are frequent under homomorphism but not under isomorphism. For example, as discussed by Dries and Nijssen (2012), assume a database is given in which vertices are labeled by **Author**, **Paper** and keywords of the papers; and edges indicate the authors and keywords of a paper (see Figure 3.1; there is also a single vertex labeled by **Publications list** which is connected to all **Author**s to build a connected network). Suppose we are looking for patterns that occur in the database graph at least twice. A pattern that is expressed using homomorphism but not using isomorphism, is depicted in Figure 3.1. This pattern presents the authors who have at least 2 papers in **Data mining** and two (not necessarily different) papers in **Databases**. Here, the key point is that under homomorphism an object can have multiple roles, while it may have only one role under isomorphism. In the given example, under homomorphism, the database **Paper** vertex which is the image of 2 pattern **Paper** vertices, finds two roles in the pattern: a paper on **Databases** and at the same time a paper on **Data mining**.

Figure 3.1: A database graph (left) and a pattern that is expressed using homomorphism but not using isomorphism (right).



The most challenging phase in single network mining under homomorphism is the pattern generation phase (that involves a *refinement operator* (Muggleton and Raedt, 1994)). In particular, as mentioned by Dries and Nijssen (2012),

(i) an infinite number of patterns can be frequent.

(ii) two patterns of different sizes may be homomorphic with each other. This makes the ordered search more complicated than in the case of subgraph isomorphism (Dries and Nijssen, 2012).

In the field of Inductive Logic Programming (ILP), there is a long history of attempts that tackle the above mentioned problems. It can be shown that under homomorphism, an optimal refinement operator does not exist, even for simple classes of patterns (Nijssen and Kok, 2010; Ramon et al., 2011). Ramon et al. (2011) showed that homomorphism-free bounded-treewidth graphs can be enumerated efficiently with a polynomial delay. Existing pattern mining algorithms mainly deal with the above mentioned problems by restricting the pattern class and/or the database class. For example, in the Warmr system proposed by Dehaspe and Toivonen (1999), frequent *cores* are discovered from an acyclic database. A graph is *core* iff it is not subgraph homomorphic to any of its (strict) subgraphs. Any non-core is equivalent, under homomorphism, to a core. Restricting the patterns to cores and also the database to acyclic graphs resolves the above mentioned problems. Dries and Nijssen (2012) presented the htreeminer algorithm to find frequent rooted patterns from an arbitrary large single network. They addressed the above mentioned problems by restricting the patterns to height-bounded trees that are core.

In the current work, we study the problem of single network mining under homomorphism for a class of patterns more general than rooted trees. We introduce the class of *rooted graph patterns* and address the aforementioned pattern generation problems for this class:

- In order to have always a finite set of frequent patterns (i.e., to address problem (i)), on the one hand, we define the notion of *height* of a rooted pattern and generate only rooted patterns that have a bounded height. On the other hand, we only generate rooted patterns that are *core*. Both these conditions, i.e., being a core and having a bounded height, are necessary to have a finite set of patterns. We show that these two conditions are sufficient for the finiteness of patterns under homomorphism.
- By generating only patterns that are core, problem (ii) is also addressed, since no two cores, of different sizes, are homomorphic to each other.
- In order to have a polynomial time matching operator (in terms of both network size and pattern size), we further restrict our patterns to bounded-treewidth graphs, i.e., every generated pattern has a bound on its treewidth.

The class of height-bounded cores that are investigated by Dries and Nijssen (2012), is a special case of our pattern class where the treewidth is bounded by 1.

Our key contributions in this work are:

1. We formulate the problem of finding frequent *bounded-treewidth height-bounded cores*. As mentioned above, this class of patterns is finite and can be discovered effectively.

2. We propose a new method for generating *bounded-treewidth height-bounded cores*, and show its completeness.

3. We present an efficient algorithm, called HoPa[1], for listing *bounded-treewidth height-bounded cores* that are frequent with respect to a database and a minimum support threshold. We also present optimization techniques to improve the algorithm.

4. We introduce a new data structure for the compact representation of all frequent rooted patterns. Compact representations are useful when the set of all frequent patterns is huge. Most of existing closedness data structures for single network mining are not based on a real closure operator. We show that the proposed data structure gives a closure operator for finding rooted patterns from a single network under homomorphism.

5. We empirically evaluate HoPa on different synthetic and real-world networks and show its high efficiency, compared to existing methods.

Our proposed pattern mining algorithm is a level-wise method that follows the following steps: first it generates a new bounded-treewidth height-bounded rooted pattern P, then it checks if P is core and, if so, it counts the frequency of P, and finally it computes the closed pattern of P. The empirical efficiency of the algorithm is improved by several optimization techniques. For example, since it is computationally very expensive to count frequency of a pattern in a large network, HoPa stores a list of already found infrequent patterns. Then, before counting frequency of the pattern, it checks if it is a supergraph, under homomorphism, of any infrequent patterns. If so, it immediately discards the pattern without counting its frequency.

The rest of this chapter is organized as follows. A short overview of related work is given in Section 3.2. In Section 3.3, we present the notion of rooted graphs and an algorithm for generating them. In Sections 3.4 and 3.5, we respectively introduce our proposed algorithms for finding frequent patterns and frequent closed patterns. In Section 3.6, we empirically evaluate the proposed algorithm and finally, the chapter is concluded in Section 3.7.

3.2 Related work

In this section, we provide a short overview of related work. First in Section 3.2.1, we discuss efficient pattern matching algorithms. Then in Section 3.2.2, we briefly investigate two main categories of pattern mining approaches. Finally in Sections

[1] HoPa is an abbreviation for **Ho**momorphism **S**ubgraph **Pa**tterns

3.2.3 and 3.2.4, we discuss some of the well-known algorithms for pattern mining in single networks and in transactional databases, respectively.

3.2.1 Efficient pattern matching algorithms

For general graphs, both subgraph isomorphism and subgraph homomorphism are NP-complete (Garey and Johnson, 1979). There are several approaches in the literature to deal with the hardness of the problems. One approach is to use heuristics that make the algorithms empirically efficient. The widely used subgraph isomorphism algorithms for the general graphs are the algorithm of Ullmann (1976) and the VF2 algorithm (Cordella et al., 2001) that both use the branch-and-bound and backtracking techniques to prune the search space. VF2 usually outperforms the Ullman algorithm in terms of running time (Cordella et al., 2004). Note that these algorithms can be used for subgraph homomorphism as well. Another approach is to restrict the pattern class and/or the database class. There exist several polynomial time algorithms, in terms of both network size and pattern size, for the specific classes of graphs. Hajiaghayi and Nishimura (2007) proposed such an algorithm for the subgraph isomorphism between a log-bounded fragmentation graph G and a bounded-treewidth graph H. A graph of size n is a *log-bounded fragmentation* graph if the removal of any set of at most k vertices from it results in $O(k \log n)$ connected components. This algorithm is an extension of the polynomial time subgraph isomorphism algorithm presented by Matousek and Thomas (1992a) between a graph G with a bounded degree and a graph H with a bounded treewidth. Matousek and Thomas (1992a) also showed that when H is a bounded-treewidth graph and G is an arbitrary general graph, subgraph homomorphism between G and H can be done in polynomial time in terms of $|V(G)|$ and $|V(H)|$. Before that, Robertson and Seymour (1986) showed that subgraph homomorphism between G and H can be done in $O(|V(H)|^3)$ time for every fixed G and in $O(|V(H)|)$ time if G is a path.

A third approach is to employ randomized techniques. Koutis and Williams (2009) presented a randomized algorithm for subgraph isomorphism between a tree and a network that runs in time $O(k^2 \log^2(k) m 2^k)$, where k is the pattern size and m is the number of edges of the network. The algorithm consists of two main parts. In the first part, it constructs an arithmetic circuit computing a polynomial that represents all possible homomorphisms of the tree in the network. Subgraph homomorphism between a tree and a network can be done in a time polynomial in terms of both network size and pattern size. In its second part, it evaluates the polynomial on an appropriate commutative group algebra, that ensures all terms that are not multilinear (i.e., all homomorphisms that are not isomorphisms) vanish. Later, Fomin et al. (2012) extended this algorithm to bounded-treewidth graphs and showed that if the treewidth of the pattern G is bounded by tw, there is a randomized algorithm for subgraph

isomorphism between G and a network H running in time $O(2^k n^{2tw})$, where k and n are the sizes of G and H, respectively.

In summary, while the morphism problems are intractable in theory, several efforts have been done to make them empirically tractable. In this section, we briefly reviewed the three main approaches that are useful in different domains, including frequent pattern discovery.

3.2.2 Pattern mining strategies

Two main categories of pattern mining strategies are *breadth-first search* algorithms and *depth-first search* algorithms. In the breadth-first approach, patterns are generated level-wise, i.e., first all patterns of size 1 are generated and counted; then patterns of size 2 are generated and counted, and so on. In these algorithms, when a pattern is generated, the frequency of all patterns that are more general is already counted. This means there is a maximal opportunity to find out if the pattern under consideration has any infrequent sub-pattern. However, this also means that the breadth-first algorithms store lots of information in memory. Examples of breadth-first algorithms include Inokuchi et al. (2002) and Kuramochi and Karypis (2001).

In the depth-first approaches, patterns are generated depth-first. These algorithms have the advantage that when one branch of the search is finished, all information concerning that branch can be discarded. As a result, only a minimum of memory is required to store patterns, and available memory can be used for other optimizations. Examples of depth-first algorithms include Borgelt and Berthold (2002) and Yan and Han (2002).

3.2.3 Single network mining

SUBDUE (Cook et al., 1995) is a well-known algorithm for finding frequent subgraphs from a single network under subgraph isomorphism. It employs the minimum description length (MDL) principle to discover subgraphs that compress the network and represent structural concepts. However, the heuristics used in SUBDUE make it hard to find large frequent patterns. SEuS, proposed by Ghazizadeh and Chawathe (2002), uses a summary data structure to prune the search space and provide interactive feedback. In this method, all vertices with the same label are collapsed together. This technique is especially useful at the presence of a relatively small number of highly frequent subgraphs.

Kuramochi and Karypis (2004) presented GREW to find large patterns that have a large number of vertex-disjoint embeddings (under isomorphism) in a dense network.

However, because of the heuristic nature of GREW, the number of patterns discovered by it is significantly smaller than those discovered by complete algorithms. In Kuramochi and Karypis (2005), the same authors presented two algorithms to find connected subgraphs that have a sufficient number of edge-disjoint embeddings in a single large sparse graph. In their method, in order to reduce the number of redundant candidates, subgraphs of size k are joined for growth only if they share a certain subgraph of size $k-1$. They also used a structure, called *anchor-edge-list*, to efficiently handle the edge-disjoint embeddings of a frequent pattern and reduce the cost of the subgraph isomorphism test.

Zhu et al. (2011) developed the Spider-Mine algorithm for mining top-K largest frequent patterns from a single network. Spider-Mine finds small patterns of a bounded diameter, called *spiders*, and uses them in a probabilistic mining framework to find the top-k largest patterns. The interesting techniques used in this work are the identification of a set of promising growth paths toward large patterns and the reduction of the cost of graph isomorphism tests with a new graph pattern representation based on a multi-set of spiders. Based on the subgraph isomorphism algorithm of Koutis and Williams (2009), Kibriya and Ramon (2012) proposed an algorithm for mining frequent trees in a network. For a fixed parameter k (maximal pattern size), their mining algorithm can mine all rooted trees with a delay linear in the size of the network and exponential in k.

Dries and Nijssen (2012) presented the first algorithm, called htreeminer, that finds frequent rooted trees from a large network under homomorphism in *incremental polynomial time*. *Incremental polynomial time* means that the time spent between listing two consecutive patterns is polynomial in the size of the input and the number of patterns found till that point. They introduced novel support measures and extended the method to find closed and maximal patterns under homomorphism. The mining problem studied in our current work is a generalization of the mining problem investigated by Dries and Nijssen (2012). In particular, we go beyond tree patterns and propose an algorithm for finding frequent graph patterns from a large single network under homomorphism. As mentioned before, the most challenging step in pattern mining under homomorphism is the pattern generation phase. Hence, the main concern in the current work is to address the pattern generation problem for a class of patterns more general than rooted trees, called *bounded-treewidth height-bounded cores*. We then present an efficient algorithm, called HoPa, for finding this class of patterns from large single networks under homomorphism. The frequent patterns found by htreeminer is a special case of the frequent patterns found by HoPa, where the treewidth of the patterns is bounded by 1. Our extensive experiments show that in this special case, HoPa mostly outperforms htreeminer.

3.2.4 Transactional graph mining

There are several efficient algorithms for finding frequent patterns in the transactional setting, where the database consists of many relatively small graphs, instead of a single large graph (Inokuchi et al., 2000; Kuramochi and Karypis, 2001; Yan and Han, 2002). Yan and Han (2002) proposed gSpan (graph-based Substructure pattern mining) for finding frequent graph substructures. The gSpan algorithm builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan adopts the depth-first search strategy to mine frequent connected subgraphs. Later, Yan and Han (2003) developed the CloseGraph algorithm for discovering frequent closed graph patterns. CloseGraph is a variation of gSpan that takes advantage of early pruning techniques in order to avoid generation of all frequent patterns. Hasan and Zaki (2009) proposed the idea of *output space sampling* in the domain of frequent subgraph mining, which samples interesting subgraph patterns without enumerating the entire set of candidate frequent patterns. They presented a generic sampling framework, that is based on the Metropolis-Hastings algorithm.

3.3 Rooted patterns and their generation

In this section, first in Section 3.3.1 we introduce rooted graphs, which will be used as patterns. Then, in Section 3.3.2, we present our proposed algorithm for generating them.

3.3.1 Rooted patterns

In choosing the class of patterns that are listed in a mining problem, someone may consider either rooted patterns or unrooted ones. Using any of them may result in different consequences, e.g.,

- For rooted patterns there exist simple and effectively computable support measures such as *the number of root images* (Bringmann and Nijssen, 2007); whereas for unrooted patterns more complicated support measures, such as those that are based on *overlap graph* (Calders et al., 2011), are required.
- Since several rooted patterns may represent the same underlying unrooted graph, the number of rooted patterns generated during the mining process may be considerably larger than the number of unrooted ones. This can affect the efficiency of the mining problem.

In frequent patterns discovery, every individual pattern can be generated effectively, however, it is computationally expensive to count its frequency. In the current work, we choose rooted patterns as our pattern language which is consistent with the support measure and the closure operator we use. As we will respectively discuss in Sections 3.4 and 3.5, we use the number of root embeddings as the support measure; and a closure operator based on root embeddings for compact representation of patterns. There are several algorithms in the literature that find unrooted patterns (Ghazizadeh and Chawathe, 2002; Kuramochi and Karypis, 2004).

Definition 1 (rooted graph)**.** *A rooted graph P^X is a graph P where a set $X \subseteq V(P)$ is distinguished.*

The class of rooted graphs is denoted by $\mathcal{G}^r$. Since rooted graphs will form our class of patterns, we also refer to them as *rooted patterns*. Throughout the chapter, we assume rooted patterns are connected, simple and finite. For a rooted pattern P^X, $R(P^X)$ refers to the subgraph of P induced by the root vertices.

The root of the pattern will allow us to define the height of a rooted pattern; putting a bound on the height of the patterns to be generated will help us to ensure termination of the generation process.

Let P^X and Q^Y be two rooted patterns. P^X is a *root subgraph* of Q^Y iff $E(P) \subseteq E(Q)$, $X \subseteq Y$ and $V(P) \setminus X \subseteq V(Q) \setminus Y$. A *root graph homomorphism* from P^X to Q^Y is a homomorphism mapping φ from P to Q that maps root vertices of P^X (i.e., vertices that are in X) to root vertices of Q^Y (i.e., vertices that are in Y) and non-root vertices of P^X (i.e., vertices that are not in X) to non-root vertices of Q^Y (i.e., vertices that are not in Y). We use $P^X \cong^{hm} Q^Y$ to mention that P^X is *root homomorphic* to Q^Y. $P^X \cong^{hm}_{\varphi} Q^Y$ is used to explicitly mention the homomorphism mapping φ. P^X is *root subgraph homomorphic* to Q^Y, denoted by $P^X \preceq^{hm} Q^Y$, if it is root homomorphic to a root subgraph of Q^Y. We use $P^X \preceq^{hm}_{\varphi} Q^Y$ to explicitly mention the mapping φ. A *root subgraph isomorphism* from a rooted pattern P^X to a rooted pattern Q^Y, denoted by $P^X \preceq^{i} Q^Y$, is a root subgraph homomorphism where no two vertices have the same image. We use $P^X \preceq^{i}_{\varphi} Q^Y$ to explicitly mention the isomorphism mapping φ. P^X is *rooted graph isomorphic* to Q^Y, denoted by $P^X \cong^{i} Q^Y$, iff $P^X \preceq^{i} Q^Y$ and $Q^Y \preceq^{i} P^X$.[2]

With φ a mapping from P to Q and S a subset of $V(P)$, $\varphi_{|S}$ denotes φ restricted to the vertices of S. Let $P^X \preceq^{i}_{\varphi} Q^Y$ ($P^X \preceq^{hm}_{\varphi} Q^Y$) and $Q^Y \preceq^{i}_{\varphi'} N^Z$ ($Q^Y \preceq^{hm}_{\varphi'} N^Z$). Then, $\varphi' \circ \varphi$ is a mapping from P^X to N^Z that maps every vertex $v \in V(P^X)$ to $\varphi'(\varphi(v))$.

[2]The matching operators presented in this paragraph are used when we compare two rooted patterns. As we will discuss in Section 3.4, for counting frequency of a rooted pattern in a network, we use the subgraph homomorphism matching operator presented in Chapter 2.

As mentioned earlier, when the pattern graph has a bounded treewidth, time complexity of subgraph homomorphism depends only polynomially on the network size and pattern size. This motivates us to restrict rooted patterns to the class of rooted bounded-treewidth graphs. Hence, in the rest of this chapter, we assume the treewidth of every rooted pattern is bounded by an integer tw. This, further, motivates us to generate rooted patterns in the form of tree decompositions, i.e., instead of enumerating rooted patterns and checking if every generated rooted pattern has a bounded treewidth, we generate rooted tree decompositions of rooted patterns in such a way that the underlying pattern has its treewidth bounded by tw.

A first difficulty with enumerating tree decompositions is that different rooted patterns can have the same tree decomposition. To overcome this, in Definition 2 we introduce *extended tree decompositions*, where the bag not only consists of vertices, but also of edges, so that each tree decomposition identifies a unique rooted pattern.

Definition 2 (extended tree decomposition (ETD))**.** *An* extended tree decomposition (ETD) *of a rooted pattern P^X is a tree decomposition of P^X in which each bag contains the subgraph of P induced by the vertices in the bag. We refer to the induced subgraph of a node z by $\mathcal{B}(z)$ and to the vertices and edges of $\mathcal{B}(z)$ by $\mathcal{V}(z)$ and $\mathcal{E}(z)$, respectively.*

Let $(T, \mathcal{B})$ be an extended tree decomposition. We denote with $Pat(T)$ the underlying rooted pattern represented by the tree decomposition, and with T_{-z} the tree T where the node z and its whole subtree are removed.

Generating extended tree decompositions is also no easy undertaking. To start with, a single pattern can have an infinite number of extended tree decompositions (even for a single node pattern "a", any tree with (identical) nodes "a" is a tree decomposition). So, we need to impose constraints on the shape of an extended tree decomposition as to eliminate redundancy (while still ensuring all patterns have a tree decomposition) and to obtain a finite set.

A first step is to limit attention to *rooted extended tree decompositions*:

Definition 3 (rooted extended tree decomposition (RETD))**.** *A* rooted extended tree decomposition *of a rooted pattern P^X is an extended tree decomposition of P^X that satisfies the following additional constraints:*

(i) The subgraph in the root node of the tree decomposition is $R(P^X)$, and

(ii) It does not have any (directed) edge (z, z') such that $\mathcal{B}(z)$ (i.e., the graph of the parent node) is a subgraph of $\mathcal{B}(z')$ (i.e., the graph of the child node).

The conditions expressed in Definition 3 put an extra constraint on rooted patterns: a rooted pattern P^X must have a RETD whose root bag consists of the vertices in X.

The *width* of an RETD $(T, \mathcal{B})$ is defined as $\max_{z \in V(T)} |\mathcal{V}(z)| - 1$. In Definition 4, we define *treewidth* and *height* of rooted patterns.

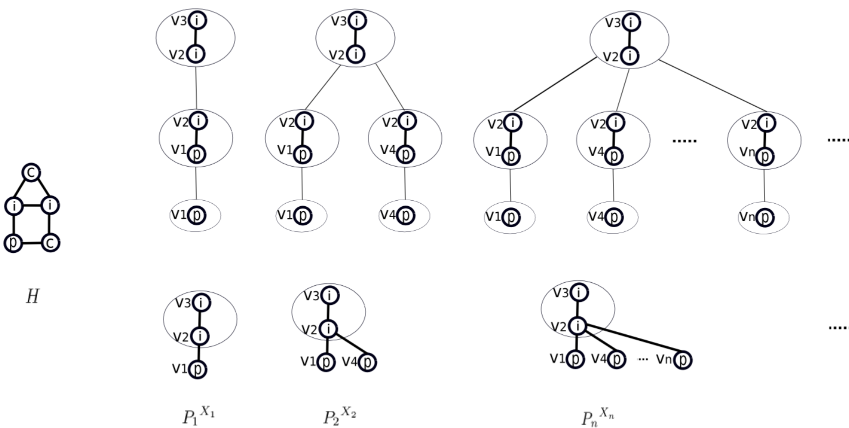
Definition 4 (*treewidth* and *height* of a rooted pattern)**.** *The* treewidth *of a rooted pattern is defined as the minimum width of all its RETDs. The* height *of a rooted pattern with treewidth tw is defined as the minimum height of all its RETDs with width at most tw.*

Another problem is that under homomorphism, there can be an unbounded number of frequent rooted patterns of treewidth tw. For example, in Figure 3.2, there are an infinite number of rooted patterns ${P_1}^{X_1}$, ${P_2}^{X_2}$, ..., ${P_n}^{X_n}$, ... that are subgraph homomorphic to the network H and to each other. This motivates us to consider only rooted patterns that are *core*.

Definition 5 (core)**.** *A rooted pattern P^X is a core iff every subgraph homomorphism $\varphi : P \to P$ is a graph automorphism.*

For example in Figure 3.2, only ${P_1}^{X_1}$ is a core. It follows directly from the definition of core that any rooted pattern P^X is equivalent, under homomorphism, to a core, denoted with $Core(P^X)$. We note that slightly different definitions of core are also possible, e.g., P^X is core iff every root subgraph homomorphism φ form $P^X \to P^X$ is a root graph automorphism. While both definitions suit our mining algorithm fine and any definition can be easily replaced by the other, in the rest of the current chapter we use the first definition.

Figure 3.2: There are an infinite number of rooted patterns ${P_1}^{X_1}$, ${P_2}^{X_2}$, ..., ${P_n}^{X_n}$, ... that are subgraph homomorphic to the network H and to each other. The tree decomposition above each rooted pattern shows its BRETD.



The set of rooted patterns that are core, without any further restriction, may still be infinite. For example, suppose that the database graph is a directed cycle $v_1 \to v_2 \to \ldots \to v_n \to v_1$, where all vertices and all edges have the same label. Then, any directed path $u_1 \to u_2 \to \ldots$ of an arbitrary size, rooted at the first vertex, will be a core and subgraph homomorphic to the database graph.

All the above motivates the following definition of BRETD.

Definition 6 (bounded rooted extended tree decomposition (BRETD))**.** *A* bounded rooted extended tree decomposition (BRETD) $(T, \mathcal{B})$ *with width bound tw and height bound h is a rooted extended tree decomposition $(T, \mathcal{B})$ that satisfies the following additional constraints:*

(i) $height(T) \leq h$,

(ii) For every non-leaf node z and every child c of z, there is exactly one vertex which is in $\mathcal{V}(z)$ but not in $\mathcal{V}(c)$, i.e., $|\mathcal{V}(z) \setminus \mathcal{V}(c)| = 1$,

(iii) For every leaf node z, $|\mathcal{V}(z)| = 1$; and for every non-leaf node z, $|\mathcal{V}(z)| > 1$,

(iv) For every node z with multiple children $c_1, \ldots c_n$ ($n > 1$), $Pat(T) \neq Pat(T_{-c_i})$, and

(v) For every node z with multiple children $c_1, \ldots c_n$ ($n > 1$), $Core(Pat(T)) \neq Core(Pat(T_{-c_i}))$.

Note that in Definition 6, Condition (v) does not imply Condition (iv). For example, in Figure 3.3, removing the subtree of T_{PX} rooted at z_7 changes the pattern but not the core.

A *candidate BRETD* is a rooted extended tree decomposition that only satisfies conditions (i)-(iii) of Definition 6. Each candidate BRETD also defines a rooted pattern. With r the root of a BRETD and c a child of r, we refer to the subtree rooted at c as the *c-child BRETD* or, when c is clear from the context, *child BRETD*. In Proposition 2, we prove that every tw-bounded-treewidth height-h-bounded core has a BRETD with width at most tw and height at most $tw \times h + 1$. Note that the rooted pattern presented in Figure 3.3 is not core. So, while every tw-bounded-treewidth core has a BRETD, not every BRETD represents a core.

Proposition 2. *Let P^X be a connected core rooted pattern with treewidth tw and height h. Then there exists a BRETD T bounded by width tw and height $tw \times h + 1$ such that $Pat(T) = P^X$.*

Proof. Given the properties of P^X, there exists a finite RETD T with width tw and height h such that $Pat(T) = P^X$. It can be transformed into a BRETD of width bound tw and height bound $tw \times h + 1$ as follows:

1. Every leaf node z with more than one vertex is replaced by a path consisting of nodes $z_1, \ldots, z_l$ such that:
    - $z_1 = z$,
    - For $1 \leq i < l$: z_{i+1} is a child of z_i and $\mathcal{V}(z_{i+1}) = \mathcal{V}(z_i) \setminus \{\text{an arbitrary vertex in } \mathcal{V}(z_i)\}$,
    - $|\mathcal{V}(z_l)| = 1$.
2. Every non-leaf and non-root node z with 0 or 1 vertices is deleted and its children are added to the children of the parent of z.
3. Every path (z, z') with z' a child of z and $|\mathcal{V}(z) \setminus \mathcal{V}(z')| > 1$ is replaced by a path consisting of nodes $z_1, \ldots, z_l$ such that:
    - $z_1 = z$ and $z_l = z'$, and
    - for $1 \leq i < l$: z_{i+1} is a child of z_i and $\mathcal{V}(z_{i+1}) = \mathcal{V}(z_i) \setminus \{\text{an arbitrary vertex in } \mathcal{V}(z_i) \cap (\mathcal{V}(z) \setminus \mathcal{V}(z'))\}$.

Note that step 1 is applied at most once on every leaf node, step 2 at most once on every non-leaf and non-root node and step 3 at most once on every parent child pair of the original tree decomposition T; moreover, these steps preserve the pattern and ensure conditions (ii) and (iii) of Definition 6 hold for the resulting tree decomposition T'.

Next, for every node z such that $Pat(T') = Pat(T'_{-z})$, the subtree rooted at z is removed; this preserves conditions (ii) and (iii) while also ensuring condition (iv). Let T'' be the resulting tree decomposition. Note that $Pat(T)$ is a core and $Pat(T'') = Pat(T)$, hence also condition (v) holds. Finally, note that application of step 1 replaces a leaf node in the worst case by a path of size $tw + 1$ and step 3 a path of size 2 in the worst case by a path of size tw, so the height of the tree decomposition T'' is bounded by $tw \times h + 1$ and hence T'' is a BRETD bounded by width tw and height $tw \times h + 1$. □

Figure 3.3 shows a rooted pattern and a BRETD of it. As we will show later in Corollary 1 of Section 3.3.2, the set of *bounded-treewidth height-bounded core BRETDs* is always finite.

A rooted pattern may still have more than one BRETD. For example, in Figure 3.4, $T1$, $T2$ and $T3$ are three different BRETDs of the rooted pattern P^X. While we cannot avoid to generate different BRETDs that represent the same rooted pattern, we need to eliminate a BRETD when the rooted pattern it represents is not new. This boils down to detecting rooted graph isomorphism between the underlying rooted patterns. In general, the check has an exponential cost in terms of the pattern size. However, for

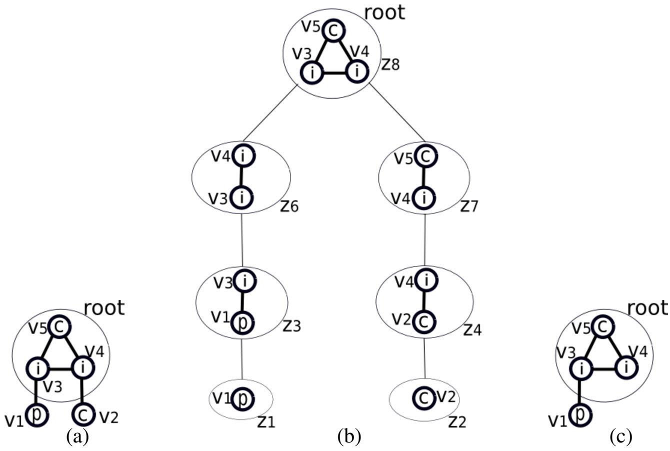


Figure 3.3: Figure (a) shows a rooted pattern P^X, (b) a BRETD of it and (c) $Core(P^X)$.

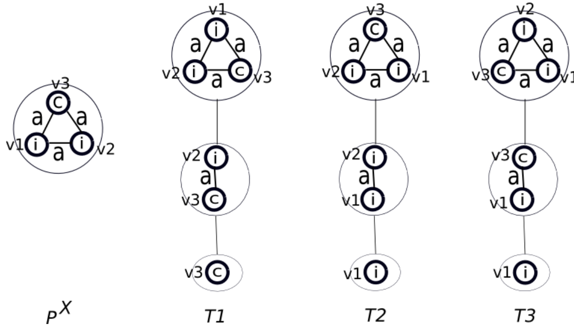


Figure 3.4: The possible BRETDs for the rooted pattern P^X are $T1$, $T2$ and $T3$.

tw-bounded-treewidth patterns, there are algorithms that check graph isomorphism with a $O(n^{tw+2})$ time cost, where n is the pattern size (Daenen, 2009; Ramon et al., 2011).

In the current chapter, when a BRETD with the underlying rooted pattern P^X is generated, we use the algorithm of Daenen (2009); Ramon et al. (2011) to compute the canonical form of P^X. Since here our patterns are rooted, we first compute the canonical form of $R(P^X)$ and then, the canonical form of the rest of the pattern, i.e., the subgraph of the pattern obtained by removing vertices in X that are only connected to other vertices in X and the edges that connect two vertices in X. Our algorithm stores the canonical forms of the generated rooted patterns in a trie data structure

to achieve a compact representation and fast access. Then, when a new BRETD is generated, its canonical form is computed and compared with the canonical forms of all already generated rooted patterns to see if the rooted pattern represented by the new BRETD is a new one.

3.3.2 Generating BRETDs

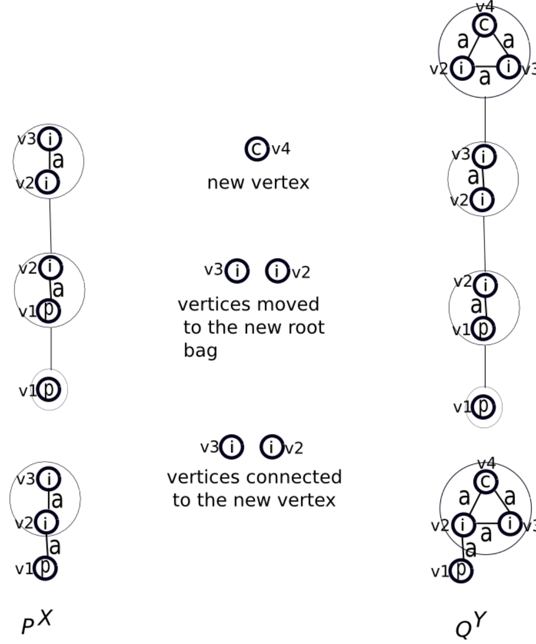
As mentioned earlier, we generate rooted patterns in the form of BRETDs which are rooted trees. Hence, in order to generate BRETDs, one can follow an approach for generating rooted trees. A first approach is rightmost path extension where a new tree is generated by adding a node to a node in the rightmost path of an input tree (see Chapter 2 for more details).

The second approach is a bottom-up method, which is based on two operators. An *extension* operator that extends an input rooted tree with a new root node that has the input tree as only child and a *join* operator that takes a set of rooted trees with isomorphic root nodes as input and produces a rooted tree with the same root node and as its children all the children of the roots of the input trees. In this approach, all core rooted trees can be generated by extending/joining smaller core rooted trees, however, in rightmost path extension some core rooted trees can only be generated by extending non-cores (Dries and Nijssen, 2012). This motivates us to use the second approach for generating BRETDs. As we will discuss later, when the second approach is extended to patterns with a root larger than 1, in order to generate all cores, some non-cores should also be generated.

We start the section with defining the *extension* and *join* operators for BRETDs. While they can generate all BRETDs, the patterns they represent are not unique and post-processing is needed to eliminate redundant ones. So we introduce some pruning and conclude with an algorithm that is guaranteed to terminate and to generate all bounded-treewidth height-bounded core BRETDs.

Definition 7 (the extension operator)**.** *Let T_{P^X} be a BRETD of the pattern P^X. The extension operator applied on T_{P^X} constructs a set of new BRETDs. Each new BRETD consists of a new root node r that has T_{P^X} as single child BRETD. The vertices Y of the root node r are obtained by adding a new vertex v to a non-empty subset X' of X. The edges of the root node are the ones from the root node of T_{P^X} induced by the subset X' extended with new edges connecting v with a (non-empty) subset X'' of X'. Labeling functions μ_1 and μ_2 assign labels to the new vertex and edges from Σ_V and Σ_E respectively. By selecting at most tw vertices from X, it is ensured that the allowed treewidth is not exceeded. The resulting BRETD is parametrized by the choices for μ_1, X', X'', and μ_2 and is denoted as $\rho_{ext}(T_{P^X}, \mu_1, X', X'', \mu_2)$. The set of patterns extending P^X*

Figure 3.5: An example of the extension operator. On the left a pattern (bottom) and a tree decomposition of it (top); on the right an extension of the tree decomposition (top) and the pattern it represents (bottom).



is given by $\{Pat(\rho_{ext}(T_{P^X}, \mu_1, X', X'', \mu_2)) \mid X' \subseteq X, |X'| \leq tw, X'' \subseteq X', \mu_1$ *a function from* v *to* Σ_V, μ_2 *a function from* $\{(v, x'') \mid x'' \in X''\}$ *to* Σ_E.

Note that each valid combination of μ_1, X', X'' and μ_2 gives a different BRETD. A BRETD (rooted pattern) generated by the extension operator is called an *extended BRETD* (*extended rooted pattern*). Figure 3.5 presents an example of extending a BRETD T_{P^X} with the underlying rooted pattern P^X. In this example, the new vertex $v4$ is labeled by c and X' contains both vertices in X, i.e., $v2$ and $v3$. X'' contains both $v2$ and $v3$, that means $v4$ is connected to both $v2$ and $v3$. The function μ_2 maps both $\{v4, v2\}$ and $\{v4, v3\}$ to a.

Definition 8 (the join operator)**.** *Let* $T_{{P_1}^{X_1}}, \ldots, T_{{P_k}^{X_k}}$ *be BRETDs of patterns* ${P_1}^{X_1}$, ..., ${P_k}^{X_k}$ *respectively, such that:*

(i) For all i*:* $height(T_{{P_i}^{X_i}}) > 1$*, and*

(ii) $R({P_1}^{X_1}), \ldots, R({P_k}^{X_k})$ *are isomorphic.*

The join operator first renames the vertices of the participating BRETDs such that for all $i \neq j$: $V({P_i}^{X_i}) \cap V({P_j}^{X_j}) = \emptyset$. Then, a candidate BRETD is generated by selecting a set of mappings $\varphi_1, \ldots, \varphi_{k-1}$ such that φ_{i-1} is an isomorphism from ${P_i}^{X_i}$ to ${P_1}^{X_1}$. Each candidate consists of the root node of ${P_1}^{X_1}$ and its children are obtained by copying the children from $T_{{P_1}^{X_1}}, T_{{P_2}^{X_2}}, \ldots, T_{{P_k}^{X_k}}$ where each $T_{{P_{i+1}}^{X_{i+1}}}$ ($i >= 1$) is transformed by the isomorphism φ_i. The new BRETD is parametrized by the isomorphisms φ_1, ..., φ_{k-1} and is denoted as $\rho_{join}(T_{{P_1}^{X_1}}, \ldots, T_{{P_k}^{X_k}}, \varphi_1, \ldots, \varphi_{k-1})$. The set of BRETDs is denoted as $\rho_{join}({P_1}^{X_1}, \ldots, {P_k}^{X_k})$ and is given by $\{T \mid$ there exist isomorphisms $\varphi_1, \ldots, \varphi_{k-1}$ such that $T = \rho_{join}(T_{{P_1}^{X_1}}, \ldots, T_{{P_k}^{X_k}}, \varphi_1, \ldots, \varphi_{k-1})$ and T is a BRETD$\}$.

Note that condition (iii) of the definition of tree decomposition (Chapter 2) implies that the only vertices shared by different children of a node are vertices from that node and hence, the join operator does not need to rename non-root vertices to obtain all possible joins (up to renaming of the vertices). Each valid combination of $T_{{P_1}^{X_1}}$, ..., $T_{{P_k}^{X_k}}$, φ_1, ... and φ_{k-1} gives a different BRETD. A BRETD (rooted pattern) generated by the join operator is called a *joint BRETD* (*joint rooted pattern*). Figure 3.6 presents an example of the join operator.

Proposition 3. *Given BRETDs ${P_1}^{X_1}, \ldots, {P_k}^{X_k}$, for any $i \in 1, \ldots, k$:*

$$\rho_{join}({P_i}^{X_i}, \rho_{join}({P_1}^{X_1}, \ldots, {P_k}^{X_k}) = \emptyset$$

Proof. The generated candidate BRETDs are not BRETD because they violate either Condition (iv) or Condition (v) of Definition 6. Condition (iv) is violated when $V({P_i}^{X_i}) = X_i$; otherwise, Condition (v) is violated. □

Proposition 4. *We have:*

$$\begin{aligned}\rho_{join}({P_1}^{X_1}, \ldots, {P_k}^{X_k}) &= \rho_{join}(\rho_{join}({P_1}^{X_1}, \ldots, {P_{k-1}}^{X_{k-1}}), {P_k}^{X_k}) \\ &= \rho_{join}(\ldots \rho_{join}(\rho_{join}({P_1}^{X_1}, {P_2}^{X_2}), {P_3}^{X_3}), \ldots, {P_k}^{X_k})\end{aligned}$$

Proof. It follows directly from the definition of the join operator. □

One can observe here that it suffices to do a binary join between every single child BRETD and every BRETD derived before.

Proposition 5. *Given all BRETDs of height 1, all BRETDs of width tw (≥ 1) and height h (> 1) can be derived by a finite number of applications of join and extension operators where each join is a binary one with one of the inputs a single child BRETD.*

Proof. The proof is done by induction. We prove that given BRETDs of width tw and height h, all BRETDs of width tw and height $h+1$ are derived by a finite number of applications of join and extension operators where each join is a binary one with one of the inputs a single child BRETD. Every BRETD T_{P^X} of height $h+1$ falls into one of the following categories:

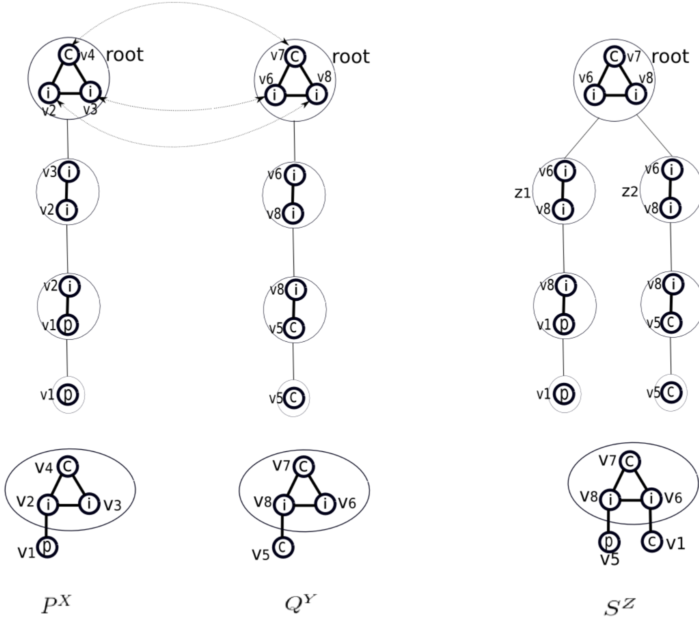
- $root(T_{P^X})$ has one child. In this case, T_{P^X} is generated by extending a BRETD T_{Q^Y}, with underlying rooted pattern Q^Y, of height h. P^X has one more vertex than Q^Y which is connected to some vertices in X. Since during extending BRETDs of height h, all possible vertex labels and all possible sets X' and X'' and mappings μ_2 (as defined in Definition 7) are considered, T_{P^X} will be generated.
- $root(T_{P^X})$ has $k > 1$ children. In this case, $root(T_{P^X})$ is generated by the join operator. Let $C_1, C_2, \ldots, C_k$ be the children of $root(T_{P^X})$. For $1 \leq i \leq k$, consider single child BRETDs $T_{P_i{}^{X_i}}$ consisting of $root(T_{P^X})$ and the subtree of T_{P^X} rooted at C_i. For any $T_{P_i{}^{X_i}}$, $root(T_{P_i{}^{X_i}})$ has only one child, therefore, all $T_{P_i{}^{X_i}}$ are generated by extension. T_{P^X} can be generated by $\rho_{join}(T_{P_1{}^{X_1}}, \ldots, T_{P_k{}^{X_k}})$ and hence as Proposition 4 says, by $\rho_{join}(\ldots \rho_{join}(\rho_{join}(P_1{}^{X_1}, P_2{}^{X_2}), P_3{}^{X_3}), \ldots, P_k{}^{X_k})$.

□

Algorithm 1 shows the high level pseudo code of our pattern generation method. Although every core pattern has a BRETD, there are patterns that can only be generated from non-core patterns. As a simple example, assume that the database graph is the undirected path d − a − b − a − c. Then, the BRETD with the underlying undirected pattern d − a − b* − a* (rooted at the vertices indicated by $*$) is not core, however, the BRETD with the underlying pattern d − a − b − a* − c* (rooted at the vertices indicated by $*$) is core and it can only be generated by extending a non-core. So our algorithm generates also non core patterns. For a similar reason (i.e., a frequent pattern may only be generated by extending/joining non-frequent ones), the algorithm also generates infrequent patterns. However, if the frequency of a rooted pattern is 0, none of the patterns generated by its extensions/joins can be frequent. Therefore, it can be pruned. In the algorithm, C_{cur} (resp. C_{next}) is intended to contain all BRETDs of height h (resp. $h+1$) with a distinct canonical form and C all BRETDs with a distinct canonical form and a height less or equal to $maxHeight$. $CanForms$ holds the canonical forms of all BRETDs found so far (in a trie).

The algorithm performs a level-wise search where BRETDs of height h are extended and joined to generate larger BRETDs. First, by one pass over the database graph H, Σ_V and Σ_E are extracted. From Σ_V, BRETDs of height 1, i.e., $\mathcal{C}_1$, are generated: for every vertex label μ_1 in Σ_V, a BRETD T_{P^X} is generated where T_{P^X} consists of one

Figure 3.6: An example of joining two BRETDs T_{P^X} and T_{Q^Y}, that have respectively the underlying rooted patterns P^X and Q^Y. The isomorphism mapping between $R(P^X)$ and $R(Q^Y)$ is depicted by arrows. It maps v_4 to v_7, v_2 to v_8, and v_3 to v_6. In the generated rooted pattern S^Z, v_3 is replaced by v_6 and v_2 is replaced by v_8.



single node that has only one vertex labeled by μ_1. These patterns are extended/joined to generate larger patterns, using the EXTEND and JOIN methods. For every $1 \leq h \leq maxHeight$, first the extended BRETDs of height $h+1$ are generated by extending BRETDs of height h (Line 10 of Algorithm 1). Then, every extended BRETD of height $h+1$ joins with already generated BRETDs to generate the joint BRETDs of height $h+1$ (Line 14 of Algorithm 1).

We store canonical forms of already generated rooted patterns in a trie, called *CanForms*. Then, when a new BRETD T_{P^X} with the underlying pattern P^X is generated, the canonical form of P^X is computed and it is checked if *CanForms* already contains this canonical form. If so, this means P^X is already generated (by some other BRETD), hence, it is discarded. Otherwise, the canonical form of P^X is added to the trie. The reason for using a trie for storing canonical forms is that looking up data in a trie is done very fast in $O(m)$ time in the worst case, where m is the length of the search string.

Algorithm 2 describes how a BRETD T is extended. In every extension of it, the new

Algorithm 1 High level pseudo code of the pattern generation algorithm.

1: PATTERNGENERATOR
2: **Input.** a database graph H, an integer tw, and an integer $maxHeight$.
3: **Output.** the set of BRETDs with treewidth bounded by tw and height bounded by $maxHeight$.
4: $CanForms \leftarrow \emptyset$ {$CanForms$ stores the canonical forms of the generated rooted patterns.}
5: $\Sigma_V \leftarrow$ extract vertex alphabet from H
6: $\Sigma_E \leftarrow$ extract edge alphabet from H
7: $\mathcal{C}_1 \leftarrow$ BRETDs of height 1
8: $h \leftarrow 1$, $\mathcal{C} \leftarrow \mathcal{C}_1$ {all BRETDs of height up to h}, $\mathcal{C}_{cur} \leftarrow \mathcal{C}_1$ {all BRETDs of height h}, $\mathcal{C}_{next} \leftarrow \emptyset$ {all BRETDs of height $h+1$}
9: **while** $h < maxHeight$ **do**
10: $C' \leftarrow \{\text{EXTEND}(T, tw, CanForms) \mid T \in C_{cur}\}$
11: **for all** $T \in C'$ **do**
12: $C'' \leftarrow \emptyset$
13: **if** $Pat(T) \preceq^{hm} H$ **then**
14: $C'' \leftarrow \{\text{JOIN}(T, T') \mid T' \in C \cup C_{next}\}$
15: Add T to $\mathcal{C}_{next}$
16: **end if**
17: **for all** $T'' \in C''$ **do**
18: **if** $Pat(T'') \preceq^{hm} H$ **then**
19: Add T'' to $\mathcal{C}_{next}$
20: **end if**
21: **end for**
22: **end for**
23: $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_{next}$
24: $\mathcal{C}_{cur} \leftarrow \mathcal{C}_{next}$
25: $\mathcal{C}_{next} \leftarrow \emptyset$
26: $h \leftarrow h + 1$
27: **end while**
28: **return** $\mathcal{C}$

root node consists of a new vertex x, a subgraph of the root node of T with at most tw vertices, so that the new node does not violate the width constraint and x is connected to the vertices in a non-empty subset of the root of the extended pattern.

Algorithm 3 shows how two BRETDs T_1 and T_2 join. For every isomorphism mapping φ, a new candidate joint BRETD $\rho_{join}(T_1, T_2, \varphi)$ is generated. Every isomorphism mapping may give a different rooted pattern. The isomorphism mapping makes the mapped vertices identical.

Algorithm 2 High level pseudo code of extending a BRETD.

1: EXTEND
2: **Input.** T: a BRETD, tw: an integer, $CanForms$: stores the canonical forms of the generated patterns.
3: **Output.** $\mathcal{C}$: BRETDs generated by extending T.
4: **Side effect.** The canonical forms of the new patterns are inserted into $CanForms$.
5: $\mathcal{C} \leftarrow \emptyset$
6: **for all** vertex labels μ_1 **do**
7:    **for all** $X' \subseteq X$ such that $|X'| \leq tw$ and $X' \neq \emptyset$ **do**
8:       **for all** $X'' \subseteq X'$ such that $X'' \neq \emptyset$ **do**
9:          **for all** mappings μ_2 that map every element of X'' to Σ_E **do**
10:             $T' \leftarrow \rho_{ext}(T, \mu_1, X', X'', \mu_2)$
11:             $Str \leftarrow$ the canonical form of $Pat(T')$
12:             **if** $Str \notin CanForms$ **then**
13:                Add T' to $\mathcal{C}$
14:                Insert Str into $CanForms$
15:             **end if**
16:          **end for**
17:       **end for**
18:    **end for**
19: **end for**
20: **return** $\mathcal{C}$

Corollary 1. *The set of bounded-treewidth height-bounded cores is finite.*

Proof. It follows directly from Proposition 5. □

3.4 Mining frequent rooted patterns

In this section, we present our proposed algorithm for finding frequent rooted patterns. We first introduce a number of notions.

Algorithm 3 High level pseudo code of joining BRETDs.

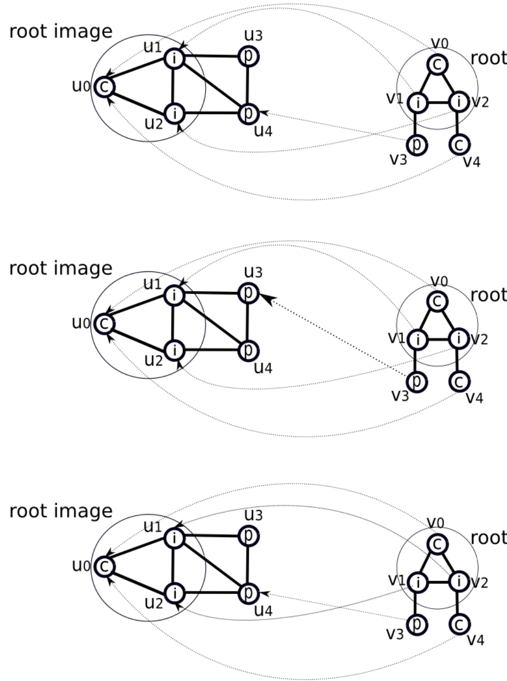
1: JOIN
2: **Input.** T_1 and T_2: two BRETDs, $CanForms$: stores the canonical forms of the generated patterns.
3: **Output.** $\mathcal{C}$: BRETDs generated by the join of T_1 and T_2.
4: **Side effect.** The canonical forms of the new patterns are inserted into $CanForms$.
5: $\mathcal{C} \leftarrow \emptyset$
6: **if** $R(Pat(T_1))$ and $R(Pat(T_2))$ are isomorphic and they have a height greater than 1 **then**
7: Rename the vertices of $Pat(T_2)$ such that $V(Pat({P_1}^{X_1})) \cap V(Pat({P_2}^{X_2})) = \emptyset$
8: **for all** isomorphism mappings φ between $R(Pat(T_1))$ and $R(Pat(T_2))$ **do**
9: $T \leftarrow \rho_{join}(T_1, T_2, \varphi)$
10: **if** T is a BRETD **then**
11: $Str \leftarrow$ the canonical form of $Pat(T)$
12: **if** $Str \notin CanForms$ **then**
13: Add T to $\mathcal{C}$
14: Insert Str into $CanForms$
15: **end if**
16: **end if**
17: **end for**
18: **end if**
19: **return** $\mathcal{C}$

Definition 9 (root embeddings)**.** *The set of root embeddings of a rooted pattern P^X in a database graph H is defined as follows:*

$$RE(P^X, H) = \{\varphi_{|X} \mid \varphi \text{ is a subgraph homomorphism mapping from } P \text{ to } H\} \tag{3.1}$$

Figure 3.7 shows an example of root embeddings, where the graph on the left is a database graph presenting a toy social network. It has 3 types of vertices: c refers to a country, i refers to an individual, and p refers to a page. Every vertex is labeled by its type. An individual might be in different countries, so a vertex of type i might be connected to several vertices of type c. If two individuals are friends, an edge is drawn between them. An edge is drawn between a vertex of type i and a vertex of type p if the individual (vertex of type i) likes the page (vertex of type p). If two pages (vertices of type p) have links to each other, an edge is drawn between them. The graph on the right side is a rooted pattern. The rooted pattern has three embeddings in the database

Figure 3.7: An example of root embedding. The graph on the left is a database graph. The graph on the right is a rooted pattern that has three embeddings in the database graph. The first and second embeddings have the same root embeddings, that is $\{u_0, u_1, u_2\}$. The third one has a different root embedding $\{u_0, u_2, u_1\}$. Therefore, the frequency of the rooted pattern is 2.



graph, where two of them, the first and second ones, have the same root embeddings. The third one has a different root embedding.

In order to precisely define a frequent pattern mining problem, a support measure (also called frequency measure) is required. Several frequency measures have been proposed in the literature for single graph mining (Bringmann and Nijssen, 2007; Calders et al., 2011; Dries and Nijssen, 2012; Wang and Ramon, 2012). Our mining method is general and can work with any support measure. However, for the sake of simplicity, we here restrict ourselves to the frequency measure computed as the number of embeddings of the root of a rooted pattern (Bringmann and Nijssen, 2007; Dries and Nijssen, 2012).

Definition 10 (support (frequency) of a rooted pattern)**.** *Let P^X be a rooted pattern and H be a database graph. Root embedding support (Root embedding frequency) of*

P^X in H, denoted by $sup_{RE}(P^X, H)$ is defined as:

$$sup_{RE}(P^X, H) = |RE(P^X, H)| \tag{3.2}$$

P^X is frequent iff its root embedding support is more than (or equal to) a user-defined threshold $minsup$.

Note that this notion of frequency is not *anti-monotonic*, i.e., the frequency of a rooted pattern may be less than the frequency of its super patterns. As a simple example, suppose that the database graph is the path a-b-a and $minsup$ is 2. Then, while the pattern consisting of a single vertex b is not frequent, the rooted pattern a-b rooted at a is frequent and it can only be generated by extending an infrequent pattern. We may develop extensions of $sup_{RE}(P^X, H)$ that are anti-monotonic. Let T_{P^X} be a BRETD of P^X and $T_1, \ldots, T_c$ the subtrees of T_{P^X} rooted at the children of $root(T_{P^X})$. Then $sup^*_{RE}(P^X, H, T_{P^X})$ defined as:

$$\min\{sup_{RE}(P^X, H), sup_{RE}(Pat(T_1), H), \ldots, sup_{RE}(Pat(T_c), H)\}$$

is an anti-monotonic support measure.

Our goal in this chapter is to find frequent (under subgraph homomorphism) rooted patterns that are core[3]. More formally, given:

- a graph (network) H,
- a user defined integer $minsup$,
- a user defined maximum treewidth tw, and
- a user defined maximum height $maxLevel$,

we want to find *tw-bounded-treewidth cores* that have a frequency higher than (or equal to) $minsup$ and a BRETD with height bound $maxLevel$.

To address this problem, we propose the HoPa algorithm. Algorithm 4 shows the high level pseudo code of HoPa. In this algorithm, $\mathcal{F}$ contains all *bounded-treewidth height-bounded cores* that are frequent. The algorithm first calls Algorithm 1 to generate candidate rooted patterns that have a height at most $maxLevel$. Then, for every generated BRETD T_{P^X} with underlying rooted pattern P^X, it checks whether P^X is a core and if so, root embeddings of P^X are computed and its frequency is counted. If P^X is frequent and core, it is stored as a frequent pattern (it is added to $\mathcal{F}$).

[3]Our proposed mining algorithm can work with both sup_{RE} and sup^*_{RE}. Our focus in Sections 3.4 and 3.5 is sup_{RE}, however in order to have an empirically efficient algorithm, in the experiments reported in Section 3.6 we use sup^*_{RE}.

If a graph G is not core, there exists a graph G_{-v}, generated by removing a vertex v and its incident edges from G, such that G is subgraph homomorphic to G_{-v}. Hence, to check if a rooted pattern P^X is core, we act as follows: by removing each vertex v of P^X (and its incident edges) a new graph P_{-v} is generated. If for at least one v, P is subgraph homomorphic to P_{-v}, P^X is not core; otherwise, it is core. For example in Figure 3.2, only ${P_1}^{X_1}$ is a core.

For enumerating root embeddings of a rooted pattern, we use an underlying querying system that takes as input a rooted tree decomposition and a database graph and computes the set of all root embeddings of the tree decomposition in the database graph (Fannes et al., 2015).

Algorithm 4 High level pseudo code of the pattern mining algorithm

1: HoPa
2: **Input.** a database graph H, an integer tw, an integer $maxLevel$ and an integer $minsup$.
3: **Output.** the set of tw-bounded-treewidth height-$maxLevel$-bounded frequent rooted patterns that are core
4: $\mathcal{F} \leftarrow \emptyset$
5: $\mathcal{C} \leftarrow$ PatternGenerator$(H, tw, maxLevel)$
6: **for all** $T_{P^X} \in \mathcal{C}$ **do**
7: {let P^X be the underlying rooted pattern of T_{P^X}}
8: **if** P^X is core **then**
9: $re \leftarrow$ compute $RE(P^X, H)$
10: **if** $|re| \geq minsup$ **then**
11: Add P^X to $\mathcal{F}$
12: **end if**
13: **end if**
14: **end for**
15: **return** $\mathcal{F}$

Optimization techniques. We discuss some optimization techniques that can be used to improve the efficiency of our proposed algorithm.

- **Infrequent parent check.** Let P^X and Q^Y be two rooted patterns such that Q^Y is root subgraph homomorphic to P^X and $R(Q^Y)$ is isomorphic to $R(P^X)$. If Q^Y is infrequent, then P^X will be infrequent, too. We use this property to reduce the number of patterns that are generated but are potentially infrequent. In this technique, called *infrequent parent check*, a set $\mathcal{I}$ of infrequent patterns is kept. Then, when a new rooted pattern P^X is generated, before counting its frequency, it is checked whether there exists any rooted pattern $Q^Y \in \mathcal{I}$

such that Q^Y is root subgraph homomorphic to P^X and $R(Q^Y)$ is isomorphic to $R(P^X)$. If Q^Y exists, P^X cannot be frequent, hence, its frequency is not counted. This technique can significantly improve the efficiency of the mining algorithm by reducing the number of rooted patterns that are tested for frequency.

- **Partitioning patterns into equivalence classes.** One way to find out what BRETDs are joinable with a given BRETD T_{P^X} is to check every already generated BRETD T_{Q^Y} and see if they have isomorphic roots. However, this can decrease the efficiency of the algorithm, as only a small percentage of the already generated BRETDs may be joinable with T_{P^X}. A more effective technique is to partition all generated BRETDs into equivalence classes so that every two BRETDs T_{P^X} and T_{Q^Y} are in the same equivalence class iff they have isomorphic roots. In this way, every BRETD is joined with all members of its equivalence class. When a new BRETD is generated, it is added to the appropriate equivalence class (or a new class is created).

3.5 Condensed representations of frequent patterns

The set of all frequent rooted patterns can be huge so that it becomes difficult to use and interpret them. Therefore, in many applications, it may be desirable to have a condensed representation of all frequent patterns. Mining closed patterns is one of the most common techniques used for compact representation of frequent patterns.

A problem with widely used closedness data structures for single network mining is that they are based on a *pseudo closure operator* rather than a *closure operator*. For example, consider the frequency based closedness data structure. The used operator maps a pattern to its maximal super pattern that have the same frequency as the pattern. Now, assume that the pattern class is the class of rooted trees and the frequency measure is the number of root embeddings. Then in the database graph H of Figure 3.8, patterns $P1$, $P2$ and $P3$ have the same frequency 3. Furthermore, $P2$ and $P3$ are the maximal super patterns of $P1$ that have frequency 3. As $P1$ has no unique maximal supper pattern, choosing an arbitrary one as its closure violates condition C2 for the other one. This means the frequency based closedness data structure is not a closure operator. It motivates us to present in Definition 13 a new closedness data structure. This data structure is based on the concept of *root embedding equivalence class*, defined in Definition 11. We later prove that it gives a closure operator for single network mining under homomorphism.

Definition 11 (equivalent under root embeddings (root embedding equivalence class))**.** *Let P^X and Q^Y be two rooted patterns. Given a database graph H, P^X and Q^Y are*

Figure 3.8: Let H be a database graph. $P2$ and $P3$ are the maximal super patterns of $P1$ that have the same frequency as $P1$. As $P1$ has no unique maximal supper pattern, choosing an arbitrary one as its closure violates condition C2 for the other one. As a result, the frequency based closedness data structure is not a closure operator.

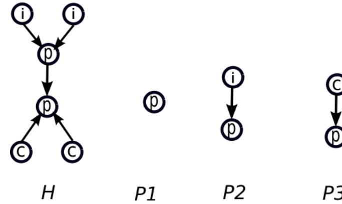
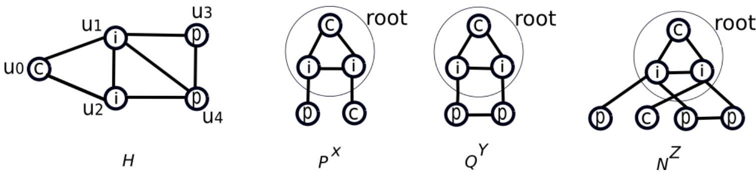


Figure 3.9: Rooted patterns P^X and Q^Y are in the same root embedding equivalence class. H is the database graph and N^Z is generated by a merge of P^X and Q^Y.



equivalent under root embeddings for H (they belong to the same root embedding equivalence class for H), denoted by $P^X \equiv_{RE} Q^Y$, iff:

1. *$R(P^X)$ and $R(Q^Y)$ are isomorphic,*
2. *$|RE(P^X, H)| = |RE(Q^Y, H)|$, and*
3. *There is a root embedding φ_P with image I of P^X in H iff there is a root embedding φ_Q with the same image I of Q^Y in H.*

For example, rooted patterns P^X and Q^Y presented in Figure 3.9 belong to the same root embedding equivalence class.

Before presenting our closure operator, we need to introduce the *merge* operator.

Definition 12 (the merge operator)**.** *The merge of a set $\{T_1, \ldots, T_n\}$ ($n > 1$) of BRETDs, denoted as $\rho_{mrg}(T_1, \ldots, T_n)$, is defined as the join of $T_1, \ldots, T_n$ for a set of isomorphism mappings between $R(Pat(T_1))$ and $R(Pat(T_2)), \ldots, R(Pat(T_n))$ that gives the pattern with the lexicographically smallest canonical form. The merge of a single BRETD T_1, denoted as $\rho_{mrg}(T_1)$, is defined as T_1 itself.*

A tree decomposition generated by merge is a candidate BRETD. For example, in Figure 3.9, the candidate BRETD with the underlying pattern N^Z is the merge of the

BRETD with the underlying pattern P^X and the BRETD with the underlying pattern Q^Y. Note that in merge the order of BRETDs is not important, i.e., $\rho_{mrg}(T_1, T_2)$ is the same as $\rho_{mrg}(T_2, T_1)$ up to root isomorphism. It is possible that in a merge more than one set of isomorphism mappings generate the pattern with the smallest canonical form. However, doing merge with any of them yields the same pattern up to root isomorphism.

Proposition 6. *Let $T_1, \ldots, T_n$ be a set of BRETDs such that $R(Pat(T_1)), \ldots, R(Pat(T_n))$ are isomorphic. We have*

- *$Pat(T_i) \preceq^{hm} Pat(\rho_{mrg}(T_1, \ldots, T_n))$ for $1 \leq i \leq n$, and*
- *If $Pat(T_i) \preceq^{hm} Q^Y$ for all i's, then $Pat(\rho_{mrg}(T_1, \ldots, T_n)) \preceq^{hm} Q^Y$.*

Proof. It follows directly from Definition 12. □

Definition 13 (the σ_{RE} closure operator)**.** *Let $\mathcal{P}$ be a set of bounded-treewidth height-bounded cores that are partitioned into root embedding equivalence classes $eq_1, \ldots, eq_l$ for a database H; and $\mathcal{P}' = \cup_{i=1}^{l}\{Pat(\rho_{mrg}(eq_i))\}$. For the pattern class $\mathcal{P} \cup \mathcal{P}'$ and partial order* root subgraph homomorphism*, the closure operator σ_{RE} is defined as follows:*

- *For each $P^X \in \mathcal{P}'$, σ_{RE} maps P^X to P^X.*
- *For each $P^X \in \mathcal{P}$, let $\{T_1, \ldots, T_n\}$ denote the root embedding equivalence class of P^X for H. Then, σ_{RE} maps P^X to $Pat(\rho_{mrg}(T_1, \ldots, T_n))$.*

Note that in Definition 13 members of the equivalence classes are merged only once, hence, since there are a finite number of finite equivalence classes, $\mathcal{P}'$ is finite, too. In Proposition 7, we prove that σ_{RE} is a closure operator.

Proposition 7. *The operator σ_{RE} introduced in Definition 13 is a closure operator.*

Proof. To be a closure operator, σ_{RE} must be *extensive*, *increasing* and *idempotent*.

- *Extensive*: We need to prove that for every rooted pattern P^X, $P^X \preceq^{hm} \sigma_{RE}(P^X)$. It follows directly from Proposition 6.
- *Increasing*: We need to prove that for every two rooted patterns $P^X, Q^Y \in \mathcal{P} \cup \mathcal{P}'$, $P^X \preceq^{hm} Q^Y$ implies $\sigma_{RE}(P^X) \preceq^{hm} \sigma_{RE}(Q^Y)$. If $P^X \in \mathcal{P}'$, by definition $\sigma_{RE}(P^X) = P^X$; hence $P^X \preceq^{hm} Q^Y$ implies $\sigma_{RE}(P^X) \preceq^{hm} Q^Y \preceq^{hm} \sigma_{RE}(Q^Y)$. Now, suppose $P^X \in \mathcal{P}$ and let $eq = \{T_1, T_2, \ldots, T_n\}$ be

the root embedding equivalence class to which the BRETD of P^X belongs. We need to prove

$$\sigma_{RE}(P^X) = Pat(\rho_{mrg}(T_1, \ldots, T_n)) \preceq^{hm} \sigma_{RE}(Q^Y)$$

The proof is by induction on the size n of the root embedding equivalence class of P^X. Without loss of generality, let P^X be $Pat(T_1)$.

Base case ($n = 1$): In this case, $Pat(\rho_{mrg}(T_1))$ is P^X. Since P^X is root subgraph homomorphic to Q^Y and Q^Y is root subgraph homomorphic to $\sigma_{RE}(Q^Y)$, P^X is root subgraph homomorphic to $\sigma_{RE}(Q^Y)$.

Induction step: Assume

$$Pat(\rho_{mrg}(T_1, \ldots, T_{i-1})) \preceq^{hm} \sigma_{RE}(Q^Y) \tag{3.3}$$

We prove

$$Pat(\rho_{mrg}(T_1, \ldots, T_{i-1}, T_i)) \preceq^{hm} \sigma_{RE}(Q^Y) \tag{3.4}$$

To do so, we prove

$$Pat(T_i) \preceq^{hm} \sigma_{RE}(Q^Y) \tag{3.5}$$

Then using Proposition 6, Relations 3.3 and 3.5 yield Relation 3.4.

To show that Relation 3.5 holds, in the following, (i) we construct a rooted pattern ${Q'}^{Y'}$ such that $Pat(T_i) \preceq^{hm} {Q'}^{Y'}$, and (ii) we show that ${Q'}^{Y'}$ and Q^Y are in the same root embedding equivalence class. ${Q'}^{Y'}$ is not necessarily a core, however, since our pattern generation algorithm produces all height-bounded bounded-treewidth rooted cores, there is a rooted core ${Q''}^{Y''}$ such that ${Q'}^{Y'} \cong^{hm} {Q''}^{Y''}$ and ${Q'}^{Y'}$ (also Q^Y) have the same set of root embeddings as ${Q''}^{Y''}$. Therefore, by definition, ${Q''}^{Y''} \preceq^{hm} \sigma_{RE}(Q^Y)$ which yields ${Q'}^{Y'} \preceq^{hm} \sigma_{RE}(Q^Y)$ and consequently, $Pat(T_i) \preceq^{hm} \sigma_{RE}(Q^Y)$.

(i) First, we introduce rooted pattern ${Q'}^{Y'}$. Let φ_1 be a canonical isomorphism mapping from $R(Pat(T_i))$ to $R(Pat(T_1))$ and φ_2 a root subgraph homomorphism mapping from $Pat(T_1)$ to Q^Y. The mapping $\varphi_2 \circ \varphi_1$ yields a subgraph homomorphism mapping from $R(Pat(T_i))$ to $R(Q^Y)$. The rooted pattern ${Q'}^{Y'}$ is constructed as follows: first, vertices of $Pat(T_i)$ are renamed such that $V(Pat(T_i)) \cap V(Q^Y) = \emptyset$; then ${Q'}^{Y'}$ is initialized by Q^Y; finally $Pat(T_i)$ is added to ${Q'}^{Y'}$ so that every root vertex v of $Pat(T_i)$ is transformed to $\varphi_2 \circ \varphi_1(v)$ and non-root vertices of $Pat(T_i)$ are added to ${Q'}^{Y'}$ as new vertices. We have:

$$Pat(T_i) \preceq^{hm} {Q'}^{Y'}$$

(ii) Now, we show that ${Q'}^{Y'}$ and Q^Y have the same set of root embeddings. Since T_1 and T_i are in the same root embedding equivalence class and for each embedding φ_Q of Q^Y in H there exists an embedding φ_{T_1} of $Pat(T_1)$ in H such that φ_{T_1} and φ_Q have the embeddings of the root vertices of $Pat(T_1)$ in common, the same holds for T_i, i.e., for each embedding φ_Q of Q^Y in H there exists an embedding φ_{T_i} of $Pat(T_i)$ in H such that φ_{T_i} and φ_Q have the embeddings of the root vertices of $Pat(T_i)$ in common. As a result, for each embedding φ_Q of Q^Y in H, there exists an embedding $\varphi_{Q'}$ of ${Q'}^{Y'}$ in H such that φ_Q and $\varphi_{Q'}$ have the embeddings of the root vertices of Q^Y (the root vertices of ${Q'}^{Y'}$) in common.

Conclusion step: By the principle of induction, Relation 3.4 is true for all i, $1 \leq i \leq n$.

- *Idempotent*: we need to prove that for every rooted pattern P^X, $\sigma_{RE}(P^X) = \sigma_{RE}(\sigma_{RE}(P^X))$.
  - If $P^X \in \mathcal{P}'$, by definition, $\sigma_{RE}(\sigma_{RE}(P^X)) = \sigma_{RE}(P^X) = P^X$.
  - If $P^X \in \mathcal{P}$, by definition, $\sigma_{RE}(\sigma_{RE}(P^X)) = \sigma_{RE}({P'}^{X'})$, where ${P'}^{X'}$ is the merge of the patterns in the root embedding equivalence class of P^X. Furthermore by definition, $\sigma_{RE}({P'}^{X'}) = {P'}^{X'}$. Hence, $\sigma_{RE}(\sigma_{RE}(P^X)) = {P'}^{X'} = \sigma_{RE}(P^X)$.

□

The closed pattern of a root embedding equivalence class gives a *complex representation* of all members of the class in the sense that all the members of the class are subgraph homomorphic to its closed pattern. We may also be interested in having a *simple representation* of the root embedding equivalence class. This motivates us to introduce in Definition 15 the notion of the *simplest member* of a root embedding equivalence class.

Definition 14 (simpler pattern)**.** *A rooted pattern P^X is simpler than another rooted pattern Q^Y, denoted by $P^X <_{simp} Q^y$, iff:*

- $treewidth(P^X) < treewidth(Q^Y)$*, or*
- $treewidth(P^X) = treewidth(Q^Y)$ *and* $|V(P^X)| + |E(P^X)| < |V(Q^Y)| + |E(Q^Y)|$.

We say $P^X \leq_{simp} Q^Y$ if in the above mentioned conditions, '$<$' is replaced by '$\leq$'.

Definition 15 (simplest pattern)**.** *A simplest member of a root embedding equivalence class eq is a rooted pattern $P^X \in eq$ so that there is no other rooted pattern $Q^Y \in eq$ which is simpler than P^X.*

Note that $\leq_{simp}$ does not apply a total ordering on the members of a root embedding equivalence class and as a result, several members of the equivalence class may satisfy the conditions of the simplest member. When during a mining process we want to find the simplest member of a root embedding equivalence class, among all members satisfying the conditions, we may choose an arbitrary one, e.g., the one discovered earlier by the mining algorithm.

In finding root embedding equivalence classes (and distinguishing their closed and simplest members), an elementary task is to check if two rooted patterns have the same set of root embeddings. Comparing two sets of root embeddings can be time and space consuming as patterns may have many root embeddings. A more effective way is to hash the set of root embeddings into keys and compare the keys, instead of the sets. It is based on the assumption that if two rooted patterns have the same key, they will have the same set of root embeddings, and vice versa. In Definition 16, we propose a method for hashing a set of root embeddings.

Definition 16 (root embeddings hash key)**.** *Let p and q be two prime numbers, P^X a rooted pattern and H a database graph. Suppose that vertices in X (and as a result, elements of every root embedding of P^X in H) are sorted by a canonical order, i.e., an order that yields the canonical form of $R(P^X)$. Root embeddings hash key of P^X in H, denoted by $hkey(P^X, H)$, is defined as follows*

$$hkey(P^X, H) = \left(\sum_{\varphi \in RE(P^X, H)} q^{\sum_{v \in X} id(\varphi(v)) \times p^{id(v)}} \right) \mod 2^{64} \tag{3.6}$$

where id is a function mapping a vertex to its (unique integer) index.

As an example, consider Figure 3.7 and suppose vertices u_0, u_1 and u_2 have vertexIds 0, 1 and 2, respectively. The rooted pattern has two root embeddings in the database graph. Suppose the root vertices are sorted as $u_0 < u_1 < u_2$. Their root embeddings hash key is $q^{0p^0+1p^1+2p^2} + q^{0p^0+2p^1+1p^2}$, where setting p to 3 and q to 2 gives $2^{21} + 2^{15}$.

Algorithm 5 shows the high level pseudo code of the algorithm of finding frequent patterns, frequent closed pattern and frequent simplest patterns, called CHoPa. The algorithm keeps a hash table HT that maps a root embeddings hash key $rekey$ to the root embedding equivalence class whose root embeddings are hashed to $rekey$.

Algorithm 5 High level pseudo code of the algorithm of finding closed and simplest patterns.

1: CHOPA
2: **Input.** a database graph H, an integer tw, an integer $maxLevel$ and an integer $minsup$.
3: **Output.** the set of frequent tw-bounded-treewidth, height-$maxLevel$-bounded cores, the set of closed patterns and the set of simplest patterns
4: {Let HT be a hash table that maps a root embeddings hash key $rekey$ to the root embedding equivalence class whose root embeddings are hashed to $rekey$.}
5: $HT \leftarrow \emptyset$
6: $\mathcal{F} \leftarrow \emptyset$
7: $\mathcal{C} \leftarrow$ PATTERNGENERATOR$(H, tw, maxLevel)$
8: **for all** $T_{P^X} \in \mathcal{C}$ **do**
9:     {Let P^X be the underlying rooted pattern of T_{P^X}}
10:     **if** P^X is core **then**
11:         $re \leftarrow$ compute $RE(P^X, H)$
12:         **if** $|re| \geq minsup$ **then**
13:             $rekey \leftarrow hkey(re)$
14:             **if** $rekey$ is already added to HT **then**
15:                 Add T_{P^X} to $HT[rekey]$
16:                 Update the closed pattern of $HT[rekey]$
17:                 **if** P^X is simpler than the current simplest member of $HT[rekey]$ **then**
18:                     Set simplest member of $HT[rekey]$ to P^X
19:                 **end if**
20:             **else**
21:                 Generate a new root embedding equivalence class eq and add T_{P^X} to it
22:                 Add the pair $(rekey, eq)$ to HT
23:                 Set the closed pattern of $HT[rekey]$ to P^X
24:                 Set the simplest member of $HT[rekey]$ to P^X
25:             **end if**
26:         **end if**
27:     **end if**
28: **end for**
29: **return** $\mathcal{F}$ and closed and simplest members of root embedding equivalence classes.

Then, the *closed* and *simplest* members of root embedding equivalence classes are distinguished. First, Algorithm 1 is called to generate candidate rooted patterns that have a height at most $maxLevel$. Then, for every generated BRETD T_{P^X} with underlying rooted pattern P^X, its root embeddings are computed, its frequency is counted, and its root embeddings hash key is calculated. The root embeddings hash key is stored in $rekey$. If the pattern is frequent and core, it is added to the set $\mathcal{F}$. The root embedding equivalence class related to $rekey$ is referred by $HT[rekey]$. If $HT[rekey]$ already exists, it is checked whether P^X can be the simplest member of the equivalence class. Otherwise, a new *root embedding equivalence class* is generated and its closed and simplest members are set to P^X.

3.6 Experimental results

In this section, we empirically evaluate the efficiency of our proposed algorithms on both real-world and synthetic data. In our experiments, HoPa and CHoPa refer to the respective algorithms. Note that CHoPa generates the same number of simplest and closed patterns, therefore, we report only the number of one of them.

3.6.1 Experimental setup

For our experiments, we are interested in answering the following experimental questions:

Q1. For listing frequent rooted trees, is HoPa more efficient than the methods with similar output, e.g., the htreeminer algorithm (Dries and Nijssen, 2012)?

Q2. For the small values of tw, what size of networks, what values of $maxLevel$ and what values of $minsup$ can our algorithm handle within reasonable time? How does the ratio

$$R1 = \frac{\text{the number of closed patterns}}{\text{the number of frequent patterns discovered for finding closed patterns}}$$

change by changing $minsup$ and $maxLevel$ for different networks?

Q3. What is the influence of the parameters $minsup$ and $maxLevel$ on the efficiency and output size of HoPa?

The experiments were performed on a AMD processor with 16 GB main memory and 2×1 MB $L2$ cache. Our program was compiled using the GNU C++ compiler $4.8.4$. In our experiments, HoPa and CHoPa have almost the same running time, therefore,

we report a single time for both of them. However, they produce different sets of patterns, hence, we distinguish the number of patterns they generate.

3.6.2 Datasets

To assess the efficiency of the proposed method, we use synthetic datasets as well as real-world networks. For synthetic data we generated networks of size $n \in \{10^6, 10^7\}$ according to a power-law model with degree distribution $P(d) \propto d^{-3}$. We assigned randomly one of a set of four labels to each of the vertices. Such graphs show significant clustering, as is often seen in real-world data. We refer to these datasets as $BA10^6$ and $BA10^7$, respectively. For real-world data, we use the Facebook social network[4] and the IMDB movie database[5].

Facebook The Facebook data is the Facebook friendship network obtained by two sampling methods: one by uniform sampling (called facebook-uniform), and the other by independent Metropolis-Hastings random walks (called facebook-mhrw) (Gjoka et al., 2010). For each method, the data consists of two files: one is the friendship network represented by an adjacency list; and the other contains extra properties such as total numbers of friends and privacy settings. Here, as was done by Kibriya and Ramon (2012), the two files of raw data are merged into a single network. They started with the friendship network and removed all userids that were not present in the properties file. Then, they labeled each vertex (userid) with its privacy setting and used a default label for all edges.

IMDB We use the movie-actor dataset, extracted by Kibriya and Ramon (2012). The network consists of movie, year, role and actor vertices. Movie and role vertices were labeled by movie and role type, whereas year vertices were labeled by the year the movie was released in. Actor vertices as well as all edges are left with a default label.

Table 3.1 presents basic statistics of all our real-world networks.

[4] http://odysseas.calit2.uci.edu/doku.php/public:online_social_networks
[5] http://www.imdb.com/interfaces

Table 3.1: Summary of real-world networks.

Dataset	# vertices	# edges	# vertex labels	# edge labels
Facebook-uniform	$984,830$	$185,508$	17	1
Facebook-mhrw	$957,359$	$1,792,188$	16	1
IMDB	$30,835,467$	$53,686,381$	144	1

3.6.3 Results

Mining frequent rooted trees.

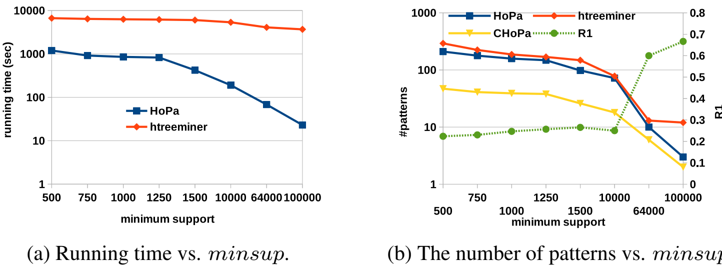
By setting tw to 1, every generated pattern will be a tree. However, the generated patterns, except those consisting of only one vertex, will have a root of size 2. Every rooted pattern generated in this way represents 2 different single root patterns; for each, we generate the canonical form (See Section 3.3.1) to ensure no duplicate patterns are generated and we count its frequency. In this way, we generate the set of frequent rooted trees where the root is a single vertex. The generated patterns are directed and the direction of the edges is from the end-point which is closer to the root to the other end-point. Therefore, in the experiments in this subsection, we consider the databases as directed graphs. Note that the facebook datasets are undirected. In the directed graphs made of them, the edges are directed from the vertex with the smaller vertex id to the vertex with the larger vertex id.

For each of the five datasets, we perform a range of experiments. First, we measure the number of patterns and runtime as functions of the minimum support. Second, we measure the number of patterns and runtime as functions of the maximum level. In each experiment we compare our proposed algorithm with htreeminer (Dries and Nijssen, 2012). The results are shown in Figures 3.10-3.19. In all charts of this section, running times of the algorithms and the numbers of patterns found are plotted in log-scale.

As discussed earlier, to find the complete set of patterns that are frequent under sup_{RE}, in Algorithm 4 all non-redundant (up to root isomorphism) generated patterns (either frequent or infrequent) are extended/joined to generate larger patterns. However, this may lead to a very large state space when using a bottom-up pattern generation strategy and sup_{RE}, and make the HoPa algorithm practically intractable. Therefore, in our experiments, we use sup^*_{RE} with respect to the generated BRETDs of the patterns and hence, find a subset of the patterns returned by htreeminer. In Figures 3.10-3.17, one can see what is the difference in the number of frequent patterns caused by the difference in frequency measure. In particular, patterns that are frequent under sup_{RE}

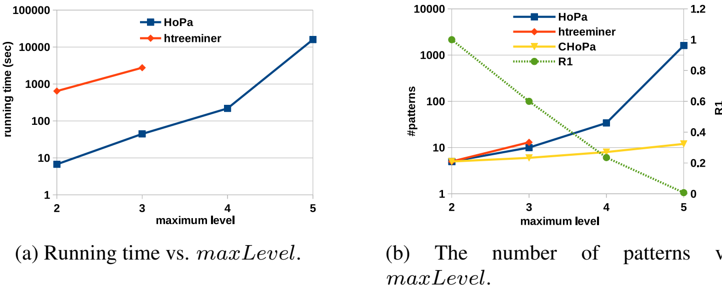
but infrequent under sup^*_{RE} usually form only a small fraction of the patterns that are frequent under sup_{RE}. A closer inspection reveals that usually these are patterns having a vertex with an infrequent label.

As reflected in Figures 3.10-3.19, in most cases HoPa significantly outperforms htreeminer in terms of running time. In particular, there are several cases where htreeminer fails (or does not terminate within a reasonable time, e.g., 2 days), but HoPa effectively finds frequent patterns (see Figures 3.11, 3.12, 3.13, 3.14, 3.15, 3.18 and 3.19, in the IMDB and $BA10^7$ figures there is no line for htreeminer because for none of the parameter settings it completed). By examining the slopes of the curves in the figures plotting runtime, one can get see that HoPa usually scales well towards more difficult settings.



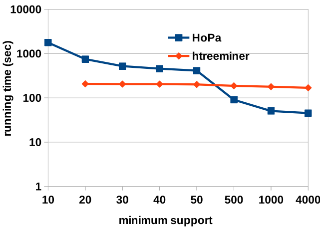
(a) Running time vs. $minsup$. (b) The number of patterns vs. $minsup$.

Figure 3.10: Experimental results for mining frequent rooted trees from facebook-mhrw for different values of $minsup$; $maxLevel$ is 3.

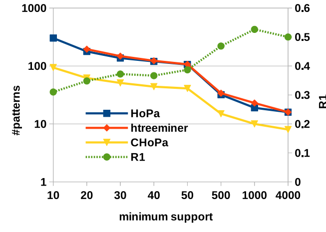


(a) Running time vs. $maxLevel$. (b) The number of patterns vs. $maxLevel$.

Figure 3.11: Experimental results for mining frequent rooted trees from facebook-mhrw for different values of $maxLevel$; $minsup$ is 64,000.

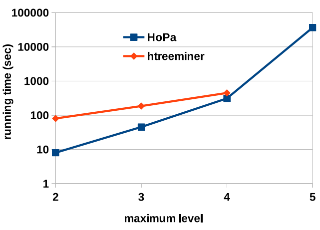


(a) Running time vs. $minsup$.

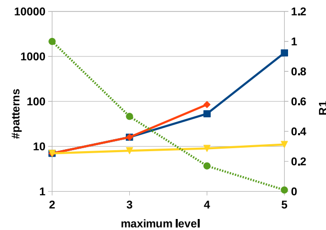


(b) The number of patterns vs. $minsup$.

Figure 3.12: Experimental results for mining frequent rooted trees from facebook-uniform for different values of $minsup$; $maxLevel$ is 3.



(a) Running time vs. $maxLevel$.



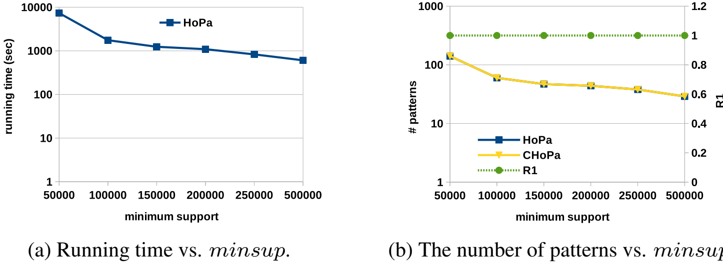
(b) The number of patterns vs. $maxLevel$.

Figure 3.13: Experimental results for mining frequent rooted trees from facebook-uniform for different values of $maxLevel$; $minsup$ is 4,000.

Mining frequent rooted graphs.

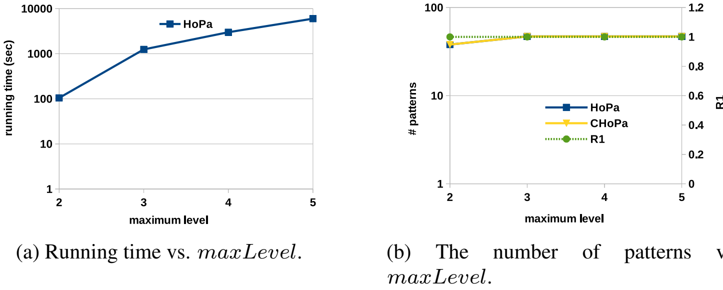
We go beyond rooted tree patterns and find frequent rooted graph patterns that may have a $treewidth$ larger than 1. Here rooted patterns larger than 1 will have a root of size 2 or 3 or ... or $treewidth + 1$; and only patterns consisting of one vertex will have a root of size 1. Therefore, rooted tree patterns with a root of size 1 that are found in the case of mining frequent rooted trees are not found anymore and instead, rooted tree patterns that have a root of size 2 or 3 or $treewidth + 1$ are found. Here, the generated patterns are not directed as there is no single root vertex to define the direction of the edges. Therefore, in the experiments of this case, we consider the databases as undirected graphs.

We noticed that in this case, there are patterns that have extremely large number of root embeddings; even several times more than the number of vertices in the network. Computing this huge number of root embeddings renders the algorithm practically intractable. To overcome this problem, we set the number of network vertices as an



(a) Running time vs. $minsup$. (b) The number of patterns vs. $minsup$.

Figure 3.14: Experimental results for mining frequent rooted trees from IMDB for different values of $minsup$; $maxLevel$ is 3.

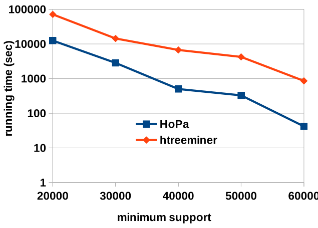


(a) Running time vs. $maxLevel$. (b) The number of patterns vs. $maxLevel$.

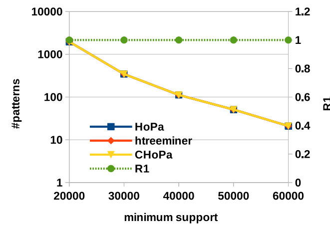
Figure 3.15: Experimental results for mining frequent rooted trees from IMDB for different values of $maxLevel$; $minsup$ is $150,000$.

upper bound on the number of root embeddings that are computed for a pattern. This means while computing the set of root embeddings of a pattern, when the set becomes larger than the network size, we stop finding the next element.

Figures 3.20, 3.21 and 3.22 present the empirical results for $treewidth = 1, 2, 3$ over the facebook-mhrw, facebook-uniform and $BA10^6$ datasets, respectively. To the best of our knowledge, there is no algorithm for finding bounded-treewidth graph patterns (with $treewidth > 1$) from large networks under homomorphism, therefore, the charts report only the empirical behavior of HoPa. In order to find patterns with $treewidth$ 3, we have to set $maxLevel$ to at least 4. This is our reason for setting $maxLevel$ to 4 in our experiments in Figures 3.20 and 3.21. However, for the $BA10^6$ dataset, when we set $maxLevel$ to 4, HoPa does not finish within a reasonable time (3-4 days), hence, we only report the results for $maxLevel = 3$. Note that in the case of $maxLevel = 3$, the set of found patterns are the same for $treewidth = 2$ and $treewidth = 3$, therefore, in Figure 3.22 we show the results for $treewidth = 1, 2$. Over the IMDB and $BA10^7$ datasets, the algorithm does not terminate within a reasonable time (3-4 days), therefore, we do not report any empirical results for them.

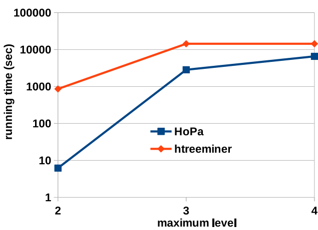


(a) Running time vs. $minsup$.

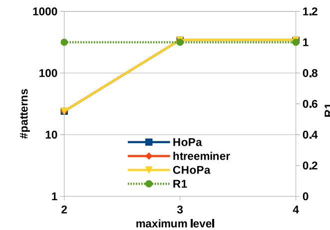


(b) The number of patterns vs. $minsup$.

Figure 3.16: Experimental results for mining frequent rooted trees from $BA10^6$ for different values of $minsup$; $maxLevel$ is 3.



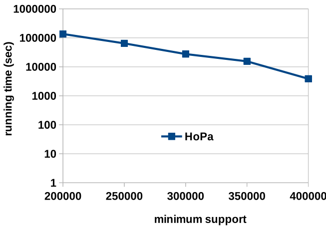
(a) Running time vs. $minsup$.



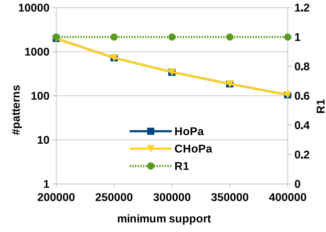
(b) The number of patterns vs. $minsup$.

Figure 3.17: Experimental results for mining frequent rooted trees from $BA10^6$ for different values of $maxLevel$; $minsup$ is $30,000$.

To better understand frequent patterns that have a root larger than 1, in Tables 3.2 and 3.3 we respectively provide for facebook-mhrw and facebook-uniform the statistics of frequent patterns, where $maxLevel$ is 4 and $treewidth$ 3. Note that patterns that have a root of size $maxLevel$ cannot join with any other pattern. In fact, BRETDs of such patterns are path. The reason is that for a pattern to have a root of size $maxLevel$, in all its extensions, the whole root of the pattern which is extended must be transfered to the new pattern. This means the pattern is joinable only with itself; such a join does not result in a valid BRETD. As a result, as shown in Tables 3.2 and 3.3, the number of patterns with a root of size 4 is less than the number of patterns with a root of size 2 (or a root of size 3); and these patterns have only 4 vertices.

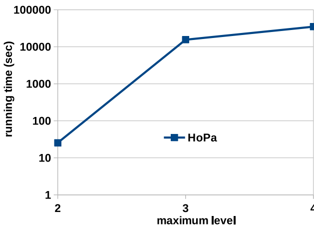


(a) Running time vs. $minsup$.

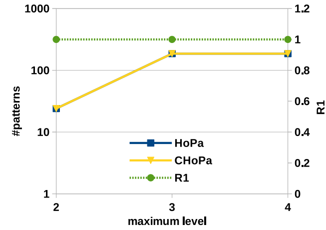


(b) The number of patterns vs. $minsup$.

Figure 3.18: Experimental results for mining frequent rooted trees from $BA10^7$ for different values of $minsup$; $maxLevel$ is 3.

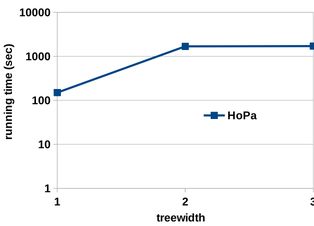


(a) Running time vs. $maxLevel$.

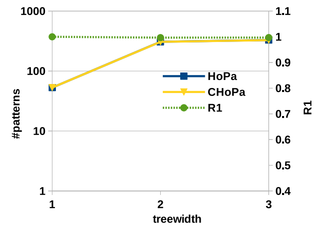


(b) The number of patterns vs. $maxLevel$.

Figure 3.19: Experimental results for mining frequent rooted trees from $BA10^7$ for different values of $maxLevel$; $minsup$ is $350,000$.

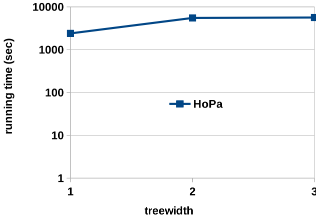


(a) Running time vs. $treewidth$.

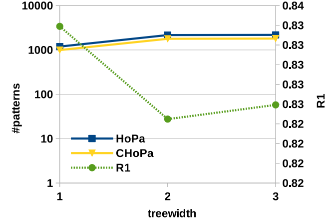


(b) The number of patterns vs. $treewidth$.

Figure 3.20: Experimental results over the facebook-mhrw dataset for $treewidth = 1, 2, 3$; $minsup$ is $1,500$ and $maxLevel$ is 4.

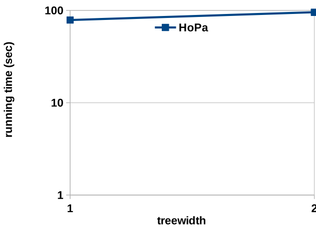


(a) Running time vs. $treewidth$.

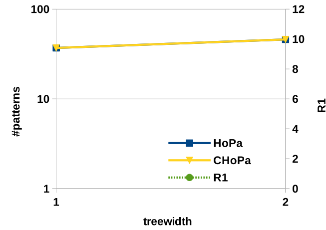


(b) The number of patterns vs. $treewidth$.

Figure 3.21: Experimental results over the facebook-uniform dataset for $treewidth = 1, 2, 3$; $minsup$ is 10 and $maxLevel$ is 4.



(a) Running time vs. $treewidth$.



(b) The number of patterns vs. $treewidth$.

Figure 3.22: Experimental results over the $BA10^6$ dataset for $treewidth = 1, 2$; $minsup$ is $30,000$ and $maxLevel$ is 3.

Table 3.2: Patterns statistics over facebook-mhrw ($minsup = 1,500$, $maxLevel = 4$ and $treewidth = 3$).

root size	pattern size																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Sum
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
2	0	4	6	11	11	25	20	44	24	34	13	12	9	9	3	2	227
3	0	0	7	30	22	14	5	0	0	0	0	0	0	0	0	0	78
4	0	0	0	21	0	0	0	0	0	0	0	0	0	0	0	0	21
Sum	3	4	13	62	33	39	25	44	24	34	13	12	9	9	3	2	329

Table 3.3: Patterns statistics over facebook-uniform ($minsup = 10$, $maxLevel = 4$ and $treewidth = 3$).

root size	pattern size																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Sum
1	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12
2	0	15	25	18	125	174	353	370	295	372	55	155	23	49	7	10	1	2047
3	0	0	18	32	26	23	6	0	1	0	0	0	0	0	0	0	0	106
4	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0	18
Sum	12	1	43	68	151	197	359	370	296	372	55	155	23	49	7	10	1	2183

3.6.4 Discussion

Based on the results reported above, we can answer the experimental questions as follows:

Q1. For finding rooted tree patterns from a large single network, HoPa is almost always significantly faster than htreeminer. In our experiments, the only situation where htreeminer outperforms HoPa is the facebook-uniform dataset. However, even over this dataset, by increasing $maxLevel$ or decreasing $minsup$, HoPa outperforms htreeminer. The reasons for high efficiency of HoPa are the use of infrequent parent check and also the faster frequency counting method.

Q2. For mining frequent rooted trees, while over very large networks such as IMDB and $BA10^7$ htreeminer fails, HoPa can effectively find frequent patterns (see e.g., Figures 3.14, 3.15, 3.18 and 3.19).

On the one hand, by fixing $minsup$ and $treewidth$ and increasing $maxLevel$, the ratio $R1$ usually decreases. This means most of patterns found in higher levels belong to one of the already formed root embedding equivalence classes. This can be seen e.g., in Figures 3.11b and 3.13b. On the other hand, by fixing $treewidth$ and $maxLevel$ and decreasing $minsup$, the ratio $R1$ usually does not change considerably, i.e., patterns that appear in lower values of $minsup$, form some new root embedding equivalence classes. This is expected as these patterns have different root embeddings and hence, belong to new root embedding equivalence classes.

Q3. In our experiments, by increasing $maxLevel$ or decreasing $minsup$, both running time and the number of frequent patterns usually grow exponentially (see e.g., Figures 3.11 and 3.13 for $maxLevel$ and Figures 3.10 and 3.12 for $minsup$).

3.7 Conclusion

In this chapter, we studied the problem of single network mining under homomorphism. We introduced a new class of patterns, called *rooted patterns*, and proposed an algorithm for complete generation of rooted patterns. We presented a new closure operator for compact representation of all frequent rooted patterns and investigated its properties. Then, we introduced a new algorithm, called HoPa, for finding frequent rooted patterns from a large single network under homomorphism. Finally, by performing extensive experiments over large real-world and synthetic networks, we showed the high efficiency of HoPa. In particular, by restricting our patterns to

rooted trees, we compared HoPa against htreeminer (Dries and Nijssen, 2012) and showed that there are several cases where htreeminer fails (due to lack of memory) or it does not terminate within a reasonable time (e.g., 3-4 days), but HoPa finds frequent patterns effectively.

Acknowledgements

We are thankful to Dr Anton Dries for providing us the htreeminer code. This work was supported by ERC Starting Grant 240186 "MiGraNT: Mining Graphs and Networks: a Theory-based approach".

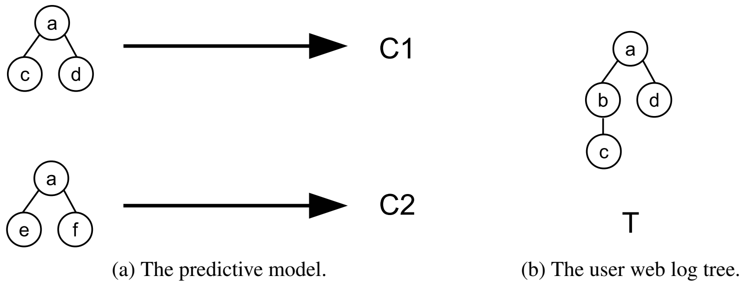
Chapter 4

Mining Rooted Ordered Trees under Subtree Homeomorphism

4.1 Introduction

Many semi-structured data such as XML documents are represented by rooted ordered trees. One of the most important problems in the data mining of such data is frequent pattern discovery. Mining frequent tree patterns is very useful in various domains such as network routing (Cui et al., 2002), bioinformatics (Zaki, 2005b) and user web log data analysis (Ivancsy and Vajk, 2006). Furthermore, it is a crucial step in several other data mining and machine learning problems such as clustering and classification (Zaki and Aggarwal, 2006).

In general, algorithms proposed for finding frequent tree patterns include two main phases: 1) generating candidate tree patterns, and 2) counting the frequency of every generated tree pattern in a given collection of trees (called the *database trees* from now on). The generation step (which involves a refinement operator) is computationally easy. There are methods, such as *rightmost path extension*, that can generate efficiently all non-redundant rooted ordered trees, i.e., each in $O(1)$ computational time (Asai et al., 2002; Zaki, 2005b). The frequency counting step is computationally expensive. Chi et al. (2003) made an empirical comparison of these two phases; they showed that a significant part of the time required for finding frequent patterns is spent on frequency counting. Thereby, the particular method used for frequency counting can



(a) The predictive model.

(b) The user web log tree.

Figure 4.1: Under subtree homeomorphism, the tree T can be classified as $C1$ while it does not match the model under subtree isomorphism and subtree homomorphism.

significantly affect the efficiency of the tree mining algorithm.

The frequency counting step involves a *matching operator* (Kilpelainen and Mannila, 1995). A widely used matching operator is *subtree homeomorphism*; an injective mapping that maps a parent-child relationship in the tree pattern onto an ancestor-descendant relationship in the database tree. Frequent tree patterns under subtree isomorphism are called *induced patterns* and frequent tree patterns under subtree homeomorphism are called *embedded patterns*. Our focus in this chapter is frequent embedded tree patterns.

Frequent embedded tree patterns have many applications in different areas. Zaki and Aggarwal (2006) presented XRules, a classifier based on frequent embedded tree patterns, and showed its high performance compared to classifiers such as SVM. In this algorithm, during the training phase, the frequent embedded tree patterns that are most closely related to a class variable are found. These tree patterns form structural rules. Then during the testing phase, these rules are used to perform the structural classification. Frequent tree patterns can be used for analyzing the navigational behavior of the web users, where they are useful for advertising, dynamic user profiling, etc (Ivancsy and Vajk, 2006). Zaki (2005b) suggested to use frequent embedded tree patterns for predicting the function of RNA. The idea is to look for RNA that is similar to RNA molecules with known structure and function. Frequent embedded patterns are used as features to predict the function.

Figure 4.1 illustrates the difference between subtree homeomorphism and subtree isomorphism in the context of prediction. Figure 4.1a presents a predictive model consisting of two structural rules (Zaki and Aggarwal, 2006). These rules characterize two classes $C1$ and $C2$. Figure 4.1b presents a tree T that models a user web log data. We aim at predicting the class of T. If either subtree isomorphism or subtree

homomorphism is used, it is not possible to determine the class of T based on the structural rules depicted in Figure 4.1a. However, if subtree homeomorphism is used, the predictive model will put T in the class $C1$. Zaki and Aggarwal (2006) used frequent tree patterns under subtree homeomorphism for prediction tasks and showed that the model outperforms algorithms such as SVM.

Two widely used frequency notions are *per-tree frequency*, where only the occurrence of a tree pattern inside a database tree is important; and *per-occurrence frequency*, where the number of occurrences is important, too. While there exist algorithms optimized for the first notion (Zaki, 2005b), (Tatikonda et al., 2006) and (Wang et al., 2004), this notion is covered also by the algorithms proposed for the second notion. Per-occurrence frequency counting is computationally more expensive than per-tree frequency counting. In the current chapter, our concern is *per-occurrence frequency*. An extensive discussion about the applications in which the second notion is preferred can be found e.g., in (Tan et al., 2008). One of the investigated examples is a digital library where author information are separately stored in database trees in some form, e.g., author–book–area–publisher. A user may be interested in finding out information about the popular publishers of every area. Then, the repetition of items within a database tree becomes important, hence, per-occurrence frequency is more suitable than per-tree frequency (Tan et al., 2008).

Two categories of approaches have been used for counting occurrences of tree patterns under subtree homeomorphism. The first category includes approaches that use one of the algorithms proposed for subtree homeomorphism between two trees. HTreeMiner (Zaki, 2005b) employs such an approach. The second category includes approaches that store the information representing/encoding the occurrences of tree patterns. Then, when the tree pattern is extended to a larger one, its stored information is also extended, in a specific way, to represent the occurrences of the extended pattern. These approaches are sometimes called *vertical* approaches. VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008) are examples of the methods that use such vertical approaches. As studied by Zaki (2005b), vertical approaches are more efficient than the approaches in the first category.

Many efficient vertical algorithms are based on the numbering scheme proposed by Dietz (1982). This scheme uses a tree traversal order to determine the ancestor-descendant relationship between pairs of vertices. It associates each vertex with a pair of numbers, sometimes called *scope*. For instance, VTreeMiner and TreeMinerD (Zaki, 2005b) and TwigList (Qin et al., 2007) use this scheme in different forms, to design efficient methods for counting occurrences of tree patterns.

The problem with these algorithms is that in order to count all occurrences, they use data-structures that represent whole occurrences. This renders the algorithms inefficient, especially when patterns are large and have many occurrences in the database trees. In the worst case, the number of occurrences of a tree pattern

can be exponential in terms of the size of pattern and database (Chi et al., 2005a). Therefore, keeping track of all occurrences can significantly reduce the efficiency of the algorithm, in particular when tree patterns have many occurrences in the database trees.

The main contribution of the current chapter is to introduce a novel vertical algorithm for the class of rooted ordered trees. It uses a more compact data-structure, called **occ** (an abbreviation for **oc**currence **c**ompressor) for representing occurrences, and a more efficient subtree homeomorphism algorithm based on Dietz's numbering scheme (Dietz, 1982). An **occ** data-structure stores only information about rightmost paths of occurrences and hence can represent/encode all occurrences that have the rightmost path in common. The number of such occurrences can be exponential, even though the size of the **occ** is only $O(d)$, where d is the length of the rightmost path of the tree pattern. We present efficient join operations on **occ** that help us to efficiently calculate the occurrence count of tree patterns from the occurrence count of their proper subtrees.

Furthermore, we observed that in most of widely used real-world databases, while many vertices of a database tree have the same label, no two vertices on the same path are identically labeled. For this class of database trees, worst case space complexity of our algorithm is linear; a result comparable to the best existing results for per-tree frequency. We note that for such databases, worst case space complexity of the well-known existing algorithms for per-occurrence frequency, such as VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008), is still exponential. Based on the proposed subtree homeomorphism method, we develop an efficient pattern mining algorithm, called TPMiner. To evaluate the efficiency of TPMiner, we perform extensive experiments on both real-world and synthetic datasets. Our results show that TPMiner always outperforms most efficient existing algorithms such as VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008). Furthermore, there are several cases where the improvement of TPMiner with respect to existing algorithm is significant.

The rest of this chapter is organized as follows. In Section 4.2, the problem studied in this chapter is introduced. In Section 4.3 a brief overview on related work is given. In Section 4.4, we present the **occ** data-structure and our subtree homeomorphism algorithm. In Section 4.5, we introduce the TPMiner algorithm for finding frequent embedded tree patterns from rooted ordered trees. We empirically evaluate the effectiveness of TPMiner in Section 4.6. Finally, the chapter is concluded in Section 4.7.

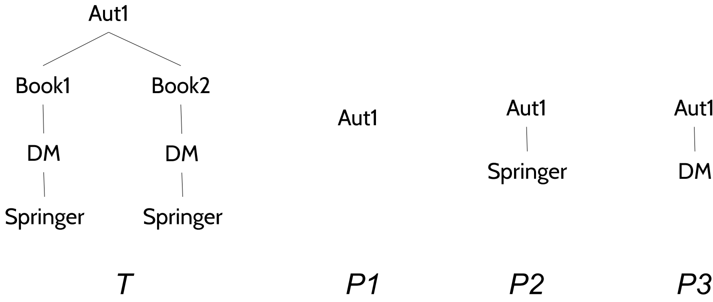


Figure 4.2: In the database tree T and for $minsup = 2$, while $P1$ is infrequent, it has two frequent supertrees $P2$ and $P3$.

4.2 Problem statement

Given a database $\mathcal{D}$ consisting of trees and a tree P, *per-tree support* (or *per-tree frequency*) of P in $\mathcal{D}$ is defined as: $|\{T \in \mathcal{D} : P \preceq^h T\}|$. *Per-occurrence support* (or *per-occurrence frequency*) of P in $\mathcal{D}$ is defined as: $\sum_{T \in \mathcal{D}} NumOcc(P, T)$. In this work, our focus is *per-occurrence support*. For the sake of simplicity, through this chapter, we use the term *support* (or *frequency*) instead of *per-occurrence support* (or *per-occurrence frequency*), and denote it by $sup(P, \mathcal{D})$. P is *frequent* (P is a *frequent embedded pattern*), iff its support is greater than or equal to an user defined integer threshold $minsup > 0$. The problem studied in this chapter is as follows: given a database $\mathcal{D}$ consisting of trees and an integer $minsup$, find every frequent pattern P such that $sup(P, \mathcal{D}) \geq minsup$.

We observe that when *per-occurrence support* is used, anti-monotonicity might be violated: it is possible that the support of P is greater than or equal to $minsup$, but it has a subtree whose support is less than $minsup$. For example, consider the database tree T of Figure 4.2 and suppose that $minsup$ is 2. Then, while the pattern $P1$ is infrequent, it has two frequent supertrees $P2$ and $P3$. Therefore, in a more precise (and practical) definition, which is also used by algorithms such as VTreeMiner (Zaki, 2005b), tree P is frequent iff: 1) $sup(P, \mathcal{D}) \geq minsup$, and 2) the subtree P' generated by removing the rightmost vertex of P is frequent. This means only frequent trees are extended to generate larger patterns.

4.3 Related work

Recently, many algorithms have been proposed in the literature for finding frequent tree patterns from a database of tree-structured data.

Mining frequent tree patterns from tree databases under subtree homeomorphism. Zaki (2002) presented VTreeMiner to find embedded patterns from trees. For frequency counting he used an efficient data structure, called *scope-list*, and proposed rightmost path extension to generate non-redundant candidates. Zaki (2005b) presented TreeMinerD to find embedded patterns when per-tree support is used. For frequency counting, he developed an efficient data structure, called *SV-list*, and introduced efficient join operators on *SV-list*s. Later, Zaki (2005a) proposed the SLEUTH algorithm to mine embedded patterns from rooted unordered trees. Xiao et al. (2005) proposed TreeGrow for mining maximal embedded tree patterns from rooted unordered trees. However, TreeGrow assumes that the labels for the children of every vertex are unique. XSpanner uses a pattern growth-based method to find embedded tree patterns (Wang et al., 2004). Tatikonda et al. (2006) proposed a generic approach for mining embedded or induced subtrees that can be labeled, unlabeled, ordered, unordered, or edge-labeled. They developed TRIPS and TIDES algorithms for the *per-tree support* setting using two sequential encodings of trees to systematically generate and evaluate the candidate patterns. Tan et al. (2008) introduced the MB3Miner algorithm, where they use a unique occurrence list representation of the tree structure, that enables efficient implementation of their Tree Model Guided (TMG) candidate generation. TMG can enumerate all the valid candidates that fit in the structural aspects of the database.

A drawback of these algorithms is that in order to count the number of occurrences of a tree pattern P in a database tree T, they need to keep track of all occurrences of P in T. For example, in VTreeMiner, for every occurrence φ of P in T a separate element is stored in *scope-list*, that consists of the following components: (i) TId which is the identifier of the database tree that contains the occurrence, (ii) m which is $\{\varphi(v)|v \in V(P) \setminus \{\text{rightmost vertex of } P\}\}$, and (iii) s which is the scope of $\varphi(u)$, where u is the rightmost vertex of P. In the current chapter, we propose a much more compact data-structure that can represent/encode all occurrences that have the rightmost path in common in $O(d)$ space, where d is the length of the rightmost path of P. The number of such occurrences can be exponential. We then present efficient algorithms that calculate the occurrence count of tree patterns from the occurrence count of their proper subtrees.

A slightly different problem over rooted ordered trees is the tree inclusion problem: can a pattern P be obtained from a tree T by deleting vertices from T. In our terminology, is P present in T under subtree homeomorphism? Bille and Gortz (2011)

recently proposed a novel algorithm that runs in linear space and subquadratic time, improving upon a series of polynomial time algorithms that started with the work of Kilpelainen and Mannila (1995).

Mining frequent tree patterns from tree databases under subtree isomorphism. Asai et al. (2002) independently proposed the rightmost path extension technique for candidate generation. They developed FreqT for mining frequent induced tree patterns. Algorithms for discovering similar structure and structural association rules among a collection of tree-structured data can be found in (Wang and Liu, 1998) and (Wang and Liu, 2000). Chi et al. (2003) proposed FreeTreeMiner for mining induced patterns from rooted unordered trees and free trees. Other algorithms for mining induced patterns from rooted unordered tree are PathJoin (Xiao et al., 2003), uFreqt (Nijssen and Kok, 2003), uNot (Asai et al., 2003) and HybridTreeMiner (Chi et al., 2004a). Miyahara et al. (2001) presented an algorithm for finding all maximally frequent tag tree patterns in semi-structured data. Chi et al. (2004b) proposed CMTreeMiner for mining both closed and maximal frequent tree patterns. Their algorithm traverses an enumeration tree that systematically enumerates all subtrees, and uses an enumeration DAG to prune the branches of the enumeration tree that do not correspond to closed or maximal frequent subtrees. Chehreghani et al. (2011) presented the OInduced algorithm for finding frequent induced tree patterns. They introduced three novel encodings for rooted ordered trees and showed that when the matching operator is subtree isomorphism, frequency of every pattern can be computed effectively, using these tree encodings. Later, Chehreghani (2011) proposed other encodings for rooted unordered trees and accordingly, presented an efficient method for frequency counting of unordered tree patterns under subtree isomorphism.

A brief comparison of pattern mining under the three matching operators.

- From the viewpoint of complexity of the frequency counting phase, in general, subgraph homomorphism is easier than subgraph isomorphism and subgraph homeomorphism. There are graph classes such as *bounded treewidth graphs* for which subgraph homomorphism is solvable in a time polynomial in terms of network size and pattern size; however, subgraph isomorphism and subgraph homeomorphism are still hard problems (Matousek and Thomas, 1992b).

- The candidate generation phase under subtree homomorphism is more challenging than subtree isomorphism and subtree homeomorphism. In particular, a larger pattern P might be more general than a smaller pattern P', i.e., P might be subtree homomorphic to P'. This makes the ordered search more complicated than the cases of subtree isomorphism and subtree homeomorphism. Furthermore, under subtree homomorphism, a pattern P might be generated

from a smaller pattern P' by adding any arbitrary number of edges, rather than a fixed number of edges. This makes traversing the search space non-trivial.

- From the applicability point of view, as mentioned in Section 4.1, when longer range relationships are relevant, subtree homeomorphism becomes more useful than subtree isomorphism and subtree homomorphism.

4.4 Efficient tree mining under subtree homeomorphism

In this section, we present our method for subtree homeomorphism of rooted ordered trees. First in Section 4.4.1, we introduce the notion of *occurrence tree* and its rightmost path extension. Then in Section 4.4.2, we present the **occ-list** data-structure and, in Section 4.4.3, the operators for this data structure. In Section 4.4.4, we analyze space and time complexities of our proposed frequency counting method. We briefly compare our approach with other vertical frequency counting methods in Section 4.4.5.

4.4.1 Occurrence trees and their extensions

Under rightmost path extension, a pattern P with $k+1$ vertices is generated from a pattern P' with k vertices by adding a vertex v, as the rightmost child, to a vertex in the rightmost path of P. Occurrences of P are rightmost path extensions of occurrences of P' with an occurrence of v. Therefore, an interesting way to construct occurrences of P is to look at the occurrences of v that can be a rightmost path extension of an occurrence of P'. First, we introduce the notion of *occurrence tree*. Then, we present the conditions under which a rightmost path extension of an occurrence tree yields another occurrence tree.

Definition 17 (Occurrence tree)**.** *Given an occurrence φ of P in T, we define the* occurrence tree $\mathcal{OT}(\varphi)$ *as follows: (i) $V(\mathcal{OT}(\varphi)) = \{\varphi(v) : v \in V(P)\}$, (ii) $root(\mathcal{OT}(\varphi)) = \varphi(root(P))$, for every $v \in V(\mathcal{OT}(\varphi))$, $\lambda_{\mathcal{OT}(\varphi)}(v) = \lambda_T(v)$, and (iii) $E(\mathcal{OT}(\varphi)) = \{(\varphi(v_1), \varphi(v_2)) | (v_1, v_2) \in E(P)\}$.*

Notice, when $(v_1, v_2) \in E(\mathcal{OT}(\varphi))$ and v_1 is not the parent of v_2 in T, then all intermediate vertices on the path from v_1 to v_2 are not part of $V(\mathcal{OT}(\varphi))$. For example, in Figure 4.3, the tree P has 3 occurrences in tree T.

Selecting a vertex not yet in the occurrence tree and performing a rightmost path extension does not always result in another occurrence tree. Proposition 8 below lists properties that hold when the rightmost path extension is an occurrence tree.

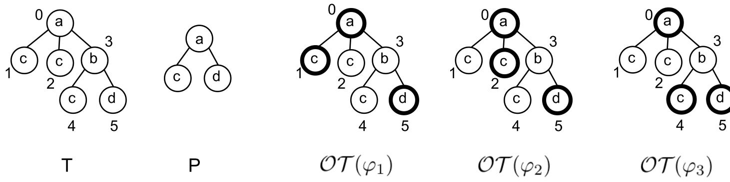


Figure 4.3: From left to right, a database tree T, a pattern P and three occurrence trees $\mathcal{OT}(\varphi_1)$, $\mathcal{OT}(\varphi_2)$, and $\mathcal{OT}(\varphi_3)$. Labels are inside vertices, preorder numbers are next to vertices. The occurrence trees are represented by showing the occurrences of the pattern vertices in bold. Their edges are the images of the edges in the pattern; for example, $\mathcal{OT}(\varphi_3)$ refers to the tree formed by vertices 0, 4 and 5, with 0 as the root, and with $(0, 4)$ and $(0, 5)$ as edges.

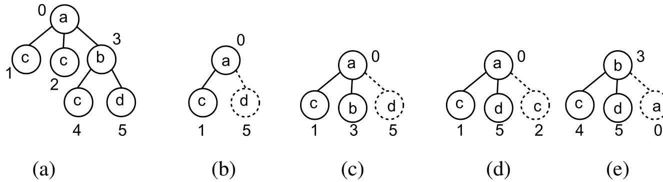


Figure 4.4: A database tree is shown in 4.4a, together with four rightmost path extensions of different occurrence trees in that database tree. However, only 4.4b is itself an occurrence tree in the database tree; 4.4c, 4.4d and 4.4e violate conditions (ii), (iii) and (i) of Proposition 9, respectively.

Proposition 8. *Let φ' be an occurrence of a pattern P' in a database tree T and $OT' = \mathcal{OT}(\varphi')$. Let x be a vertex of T outside OT', and y a vertex on the rightmost path of OT'. If $OT = RExtend(OT', x, y)$ is an occurrence tree in T, then: (i) $root(OT')$ is an ancestor of x in T, (ii) of all ancestors of x in T that belong to OT', y is the largest one in the preorder over T, and (iii) for each vertex w in OT' such that $p(w) > p(y)$, w is a left relative of x in OT and in T.*

Proof. By the definition of rightmost path extension presented in Chapter 2, either y is $root(OT')$ or it is a descendant of $root(OT')$; moreover y is the parent of x, hence (i) holds. As y is on the rightmost path in OT' and y is the parent of x in OT (and children of y are relatives of x in T), y is, among the ancestors of x in T, the largest one that belongs to OT' (ii). If $p(w) > p(y)$ for a vertex $w \in V(OT')$, since y is on the rightmost path of OT', w is a descendant of y. Furthermore, since x is the rightmost child of y in OT, w is a left relative of x in OT and in T (iii). $\square$

As an example of Proposition 8, consider Figure 4.4, where 4.4a presents a database tree and 4.4b shows that an occurrence tree OT' consisting of vertices 0 and 1 is extended to another occurrence tree OT consisting of vertices 0, 1 and 5. Vertex 0 is an ancestor of vertex 5 in the database tree (condition (i)); among all ancestors of vertex 5 in the database tree that belong to OT', vertex 0 is the largest one in the preorder over the database tree (condition (ii)); and vertex 1 is a left relative of vertex 5 in OT and in the database tree (condition (iii)).

Let P' be a tree pattern. The next proposition lists the conditions that are sufficient to ensure that a rightmost path extension of an occurrence tree $OT(\varphi')$ of P' with an edge (y, x) yields an occurrence tree of another tree pattern P, where P is a rightmost path extension of P'.

Proposition 9. *Let φ' be an occurrence of a pattern P' in a database tree T and $OT' = \mathcal{OT}(\varphi')$. Let x be a vertex of T outside OT', and y a vertex on the rightmost path of OT'. If: (i) y is an ancestor of x in the database tree T, (ii) of all vertices of OT' that are ancestors of x in the database tree T, y is the largest one, and (iii) $p(x) > p(w)$ for all $w \in OT'$, then $RExtend(OT', x, y)$ is an occurrence tree. For a given vertex x and an occurrence tree OT, if there exists a vertex y such that $RExtend(OT, x, y)$, y is unique.*

Proof. Let u be the vertex of P' such that $y = \varphi(u)$. To define P, we set $V(P) = V(P') \cup \{v\}$ with v a new vertex with the same label as x, and $E(P) = E(P') \cup \{(u, v)\}$ such that v is the rightmost child of u. Define OT as $V(OT) = V(OT') \cup \{x\}$ and $E(OT) = E(OT) \cup \{(y, x)\}$ and set $\varphi(P) = \varphi'(P') \cup \{v \to x\}$. By the construction and the assumptions, $OT = RExtend(OT', x, y)$. We show OT is a occurrence tree of P in T. From (i) and (ii) it follows that x is on the rightmost path from $root(OT)$ and it is a descendant of y in T and from (iii) that the children of y in OT are left relatives of x, hence, φ is an embedding of P in T and OT is an occurrence tree of P in T. We note since all vertices of a tree have a unique preorder number, y is unique, if it exists. □

Figure 4.4 shows a database tree (4.4a) and four rightmost path extensions of occurrence trees; however, only one of these extensions is another occurrence tree (4.4b), the other ones violate the conditions of Proposition 9.

To turn the conditions of Proposition 9 into a practical method, we need a compact way to store occurrence trees and an efficient way to check the conditions. For the latter, we take advantage of the solution of Dietz (1982). He has designed a numbering scheme based on tree traversal order to determine the ancestor-descendant relationship between any pair of vertices. This scheme associates each vertex v with a pair of numbers $\langle p(v), p(v) + size(v) \rangle$, where $size(v)$ is an integer with certain properties (which are met when e.g., $size(v)$ is the number of descendants of v).

Then, for two vertices u and v in a given database tree, u is an ancestor of v iff $p(u) < p(v)$ and $p(v) + size(v) \leq p(u) + size(u)$, and v is a right relative of u iff $p(u) + size(u) < p(v)$. In several algorithms, such as VTreeMiner and TreeMinerD (Zaki, 2005b) and TwigList (Qin et al., 2007), variants of this scheme have been used to design efficient methods for counting occurrences of tree patterns based on occurrences of their subtrees.

Our contribution is to introduce data structures, based on the Dietz numbering scheme, that allow us to speed up counting of all occurrences of tree patterns. For example, while the algorithm of Zaki (2005b) keeps track of all occurrences, we only store the occurrences that have distinct rightmost paths. We start with introducing some additional notations. The scope of a vertex x in a database tree T, denoted $x.scope$, is a pair (l, u), where l is the preorder number of x in T and u is the preorder number of the rightmost descendant of x in T. We use the notations $x.scope.l$ and $x.scope.u$ to refer to l and u of the scope of x.

Definition 18 (rdepth)**.** *Let x be a vertex on the rightmost path of a tree T. The $rdepth$ of x in T, denoted $rdep_T(x)$, is the length of the path from the root of T to x.*

A vertex x on the rightmost path of T is uniquely distinguished by $rdep_T(x)$. For example in Figure 4.4a, the *rdepth* of vertices 0, 3 and 5 is 0, 1 and 2, respectively (and for the other vertices, *rdepth* is undefined).

Proposition 10. *Let OT' be an occurrence tree of a tree pattern P' in a database tree T, $x \in V(T) \setminus V(OT')$ and y a vertex on the rightmost path of OT' but not the rightmost vertex (i.e., the rightmost path of OT' has a vertex z such that $rdep_{OT'}(z) = rdep_{OT'}(y) + 1$). We have: $RExtend(OT', x, y)$ is an occurrence tree iff*

$$z.scope.u < x.scope.l \leq y.scope.u \tag{4.1}$$

Proof. First, assume $RExtend(OT', x, y)$ is an occurrence tree. By Proposition 8, y is the largest vertex of OT' that is an ancestor of x, hence, in the database tree T, the tree rooted at x is a subtree of the tree rooted at y. That means

$$y.scope.l < x.scope.l \leq x.scope.u \leq y.scope.u \tag{4.2}$$

Vertex z and all vertices in the subtree of z are left relatives of x and have a preorder number smaller than that of x. Hence $z.scope.u < x.scope.l$. Combining with Inequation 4.2, we obtain $z.scope.u < x.scope.l \leq y.scope.u$.

For the other direction, Inequation 4.1 yields that x is a right relative of z and it is in the scope of the subtree of y, hence, y is an ancestor of x ((i) of Proposition 9) and z is not an ancestor of x, so y is the largest ancestor of x that belongs to OT' ((ii) of Proposition 9). Also, the preorder number of x is larger than that of any vertex in

the subtree of z and hence of any vertex in OT' ((iii) of Proposition 9). Hence, by Proposition 9, $RExtend(OT', x, y)$ is an occurrence tree. □

Proposition 11. *Let OT' be an occurrence tree of a tree pattern P' in a database tree T, $x \in V(T) \setminus V(OT')$ and y the rightmost vertex of OT'. We have: $RExtend(OT', x, y)$ is an occurrence tree iff*

$$y.scope.l < x.scope.l \text{ and } x.scope.u \leq y.scope.u \tag{4.3}$$

Proof. First, assume $RExtend(OT', x, y)$ is an occurrence tree. Similar to the proof of Proposition 10, by Proposition 8, we get

$$y.scope.l < x.scope.l \leq x.scope.u \leq y.scope.u \tag{4.4}$$

For the other direction, we assume $y.scope.l < x.scope.l$ and $x.scope.u \leq y.scope.u$. This implies that the tree rooted at x is a subtree of the tree rooted at y and that the preorder number of x is larger than the preorder number of y and hence that all conditions of Proposition 9 are satisfied, and $RExtend(OT', x, y)$ is an occurrence tree. □

For example, in Figure 4.4, first let OT' refer to the occurrence tree consisting of a single vertex 0. The scopes of vertices 0 and 1 are $(0, 5)$ and $(1, 1)$, respectively. The lower bound of the scope of vertex 1 is greater than the the lower bound of the scope of vertex 0; and the upper bound of the scope of vertex 1 is smaller than or equal to the upper bound of the scope of vertex 0. Therefore, Inequation 4.3 holds and $RExtend(OT', 1, 0)$ is an occurrence tree. Now, let OT' refer to the occurrence tree consisting of vertices 0 and 1. The scope of vertex 5 is $(5, 5)$. The lower bound of the scope of vertex 5 is greater than the upper bound of the scope of vertex 1; and it is smaller than or equal to the upper bound of the scope of vertex 0. Hence, Inequation 4.1 holds and $RExtend(OT', 5, 0)$ is an occurrence tree.

4.4.2 Occ-list: an efficient data structure for tree mining under subtree homeomorphism

For the rightmost path extension of an occurrence tree, it suffices to know its rightmost path. Different occurrences of a pattern can have the same rightmost path. The key improvement over previous work is that we only store information about the rightmost path of occurrence trees and that different occurrence trees with the same rightmost path are represented by the same data element. All occurrences of a pattern in a database tree are represented by **occ-list**, a list of occurrences. An element **occ** of **occ-list** represents all occurrences with a particular rightmost path. The element has four components:

- TId: the identifier of the database tree that contains the occurrences represented by **occ**.
- $scope$: the scope in the database tree TId of the last vertex in the rightmost path of the occurrences represented by **occ**,
- RP: an array containing the upper bounds of the scopes of the vertices in the rightmost path of the occurrences represented by **occ**, i.e., with x the vertex at rdepth j in the rightmost path of the pattern P and φ one of the occurrences represented by **occ**, $RP[j] = \varphi(x).scope.u$; note that this is the same value for all occurrences φ represented by **occ**[1].
- $multiplicity$: the number of occurrences represented by **occ**.

For all occurrences of a pattern P that have the same TId, $scope$ and RP, one **occ** in the **occ-list** of P is generated and its $multiplicity$ shows the number of such occurrences. All **occ**s of a pattern of size 1 have $multiplicity$ 1 as their (single vertex) rightmost paths are all different. Every occurrence is represented by exactly one **occ**. We refer to the **occ-list** of P by **occ-list**(P). It is easy to see that the frequency of P is equal to $\sum_{oc \in \textbf{occ-list}(P)} oc.multiplicity$. An example of **occ-list** is shown in Figure 4.5.

4.4.3 Operations on the occ-list data structure

Let P be a tree of size $k+1$ generated by adding a vertex v to a tree P' of size k. There are two cases:

1. v is added to the rightmost vertex of P'. We refer to this case as *leaf join.*
2. v is added to a vertex in the rightmost path of P', but not its rightmost vertex. We refer to this case as *inner join.*

They correspond to Propositions 10 and 11.

Proposition 12 (leaf join)**.** *Let v be a one element pattern, P' a pattern with u as the rightmost vertex, $d = rdep_{P'}(u)$ and $P = RExtend(P', v, u)$. Let* ***occ-list(**P'**)**, **occ-list(**v**)*** *and* ***occ-list(**P**)*** *be the representations of the occurrences of P', v and P, respectively. We have: $oc \in$* ***occ-list***(P) *iff there exist an $oc' \in$* ***occ-list***(P') *and an $ov \in$* ***occ-list***(v) *such that*

[1] The upper bound of the scope of the last vertex is already available in scope; for convenience of presentation, the information is duplicated in RP.

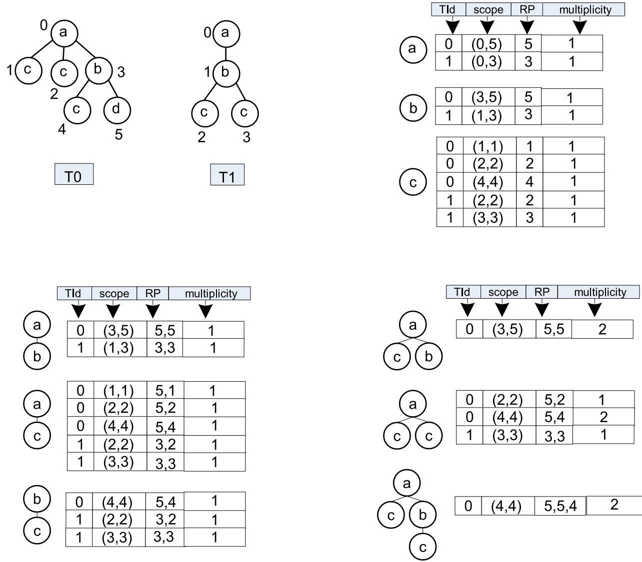


Figure 4.5: An example of **occ-list**. $T0$ and $T1$ are two database trees and minimum-support is equal to 2. The figure presents the **occ-list**s of some frequent 1-tree patterns, frequent 2-tree patterns, frequent 3-tree patterns and frequent 4-tree patterns.

(i) $oc'.TId = ov.TId = oc.TId$ *(all occurrences are from the same database tree),*

(ii) $oc'.scope.l < ov.scope.l$ *and* $ov.scope.u \leq oc'.scope.u$*,*

(iii) $oc.RP[i] = oc'.RP[i]$ $(0 \leq i \leq d)$ *and* $oc.RP[d+1] = ov.scope.u$ *(i.e., copy of* $oc'.RP$ *and an extra element),*

(iv) $oc.scope = ov.scope$*, and*

(v) $oc.multiplicity = oc'.multiplicity$

Proof. First, assume $oc \in$ **occ-list**(P), hence it represents $oc.multiplicity$ occurrence trees of pattern P in database tree $oc.TId$. Each of these occurrence trees share the same rightmost path. Hence they can be decomposed into occurrence trees of pattern P' sharing the same rightmost path and a particular occurrence of v. The latter is represented by an element ov of **occ-list**(v). The formers are represented by an element oc' of **occ-list**(P'). Now we show that conditions (i), ..., (v) hold between oc and these elements oc' and ov. Condition (i) holds because all occurrences are in database tree $oc.TId$, (ii) follows from Proposition 11, condition (iii) follows because

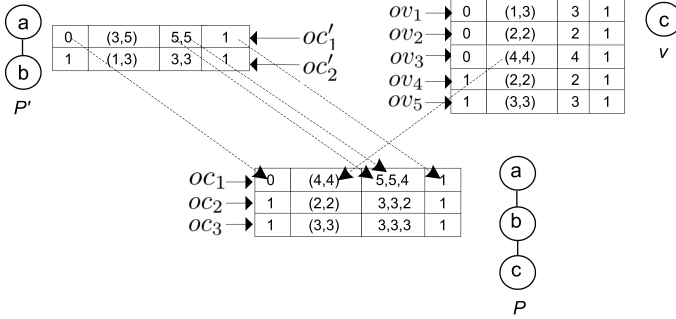


Figure 4.6: Details of the relationship between **occ-list**(P'), **occ-list**(v) and **occ-list**(P) for the database trees of Figure 4.5. The entries oc'_1, ov_3 and oc_1 satisfy the properties of Proposition 12. Also the tuples (oc'_2, ov_4, oc_2) and (oc'_2, ov_5, oc_3) satisfy the properties. The proposition is exploited in Algorithm 6 below. Its $leaf_join$ operation uses **occ-list**(P') and **occ-list**(v) to compute **occ-list**(P).

the rightmost path of the occurrence trees represented by oc' is identical to that of oc, except for the last element which is removed, (iv) follows from the definition of $scope$, and (v) holds because oc and oc' represent the same number of occurrences.

Second, assume there are an $oc' \in$ **occ-list**(P') and an $ov \in$ **occ-list**(v) such that conditions (i), ..., (v) hold. From (i) it follows that oc' and ov present occurrences in the same database tree. Because (ii) holds, if follows from Proposition 11 that all occurrence trees of P' represented by oc' can be extended with ov into occurrence trees of P; all these occurrence trees have the same rightmost path and have ov as their rightmost vertex, moreover they are the only ones in the database tree $oc'.TId$ with such a rightmost path. Hence, the element oc that satisfies properties (i), ..., (v) is indeed an element of **occ-list**(P) as has to be proven. $\square$

Figure 4.6 illustrates the proposition. The proposition is the basis for the $leaf_join$ operation in Algorithm 6 below.

In contrast with leaf join, which is performed between _one_ **occ** of a tree pattern and _one_ **occ** of a vertex, inner join is performed between _a set of **occs**_ of a tree pattern and _one_ **occ** of a vertex.

Proposition 13 (Inner join)**.** *Let v be a one element pattern, P' a pattern, u a vertex on the rightmost path of P' but not the rightmost vertex, $c = rdep_{P'}(u)$ and $P = RExtend(P', v, u)$. Let **occ-list**(P'), **occ-list**(v) and **occ-list**(P) be the representations of the occurrences of P', v and P, respectively. We have: $oc \in$ **occ-list**(P) iff there exist a subset $oc'_1, \ldots, oc'_m$ ($m \geq 1$) of **occ-list**(P') and an $ov \in$ **occ-list**(v) such that*

(i) $ov.TId = oc'_1.TId = \ldots = oc'_m.TId = oc.TId$ *(all occurrences are from the same database tree),*

(ii) $oc'_i.RP[k] = oc'_j.RP[k]$*, for all* $i, j \in [1..m]$ *and for all* $k \in [0..c]$*,*

(iii) $oc'_i.RP[c+1] < ov.scope.l \leq oc'_i.RP[c]$ *for all* $i \in [1..m]$*,*

(iv) $oc'_1, \ldots, oc'_m$ *is maximal with the conditions (i)-(iii),*

(v) $oc.RP[i] = oc'.RP[i]$*,* $0 \leq i \leq c$*, and* $oc.RP[c+1] = ov.scope.u$ *(copy of part of* $oc'.RP$ *and an extra element* $ov.scope.u$*),*

(vi) $oc.scope = ov.scope$*, and*

(vii) $oc.multiplicity = \sum_{i=1}^{m} oc'_i.multiplicity$*.*

Proof. First, assume $oc \in$ **occ-list**(P), hence it represents $oc.multiplicity$ occurrence trees of pattern P in database tree $oc.TId$. All these occurrence trees share the same rightmost path. Hence, they can be decomposed into occurrence trees of pattern P' sharing the first $c+1$ vertices of the rightmost path and a particular occurrence of v. The latter is represented by an element ov of **occ-list**(v). The formers are represented by elements $oc'_1, \ldots, oc'_m$ of **occ-list**(P'). Now we show that conditions (i), ..., (vii) hold between oc and these elements $oc'_1, \ldots, oc'_m$ and ov. Condition (i) holds because all occurrence trees are in database tree $oc.TId$, (ii) holds because all occurrence trees represented by $oc'_1, \ldots, oc'_m$ share the first $c+1$ vertices of the rightmost paths, (iii) and (iv) follow from Proposition 10, (v) follows because the first $c+1$ vertices of the rightmost paths of the occurrence trees represented by oc are identical to those of $oc'_1, \ldots, oc'_m$, and the rightmost vertex of the occurrence trees represented by oc is the vertex represented by ov, (vi) follows from the definition of $scope$, and (vii) holds because oc represents $\sum_{i=1}^{m} oc'_i.multiplicity$ occurrences.

Second, assume there are a maximal subset $oc'_1, \ldots, oc'_m$ of **occ-list**(P') and an $ov \in$ **occ-list**(v) such that conditions ((i), ..., (vii) hold. From (i) it follows that oc' and ov present occurrences in the same database tree. Because (iii) and (iv) hold, if follows from Proposition 10 that all occurrence trees of P' represented by $oc'_1, \ldots, oc'_m$ can be extended with ov into occurrence trees of P; all these extended occurrence trees have the same rightmost path and have ov as their rightmost vertex, moreover they are the only ones in the database tree $oc'.TId$ with such a rightmost path. Hence, the element oc that satisfies properties (i), ..., (vii) is indeed an element of **occ-list**(P) as has to be proven. □

Figure 4.7 illustrates the proposition. The proposition is the basis for the $inner_join$ operation in Algorithm 6 below.

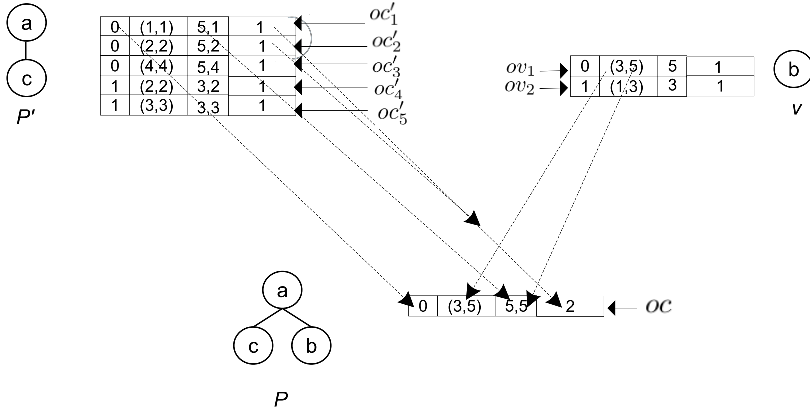


Figure 4.7: Details of the relationship between **occ-list**(P'), **occ-list**(v) and **occ-list**(P) for the database trees of Figure 4.5. The entries $\{oc'_1, oc'_2\}$, ov_1 and oc satisfy the properties of Proposition 13. As $c = 0$, rightmost paths of the occurrence trees represented by oc'_1 and oc'_2 share the first vertex, that is vertex 0 of $T0$; and rightmost paths of the occurrence trees represented by oc share the first and second vertices, that are vertices 0 and 3 of $T0$. The proposition is exploited in Algorithm 6 below. Its $inner_join$ operation uses **occ-list**(P') and **occ-list**(v) to compute **occ-list**(P).

4.4.4 Complexity analysis

Space complexity. Given a database $\mathcal{D}$, space complexity of the **occ-list** of a pattern of size 1 is $O(n \times |\mathcal{D}|)$, where n is the maximum number of vertices that a database tree $T \in \mathcal{D}$ has. For larger patterns, space complexity of the **occ-list** of a pattern P is $O(b \times d \times |\mathcal{D}|)$, where b is the maximum number of occurrences with distinct rightmost paths that P has in a database tree $T \in \mathcal{D}$, and d is the length of the rightmost path of P.

Compared to data-structures generated by other algorithms such as VTreeMiner, **occ-list** is often substantially more compact. Given a database $\mathcal{D}$, the size of the data-structure generated by VTreeMiner for a pattern P is $O(e \times k \times |\mathcal{D}|)$, where e is the maximum number of occurrences that P has in a database tree $T \in \mathcal{D}$ and k is $|V(P)|$. We note that on one hand, $k \geq d$ and the other hand, $e \geq b$. In particular, e can be significantly larger than b, e.g., it can be exponentially (in terms of n and k) larger than b. Therefore, in the worst case, the size of the data-structure generated by VTreeMiner is exponentially larger than **occ-list** (and it is never smaller than **occ-list**). An example of this situation is shown in Figure 4.8. In this figure, pattern P has $\binom{n-1}{k-1} \geq \left(\frac{n-1}{k-1}\right)^{k-1} \geq \left(\frac{n}{k}\right)^{k/2}$ occurrences in the database tree T ($k > 2$ and $n \geq k$).

Figure 4.8: Figure (a) shows a tree pattern P and (b) a database tree T. The number of occurrences of P in T is exponential in terms of n and k. In this case, the size of **occ-list** is linear, however, the size of the data-structures generated by the other algorithms is exponential.

Figure 4.9: Figure (a) shows a tree pattern P and (b) a database tree T. The number of occurrences of P in T is exponential in terms of n and k. In this case, the size of the **occ-list** and the size of the data-structures generated by the other algorithms for P are exponential.

If $k = (n + 1)/2$, the size of the data-structure generated by VTreeMiner will be $\Omega(2^{n/4})$. However, in this case, the size of the **occ-list** generated for P is $\Omega(n)$. More precisely, **occ-list**(P) has $n - k + 1$ **occ**s, where for each i, $k - 1 \leq i \leq n - 1$, there exists an **occ** with $TId = 0$, $scope = (i, i)$, $RP = \{n - 1, i\}$ and $multiplicity = \binom{i-1}{k-2}$.

We note that there are cases where the size of **occ-list** (as well as the size of the data-structures used by the other algorithms) becomes exponential. An example of this situation is presented in Figure 4.9, where for $k = n/2$, the size of **occ-list** as well as the size of *scope-list* used by VTreeMiner become exponential. More precisely, for each k-element combination of the vertices of the database tree, there exists an **occ** in the **occ-list** of the tree pattern, where TId is 0, the lower bound of *scope* is the preorder number of the vertex with the largest depth in the combination, the upper bound of *scope* is $n - 1$, RP is an array of size k filled by $n - 1$, and $multiplicity$ is

1.

In most of real-world databases, such as CSLOGS (Zaki, 2005b) and NASA (Chalmers et al., 2003), while several vertices of a database tree have the same label, no two vertices on the same path are identically labeled. For trees with this property, while worst case space complexity of **occ-list** becomes linear (with an efficient implementation of **occ-list** using linked lists), worst case size of *scope-list* remains exponential.

Time complexity. We study time complexity of leaf join and inner join:

- A leaf join between two **occ**s takes $O(d)$ time with d the length of the rightmost path of P. Since a pattern larger than 1 has $O(b \times |\mathcal{D}|)$ **occ**s and a pattern of size 1 has $O(n \times |\mathcal{D}|)$ **occ**s and leaf join is performed between every pairs of **occ**s with the same TId, worst case time complexity of leaf join between two **occ-list**s will be $O(d \times b \times n \times |\mathcal{D}|)$.
- In the inner join of the **occ-list**s of a tree pattern P' and a vertex v, given an **occ** ov of v, it takes $O(h \times d)$ time to find subsets of the **occ-list** of P' that satisfy the conditions of Proposition 13, where h is the number of **occ**s in the **occ-list** of P'. We note that **occ**s of an **occ-list** can be automatically sorted first based on their TIds and second, based on their RPs. This makes it possible to find the subsets of the **occ-list** of P' that satisfy the conditions of Proposition 13 only by one scan of the **occ-list** of P'. During this scan, it is checked whether: (i) the current and the previous **occ**s have the same TId, (ii) the current and the previous **occ**s have the same $RP[0], \ldots, RP[c]$, (iii) $RP[c+1]$ of the current **occ** is less than $ov.scope.l$, and (iv) $RP[c]$ of the current **occ** is greater than or equal to $ov.scope.l$. During the scan of **occ-list**(P'), after finding a subset S that satisfies the conditions of Proposition 13, it takes $O(d)$ time to perform the inner join of S and ov. As a result, it takes $O(h \times d)$ time to perform the inner join of the **occ-list** of P' and an **occ** of v. Since h is $O(b \times |\mathcal{D}|)$ and v has $O(n \times |\mathcal{D}|)$ **occ**s, and since inner join is done between every pairs of subsets S and **occ**s ov that have the same TIds, time complexity of inner join will be $O(d \times b \times n \times |\mathcal{D}|)$.

Therefore, frequency of a pattern can be counted in $O(d \times b \times n \times |\mathcal{D}|)$ time. We note that in VTreeMiner, frequency of a pattern is counted in $O(k \times e^2 \times |\mathcal{D}|)$ time (recall $k \geq d$ and $e \geq b$, in particular, it is possible that $e >> b$).

4.4.5 A brief comparison with other vertical frequency counting approaches

As mentioned before, algorithms such as VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008) need to keep track of all occurrences. Obviously, our approach is more efficient as it simultaneously represents and processes all occurrences sharing the rightmost path in $O(d)$ space, where d is the length of the rightmost path of the pattern. TreeMinerD developed by Zaki (2005b) for computing *per-tree frequency*, is more similar to our approach as it also processes rightmost paths of occurrences. However, there are significant differences. First, while TreeMinerD computes only *per-tree frequency*, our algorithm performs a much more expensive task and computes *per-occurrence frequency*. Second, TreeMinerD applies different join strategies.

4.5 TPMiner: an efficient algorithm for finding frequent embedded tree patterns

In this section, we first introduce the TPMiner algorithm and then, we discuss an optimization technique used to reduce the number of generated infrequent patterns.

4.5.1 The TPMiner algorithm

Having defined the operations of leaf join and inner join and having analyzed their properties, we can now introduce our tree pattern miner, called TPMiner (**T**ree **P**attern **Miner**). TPMiner builds all frequent patterns and maintains the rightmost paths of all their occurrences.

TPMiner follows a depth-first strategy for generating tree patterns. First, it extracts frequent patterns of size 1 (frequent vertex labels) and constructs their **occ-list**s. This step can be done by one scan of the database. Every **occ** of a pattern of size 1 represents one occurrence of the pattern, where its RP contains the upper bound of the scope of the occurrence, its $scope$ contains the scope of the occurrence, and its $multiplicity$ is 1. Then, larger patterns are generated using rightmost path extension. For every tree P of size $k+1$ ($k \geq 1$) which is generated by adding a vertex v to a vertex on the rightmost path of a tree P', the algorithm computes the **occ-list** of P by joining the **occ-list**s of P' and v. If v is added to the the rightmost vertex of P', a *leaf join* is performed; otherwise, an *inner join* is done. The high level pseudo code of TPMiner is given in Algorithm 6. $\mathcal{P}$ is used to store all frequent patterns. Every tree pattern is generated in $O(1)$ time, hence, time complexity of Algorithm 6 is $O(d \times b \times n \times |\mathcal{D}| \times \mathcal{C})$, where $\mathcal{C}$ is the number of generated candidates.

Algorithm 6 High level pseudo code of the TPMiner algorithm.

1: TPMiner
2: **Input:** $\mathcal{D}$ {a set of database trees}, $minsup$ {the minimum support threshold}
3: **Output:** $\mathcal{P}$ {the set of frequent patterns}
4: Compute the set $\mathcal{P}_1$ of frequent patterns of size 1 along with their **occ-list**s
5: $\mathcal{P} \leftarrow \mathcal{P}_1$
6: **for each** P in $\mathcal{P}_1$ **do**
7: Extend(P,$\mathcal{P}_1$, $minsup$, $\mathcal{P}$)
8: **end for**

1: Extend(P,$\mathcal{P}_1$, $minsup$, $\mathcal{P}$)
2: **Input:** P {a frequent pattern}, $\mathcal{P}_1$ {the set of frequent patterns of size 1}, $minsup$ {the minimum support threshold}
3: **Input and Output:** $\mathcal{P}$ {the set of frequent patterns}
4: **Side effect:** $\mathcal{P}$ is updated with frequent rightmost path extensions of P
5: **for each** P_1 in $\mathcal{P}_1$ **do**
6: {Let d be the length of the rightmost path of P}
7: **for** $i = 0$ **to** d **do**
8: {Let u be the vertex of P such that $rdep_P(u) = i$}
9: $P_n \leftarrow RExtend(P, P_1, u)$
10: **if** $i = d$ **then**
11: **occ-list**$(P_n) \leftarrow leaf_join(P, P_1)$
12: **else**
13: **occ-list**$(P_n) \leftarrow inner_join(P, P_1)$
14: **end if**
15: **if** $sup(P_n, \mathcal{D}) \geq minsup$ **then**
16: Add P_n to $\mathcal{P}$
17: Extend(P_n,$\mathcal{P}_1$, $minsup$, $\mathcal{P}$)
18: **end if**
19: **end for**
20: **end for**

4.5.2 An optimization technique for candidate generation

In rightmost path extension, a new tree P is generated by adding a new vertex to some vertex on the rightmost path of an existing frequent pattern P', therefore, it is already known that P has (at least) one frequent subtree. In an improved method proposed by Zaki (2005b), to generate the tree P, two patterns $P1$ and $P2$ such that their subtrees induced by all but the rightmost vertices are the same, are merged. In this merge, the rightmost vertex of $P2$ (which is not in $P1$) is added to $P1$ as the new vertex. In this way, we already know that P has (at least) two subtrees that are frequent, therefore, trees that have only one frequent subtree are not generated. This can reduce the number of trees that are generated but are infrequent.

4.6 Experimental Results

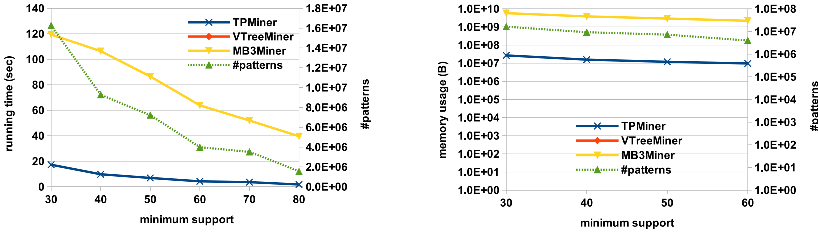
We performed extensive experiments to evaluate the efficiency of the proposed algorithm, using data from real applications as well as synthetic datasets. The experiments were done on a AMD Processor with 16 GB main memory and 2×1 MB L2 cache.

VTreeMiner (also called TreeMiner) (Zaki, 2005b) is a well-known algorithm for finding all frequent embedded patterns from trees. Therefore, we select this algorithm for our comparisons. Recently more efficient algorithms, such as TreeMinerD (Zaki, 2005b), XSpanner (Wang et al., 2004), TRIPS and TIDES (Tatikonda et al., 2006), have been proposed for finding frequent embedded tree patterns. However, they only compute the per-tree frequency instead of the per-occurrence frequency. Some other algorithms, such as (Xiao et al., 2003) and (Miyahara et al., 2004), find maximal embedded patterns which are a small subset of all frequent tree patterns. To the best of our knowledge at the time of writing this chapter, MB3Miner (Tan et al., 2008) is the most efficient recent algorithm for finding frequent embedded tree patterns. MB3Miner works with both per-tree frequency and per-occurrence frequency. Here, we use the version that works with the per-occurrence frequency. We note MB3Miner generates only the frequent patterns such that all subtrees are frequent, therefore, it might produce fewer frequent patterns than VTreeMiner and TPMiner. Tatikonda and Parthasarathy (2009) proposed efficient techniques for parallel mining of trees on multicore systems. Since we do not aim at parallel tree mining, their system is not proper for our comparisons.

We used several real-world datasets from different areas to evaluate the efficiency of TPMiner. The datasets do neither have noise nor missing values. Table 4.1 reports basic statistics of our real-world datasets.

Table 4.1: Summary of real-world datasets.

Dataset	# Transactions	# Vertices	# Vertex labels	Transaction size	
				Maximum	Average
CSLOGS32241	32,241	240,716	10,698	435	13.9323
Prions	17,551	227,203	111	37	24.9497
NASA	1,000	163,753	333	471	326.506



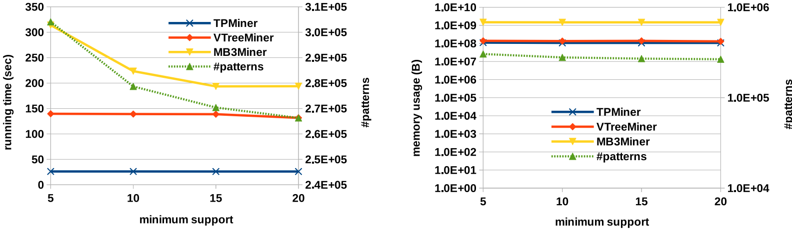
(a) The horizontal axis shows the minimum support; the left vertical axis the running time (sec) and the right vertical axis the number of patterns generated by TPMiner (and VTreeMiner).

(b) The horizontal axis shows the minimum support; the left vertical axis the memory usage (Byte) and the right vertical axis the number of patterns. The vertical axes are in logarithmic scale.

Figure 4.10: Comparison over CSLOGS32241.

The first real-world dataset is CSLOGS (Zaki, 2005b) that contains the web access trees of the CS department of the Rensselaer Polytechnic Institute. It has $59,691$ trees, $716,263$ vertices and $13,209$ vertex labels. Each distinct label corresponds to the URLs of a web page. As discussed by Tan et al. (2008), when *per-occurrence* frequency is used, none of the algorithms can find meaningful patterns, because by decreasing $minsup$, suddenly lots of frequent patterns with many occurrences appear in the dataset that makes the mining task practically intractable. Tan et al. (2008) progressively reduced the dataset and generated the CSLOGS32241 dataset that contains $32,241$ trees. Figure 4.10 compares the algorithms over CSLOGS32241. For all given values of $minsup$, VTreeMiner does not terminate within a reasonable time (i.e., 1 day!), therefore, the figure does not contain it. TPMiner is faster than MB3Miner by a factor of 5-20 and it requires significantly less memory cells. In order to have a comparison with VTreeMiner, we tested the algorithms for $minsup = 1000$; while TPMiner finds frequent patterns within around 1.3 seconds, VTreeMiner takes more than 1000 seconds to find the same patterns.

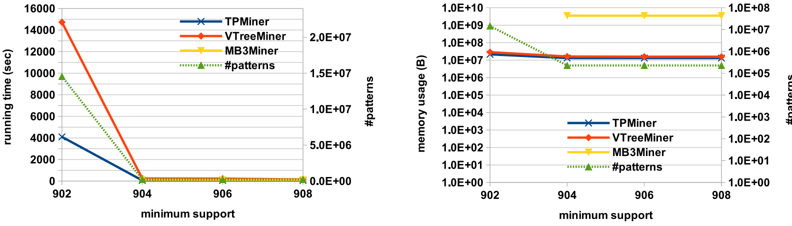
The second real-world dataset used in this chapter is Prions that describes a protein ontology database for Human Prion proteins in XML format (Sidhu et al., 2006). Tan et al. (2008) converted it into a tree-structured dataset by considering tags as



(a) Minimum support vs. running time.

(b) Minimum support vs. memory usage. The vertical axes are in logarithmic scale.

Figure 4.11: Comparison over Prions.



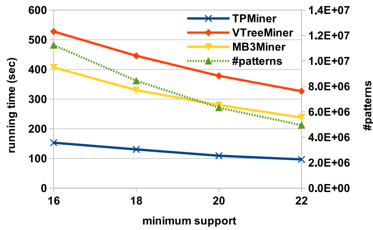
(a) Minimum support vs. running time.

(b) Minimum support vs. memory usage. The vertical axes are in logarithmic scale.

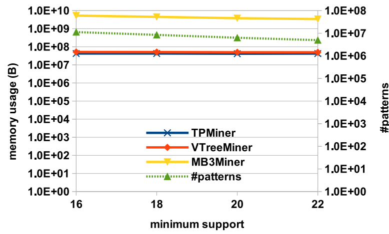
Figure 4.12: Comparison over NASA.

vertex labels. It has $17,551$ wide trees. Figure 4.11 reports the empirical results over this dataset, where TPMiner is faster than VTreeMiner by a factor of 5-5.2, and it is faster than MB3Miner by a factor of 7.3-11. The third real-world dataset is a dataset of IP multicast. The NASA dataset consists of MBONE multicast data that was measured during the NASA shuttle launch between the 14th and 21st of February, 1999 (Chalmers and Almeroth, 2001; Chalmers et al., 2003). It has 333 distinct vertex labels where each vertex label is an IP address. The data was sampled from this NASA dataset with 10 minutes sampling interval and has $1,000$ trees. In this dataset, large frequent patterns are found at high minimum support values. As depicted in Figure 4.12, over this dataset, TPMiner is 3-4 times faster than VTreeMiner and both methods are significantly faster than MB3Miner. At $minsup = 902$, MB3Miner fails.

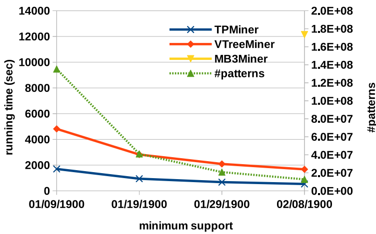
We also evaluated the efficiency of the proposed algorithm on several synthetic datasets generated by the method of Zaki (2005b). The synthetic data generation program mimics the web site browsing behavior of the user. First a master web site browsing tree is built and then the subtrees of the master tree are generated. The



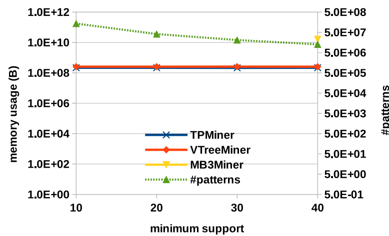
(a) D10: minimum support vs. running time.



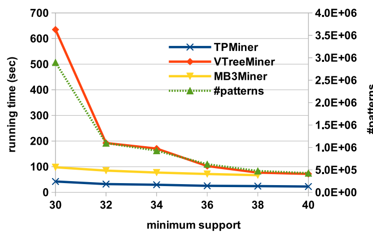
(b) D10: minimum support vs. memory usage. The vertical axes are in logarithmic scale.



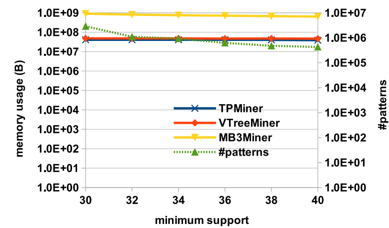
(c) D5: minimum support vs. running time.



(d) D5: minimum support vs. memory usage. The vertical axes are in logarithmic scale.



(e) M10K: minimum support vs. running time.



(f) M10K: minimum support vs. memory usage. The vertical axes are in logarithmic scale.

Figure 4.13: Comparison over synthetic datasets.

program is adjusted by 5 parameters: (i) the number of labels (N), (ii) the number of vertices in the master tree (M), (iii) the maximum fan-out of a vertex in the master tree (F), (iv) the maximum depth of the master tree (D), and (v) the total number of trees in the dataset (T). Figure 4.13 compares the algorithms over the synthetic datasets. The first synthetic dataset is D10 and uses the following default values for the parameters: $N = 100$, $M = 10,000$, $D = 10$, $F = 10$ and $T = 100,000$. Over this dataset, TPMiner is around 3 times faster than VTreeMiner and VTreeMiner is slightly faster than MB3Miner. The next synthetic dataset is D5, where D is set to 5 and for the other parameters, the default values are used. Over this dataset, at $minsup = 10$, 20 and 30, MB3Miner is aborted due to lack of memory. We also evaluated the effect of M. We set M to $100,000$ and used the default values for the other parameters and generated the M100k dataset. Over this dataset, TPMiner is faster than MB3Miner by a factor of 2-3, and both TPMiner and MB3Miner are significantly faster than VTreeMiner.

Discussion. Our extensive experiments report that TPMiner always outperforms well-known existing algorithms. Furthermore, there are several cases where TPMiner by order of magnitude is more efficient than any specific given algorithm. TPMiner and VTreeMiner require significantly less memory cells than MB3Miner. This is due to the different large data-structures used by MB3Miner such as the so-called EL, OC and VOL data structures and also to the breadth-first search (BFS) strategy followed by MB3Miner (Tan et al., 2008). Although TPMiner uses a more compact representation of occurrences than VTreeMiner, this is hardly noticeable in the charts. The reason is that the memory use is dominated by the storage of the frequent patterns.

In our experiments, we can distinguish two cases. First, over datasets such as NASA and D5 (in particular for low values of $minsup$), the Tree Model Guided technique used by MB3Miner does not significantly reduce the state space, therefore, the algorithm fails or it does not terminate within a reasonable time. In such cases, TPMiner find all frequent patterns very effectively. Second, over very large datasets (such as CSLOGS32241) or dense datasets (such as M100K) where patterns have many occurrences, TPMiner becomes faster than VTreeMiner by order of magnitude. This is due to the ability of TPMiner in frequency counting of patterns with many occurrences. As discussed earlier, the **occ** data-structure used by TPMiner can often represent and handle exponentially many occurrences with a single **occ** element, while in VTreeMiner these occurrences are represented and handled one by one.

4.7 Conclusion

In this chapter, we proposed an efficient algorithm for subtree homeomorphism with application to frequent pattern mining. We developed a compact data-structure, called **occ**, that effectively represents/encodes several occurrences of a tree pattern. We then defined efficient join operations on **occ** that help us to count occurrences of tree patterns according to occurrences of their proper subtrees. Based on the proposed subtree homeomorphism method, we introduced TPMiner, an effective algorithm for finding frequent tree patterns. We evaluated the efficiency of TPMiner on several real-world and synthetic datasets. Our extensive experiments show that TPMiner always outperforms well-known existing algorithms, and there are several situations where the improvement compared to existing algorithms is significant.

Acknowledgements

We are grateful to Professor Mohammed Javeed Zaki for providing the VTreeMiner code, the CSLOGS datasets and the TreeGenerator program, to Dr Henry Tan for providing the MB3Miner code, to Dr Fedja Hadzic for providing the Prions dataset and to Professor Jun-Hong Cui for providing the NASA dataset. Finally, we would like to thank Dr Morteza Haghir Chehreghani for his discussion and suggestions.

Chapter 5

An Efficient Algorithm for Approximate Betweenness Centrality Computation

5.1 Introduction

Centrality is a structural property of vertices in a network that measures the importance of a vertex within the network (Freeman, 1979). For example, it determines how important a person is within a social network, or how well-used a road is within a road network. *Betweenness centrality* of a vertex, introduced by Freeman, is defined as the number of shortest paths (geodesic paths) from all (source) vertices to all others that pass through that vertex. He used it as a measure for quantifying the control of a human over the communications among others in a social network (Freeman, 1977). Betweenness centrality is also used in some well-known algorithms for clustering and community detection in social and information networks. For example, the community detection algorithm proposed by Girvan and Newman (2002) iteratively partitions the network by finding edges with high betweenness centrality, removing them from the network and recomputing betweenness centrality of remaining edges.

Although there exist polynomial time and space algorithms for betweenness centrality computation, the algorithms are expensive in practice. Currently, the most efficient existing exact method is the algorithm of Brandes (2001). Time complexity of this algorithm is $O(nm)$ for unweighted graphs and $O(nm+n^2 \log n)$ for weighted graphs with positive weights (n is the number of vertices and m is the number of edges in

the network). Therefore, exact betweenness centrality computation is not practically applicable, even for mid-size networks.

Fortunately, in several applications it is required to only compute betweenness centrality of one vertex (or a few vertices). For instance, this index may be computed only for core vertices of communities in social/information networks (Wang et al., 2011), or hubs in communication networks. However, the next bad news is that computing exact betweenness centrality of a single vertex is not easier than computing betweenness centrality of all vertices. Therefore, the above mentioned complexities also hold if someone wants to compute betweenness centrality of only one vertex (or a few vertices).

As a simple example motivating betweenness centrality computation of only one vertex, consider Figure 5.1(a) that shows a toy *road network*. In a road network, vertices are intersections of roads and (undirected) edges are roads connecting these intersections. Suppose that the intersection presented by vertex 1 is very crowded and we want to change the structure of the network to reduce the traffic-jam in this intersection. Three new configurations are suggested:

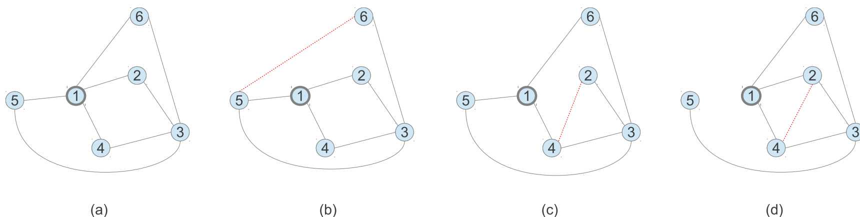
1. blocking the road between vertices 1 and 6 and building a new road between vertices 5 and 6 (Figure 5.1(b)),
2. blocking the road between vertices 1 and 2 and building a road between vertices 2 and 4 (Figure 5.1(c)), and
3. blocking the road between vertices 1 and 5 and building a new road between vertices 2 and 4 (Figure 5.1(d)).

We want to examine the effect of the suggested configurations on the traffic of the intersection 1 and choose the best one. In the existing road network which is depicted in Figure 5.1(a), betweenness score of vertex 1 is 6. In the networks of Figures 5.1(b), 5.1(c), and 5.1(d) it is 3, 3 and 2, respectively. Therefore, among the three suggested configurations, the one depicted in Figure 5.1(d) is the best one for reducing the traffic-jam of intersection 1. Note that real-world road networks can be very large. For example, the road network of California has $1,965,206$ vertices and $2,766,607$ edges (Leskovec et al., 2009).

To make computation of betweenness centrality tractable in practice, in recent years several algorithms have been proposed for approximate betweenness centrality computation (Brandes and Pich, 2007; Bader et al., 2007; Geisberger et al., 2008). Existing algorithms fall into one of the following categories.

1. Some algorithms such as the algorithms of Brandes and Pich (2007) and Geisberger et al. (2008) approximate betweenness centrality of all vertices in

Figure 5.1: Figure (a) shows a toy road network where betweenness score of vertex 1 is 6 and it is desired to reduce this score. Three new configurations are suggested, that are depicted in Figures (b), (c) and (d). Betweenness scores of vertex 1 in the networks of Figures (b), (c), and (d) are 3, 3 and 2, respectively.



the network. For these methods the value computed for every vertex is not of high importance, instead, the main goal is to correctly estimate the relative rank of all vertices.

2. Some others, such as the method of Bader et al. (2007), aim to approximate betweenness centrality of a single vertex (or a few vertices) in time faster than computing betweenness centrality of all vertices. For these methods, the accuracy of the estimated betweenness centrality is important.

Our focus in this chapter is the second category, i.e., we aim at developing an efficient and accurate algorithm for betweenness centrality computation of a single vertex (or a few vertices) in the network.

In this chapter, we propose a randomized algorithm for unbiased approximation of betweenness centrality. In the proposed algorithm, a source vertex i is selected by some strategy, single-source betweenness scores of all vertices on i are computed, and the scores are *scaled* as estimations of betweenness centralities. Our proposed algorithm can be adapted with different sampling techniques to give diverse methods for approximating betweenness centrality. As we will see later, some existing methods can be seen as special cases of our proposed algorithm adapted with particular samplings. We propose a condition that a promising sampling technique should satisfy to minimize the approximation error for a single vertex. Then, we present a sampling technique that fits better with the condition.

While the algorithm of Bader et al. (2007) is intuitively presented for high centrality vertices, in our method, the sampling technique can be optimized for both high centrality vertices and low centrality vertices. Finally, our proposed method can be used to compute similar centrality notions, such as *stress centrality* (Shimbel, 1953), that are also based on counting shortest paths.

We perform extensive experiments on synthetic networks as well as networks from real-world, and show that compared to existing exact and inexact algorithms, our method works with higher accuracy or gives significant speedups.

The rest of this chapter is organized as follows. A brief overview on related work is given in Section 5.2. In Section 5.3, we present a randomized algorithm for betweenness centrality computation. In Section 5.4, we discuss the sampling methods. We empirically evaluate the proposed method in Section 5.5 and show its efficiency and high accuracy. Finally, the chapter is concluded in Section 5.6.

5.2 Related work

Centrality measures defined for the vertices of a network, are an important and essential tool for the analysis of social networks. The widely used centrality indices include *betweenness centrality* (Freeman, 1977), *closeness centrality* (Sabidussi, 1966), *degree centrality* (Wasserman and Faust, 1994) and *eigenvector centrality* (Bonacich and Lloyd, 2001).

Betweenness centrality, which is widely used as a precise estimation of the information flow controlled by a vertex in social and information networks, assumes that information flow is done through shortest paths (Yan et al., 2006). Brandes (2001) introduced new algorithms for computing betweenness centrality of a vertex, which is performed in $O(nm)$ and $O(nm + n^2 \log n)$ times for unweighted and weighted networks, respectively.

Holme (2003) showed that betweenness centrality of a vertex is highly correlated with the fraction of time that the vertex is occupied by the traffic of the network. Barthelemy (2004) showed that many scale-free networks (Barabasi and Albert, 1999) have a power-law distribution of betweenness centrality. Borgatti (2005) proposed a dynamic model-based view of centrality that focuses on the outcomes of vertices in a graph. He said that here the fundamental questions are: *i*) how often does traffic flow through a vertex, and *ii*) how long do things take to get to a vertex.

Variants Newman (2005) proposed *random walk betweenness*, that prefers shorter paths over the longer ones. Goh et al. (2001) defined Load Centrality (LC), which is a variant of betweenness centrality. It assumes that traffic flows over shortest paths, but uses a different routing mechanism. Another set of variants is obtained by limiting the length of paths. It is based on the idea that very long paths are used rarely and should not contribute to betweenness of a vertex. Such measures are called k-*betweenness centrality*, where k is the maximum length of counted paths. Friedkin (1991) proposed a 2-betweenness centrality measure. Similarly, Gould and Fernandez

(1990) developed *brokerage measures* that are specific variants of 2-betweenness centrality. There are also several variants of betweenness centrality that are used to determine the structural prominence of web pages (Kleinberg, 1999) and (Brin et al., 1998).

Generalization to sets There are several applications where a centrality notion for sets of vertices is more useful (Everett and Borgatti, 1999). Everett and Borgatti (1999) defined *group betweenness centrality* as a natural extension of betweenness centrality for sets of vertices. Group betweenness centrality of a set is defined as the number of shortest paths passing through at least one of the vertices in the set (Everett and Borgatti, 1999). The other natural extension of betweenness centrality is *co-betweenness centrality*. Co-betweenness centrality is defined as the number of shortest paths passing through all vertices in the set. Kolaczyk et al. (2009) presented an $O(n^3)$ time algorithm for co-betweenness centrality computation of sets of size 2. Chehreghani (2014) presented efficient algorithms for co-betweenness centrality computation of any set or sequence of vertices in weighted and unweighted networks. For example, he showed that co-betweenness centrality of a set K of vertices can be computed in $O(nm - |K|m + n|K|\log|K| - |K|^2\log|K|)$ time in unweighted graphs.

Puzis et al. (2007a) proposed an $O(|K|^3)$ time algorithm for computing successive group betweenness centrality, where $|K|$ is the size of the set. Puzis et al. (2007b) presented two algorithms for finding *most prominent group*. A *most prominent group* of a network is a set vertices of minimum size, so that every shortest path in the network passes through at least one of the vertices in the set. The first algorithm is based on a heuristic search and the second one is based on iterative greedy choice of vertices. Dolev et al. (2010) defined the Routing Betweenness Centrality (RBC) measure and presented algorithms for computing RBC of single vertices in the network and algorithms for computing group RBC of sets or sequences of vertices.

Approximate algorithms Brandes and Pich (2007) proposed an approximate betweenness centrality computation algorithm based on selecting k vertices, computing dependency scores for them, and extrapolating dependency scores of the rest. In the method of Bader et al. (2007), some source vertices are selected uniformly at random, and their dependency scores are computed and scaled for all vertices. Their sampling technique is adaptive in the sense that the number of samples varies based on the betweenness score. Geisberger et al. (2008) presented a framework for approximate ranking of vertices based on their betweenness scores. In this method, the method for aggregating dependency scores changes so that vertices do not profit from being near the selected source vertices.

Lee et al. (2012) proposed an algorithm to efficiently update betweenness centralities of vertices in a graph, when the graph obtains a new edge. They tried to reduce the search space by finding a candidate set of vertices whose betweenness centralities can be updated. Then, they proposed a method to compute betweenness centralities using candidate vertices only.

5.3 Approximate betweenness centrality computation

Algorithm 7 shows the high level pseudo code of our proposed algorithm for approximate betweenness centrality computation. First the following probabilities are computed in accordance with the used/applied sampling method (see below)

$$p_1, p_2, \ldots, p_n > 0 \text{ such that } \sum_{i=1}^{n} p_i = 1 \tag{5.1}$$

Then, at every iteration t of the loop in Lines 8-15 of Algorithm 7:

- an $i \in \{1, \ldots, n\}$ is selected with probability p_i,
- the SPD rooted at i is computed,
- dependency score of vertex i on v, $\delta_{i\bullet}(v)$, is computed,
- $\frac{\delta_{i\bullet}(v)}{p_i}$ is the estimation of $BC(v)$ in iteration t.

The average of betweenness centralities estimated in different iterations is returned as the final estimation of the betweenness centrality.

Algorithm 7 estimates betweenness centrality of *all* vertices of the graph. The reason is that after forming the SPD rooted at a vertex i, in the worst case time complexity of computing dependency score of i on one vertex is the same as time complexity of computing dependency scores of i on all vertices. However, as we will see later in Section 5.4, probabilities $p_1, p_2, \ldots, p_n$ can be calculated in a way to minimize the approximation error of a specific vertex in the graph.

Lemma 1. *In Algorithm 7, for a vertex v we have*

$$\mathbf{E}(B[v]) = BC(v) \tag{5.2}$$

and

$$\mathbf{Var}(B[v]) = \frac{1}{T} \sum_{i=1}^{n} \frac{\delta_{i\bullet}(v)^2}{p_i} - \frac{BC(v)^2}{T} \tag{5.3}$$

Algorithm 7 High level pseudo code of the algorithm of approximate betweenness centrality computation.

```
1: APPROXIMATEBETWEENNESS
2: Require. A network (graph) G, the number of samples T.
3: Ensure. Betweenness centrality of vertices of G.
4: Compute probabilities p_1, ..., p_n
5: for each vertex v ∈ V(G) do
6:     B[v] ← 0
7: end for
8: for each t = 1 to T do
9:     Select a vertex i with probability p_i
10:    Form the SPD D rooted at i
11:    Compute dependency scores of vertex i on all vertices v
12:    for each vertex v ∈ V(G) do
13:        B[v] ← B[v] + δ_{i•}(v) / p_i
14:    end for
15: end for
16: for each i ∈ {1, ..., n} do
17:     B[i] ← B[i] / T
18: end for
19: return B
```

Proof. We have:

$$\mathbf{E}(B[v]) = \frac{T\sum_{i=1}^{n}\frac{p_i\delta_{i\bullet}(v)}{p_i}}{T} = BC(v)$$

and

$$\begin{aligned}\mathbf{Var}(B_t[v]) &= \mathbf{E}(B_t[v]^2) - \mathbf{E}(B_t[v])^2 \\ &= \sum_{i=1}^{n}\frac{\delta_{i\bullet}(v)^2}{p_i} - BC(v)^2\end{aligned}$$

where index t stands for the iteration t. Since $B[v]$ is the average of T independent copies of $B_t[v]$, we have

$$\mathbf{Var}(B[v]) = \frac{1}{T}\sum_{i=1}^{n}\frac{\delta_{i\bullet}(v)^2}{p_i} - \frac{BC(v)^2}{T} \tag{5.4}$$

□

For unweighted graphs, in every iteration of the loop in Lines 8-15 of Algorithm 7, forming the SPD rooted at i and computing dependency scores of i on all vertices takes $O(m)$ time. Other steps inside the loop can be performed in $O(1)$ time. This means that for unweighted graphs, if probabilities $p_1, p_2, \ldots, p_n$ are already known, time complexity of Algorithm 7 will be $O(Tm)$.

For weighted graphs with positive weights, in every iteration of the loop in Lines 8-15, it takes $O(m + n \log n)$ time to form the SPD rooted at i and $O(m)$ time to compute dependency scores of i on every vertex v. Therefore, for weighted graphs with positive weights, if probabilities $p_1, p_2, \ldots, p_n$ are already known, time complexity of Algorithm 7 will be $O(Tm + Tn \log n)$. For weighted graphs where negative weights are allowed, the problem is NP-Hard.

Algorithm 7 provides a randomized framework for approximate betweenness centrality computation, so that some existing algorithms can be described as adaptations of Algorithm 7 with specific sampling methods. For example, if vertices i are selected uniformly at random (i.e. $p_i = \frac{1}{n}$ for $1 \leq i \leq n$), then it will give the randomized algorithm presented by Bader et al. (2007). Note that instead of taking exactly T samples, we can define a *condition* for the termination of the loop in Lines 8-15 of Algorithm 7. For example, similar to the algorithm of Bader et al. (2007), our algorithm can be terminated when $B[v] \geq cn$ for some constant c.

5.4 Sampling methods

In this section, we discuss sampling methods, i.e. how probabilities $p_1, \ldots, p_n$ are computed. Suppose that we want to estimate betweenness centrality of a vertex v. We first present the optimal sampling which minimizes variance of $B[v]$. It might be computationally expensive to use the optimal sampling. Based on the optimal sampling, we present conditions that a promising sampling technique should satisfy, in order to give a better approximation of betweenness score of v. We then introduce a sampling technique, called *distance-based sampling*, that fits better with the mentioned condition.

5.4.1 Optimal sampling

Suppose that we want to estimate betweenness centrality of a vertex v. The following Lemma defines the probabilities minimizing variance of $B[v]$.

Lemma 2. *If in Algorithm 7 source vertices i are selected with probabilities*

$$p_i = \frac{\delta_{i\bullet}(v)}{\sum_{j=1}^{n} \delta_{j\bullet}(v)} \tag{5.5}$$

the approximation error (i.e., variance of $B[v]$) is minimized. In this case, variance of $B[v]$ will be 0*.*

Proof. In order to minimize $\mathbf{Var}(B[v])$, we need to minimize $\sum_{i=1}^{n} \frac{\delta_{i\bullet}(v)^2}{p_i}$, because other parts of $\mathbf{Var}(B[v])$ in Equation 5.4 are independent of i.

We define

$$f(p_1, \ldots, p_n) = \sum_{i=1}^{n} \frac{\delta_{i\bullet}(v)^2}{p_i}$$

and substitute p_n by $1 - \sum_{j=1}^{n-1} p_j$ and form equations $\frac{\partial f}{\partial p_i} = 0$, for $1 \leq i \leq n-1$.

We get

$$\frac{\delta_{i\bullet}(v)^2}{{p_i}^2} = \frac{\delta_{n\bullet}(v)^2}{\left(1 - \sum_{j=1}^{n-1} p_j\right)^2} \tag{5.6}$$

which gives:

$$p_i = \frac{1 - \sum_{j=1}^{n-1} p_j}{\delta_{n\bullet}(v)} \delta_{i\bullet}(v) = \frac{p_n}{\delta_{n\bullet}(v)} \delta_{i\bullet}(v) \tag{5.7}$$

Summing p_i's, for $1 \leq i \leq n-1$, and doing simplifications, we get

$$1 - p_n = \frac{p_n}{\delta_{n\bullet}} \sum_{i=1}^{n-1} \delta_{i\bullet}(v)$$

which gives

$$p_n = \delta_{n\bullet}(v) \sum_{i=1}^{n} \delta_{i\bullet}(v)$$

Plugging this in Equation 5.7 gives

$$p_i = \frac{\delta_{i\bullet}(v)}{\sum_{j=1}^{n} \delta_{j\bullet}(v)}$$

If we put this value of p_i into Equation 5.4, variance of $B[v]$ becomes 0. □

Hence in optimal sampling, p_i is proportional to the fraction of shortest paths passing through v that starts in i. Using probabilities p_i defined in Equation 5.5, gives an exact method in the sense that it makes the approximation error 0. However, time complexity of computing optimal p_i's is the same as exact betweenness centrality computation.

5.4.2 A property of promising sampling methods

Although it is not practically efficient to use probabilities p_i defined in Equation 5.5, they can help us to define the desired properties of sampling techniques.

Based on optimal sampling, when estimating betweenness centrality of a vertex v, we present the following as a property desired for a sampling technique:

$$\forall i, i' \in V(G) \setminus \{v\} : p_i < p_{i'} \Leftrightarrow \delta_{i\bullet}(v) < \delta_{i'\bullet}(v) \tag{5.8}$$

which means vertices with higher dependency scores on v, must be chosen as source vertices with a higher probability.

Then, the *quality* of a sampling technique with respect to a network can be defined in terms of the number of (unordered) pairs of vertices i and i' satisfying the above mentioned property, divided by $\frac{(n-1)(n-2)}{2}$ which is the number of subsets of size 2 of $V(G) \setminus \{v\}$. In other words, the quality of the sampling is the fraction of pairs for which Equation 5.8 holds.

What the property mentioned in Equation 5.8 suggests, somehow contradicts the source vertex selection procedure presented by Geisberger et al. (2008). In the method of Geisberger et al. (2008) the procedure for aggregating dependency scores changes so that vertices do not profit from being near the selected source vertices. However, Equation 5.8 says that it is better to select source vertices based on their dependency scores on v, and as we will see later, it may result in preferring the source vertices that are closer to v. The reason for this contradiction is that while we here aim at approximating betweenness centrality of *some specific vertex* v, the method of Geisberger et al. (2008) aims to rank *all vertices* based on their betweenness scores.

5.4.3 A new sampling technique

In this section, we present a new sampling technique that fits better with the property mentioned in Section 5.4.2. Suppose we want to estimate betweenness score of a vertex v. In our proposed sampling, every vertex $k \neq v$ is chosen as a source vertex with probability p_k defined as follows:

$$p_k = \frac{\frac{1}{d(k,v)}}{\sum_{j=1}^{n} \frac{1}{d(j,v)}} \tag{5.9}$$

i.e., p_k is proportional to the inverse of the distance between vertices k and v. Note that in Equation 5.9, $\sum_{k=1}^{n} p_k = 1$.

The rationale behind this sampling is as follows:

1. Let v, v' and i be three vertices such that $d(v,i) < d(v',i)$ and v and v' have (almost) the same betweenness score. Suppose v is the only ancestor of v' in the SPD rooted at i that has depth $d(i,v)$. Every shortest path from i to some vertex t' that passes through v', also passes through v, therefore, for each $t' \in V(G) \setminus \{i,v,v'\}$, we have: $\sigma_{it'}(v') \leq \sigma_{it'}(v)$ and hence

   $$\frac{\sigma_{it'}(v')}{\sigma_{it'}} \leq \frac{\sigma_{it'}(v)}{\sigma_{it'}}$$

   which yields

   $$\sum_{t' \in V(G) \setminus \{i,v,v'\}} \frac{\sigma_{it'}(v')}{\sigma_{it'}} \leq \frac{\sigma_{it'}(v)}{\sigma_{it'}} \tag{5.10}$$

   Furthermore, all shortest paths between i and v' pass through v, i.e.,

   $$\frac{\sigma_{iv'}(v)}{\sigma_{iv'}} = 1 \tag{5.11}$$

   Equations 5.10 and 5.11 yield $\delta_{i\bullet}(v) > \delta_{i\bullet}(v')$.

2. Consider a random ER graph $G = (n,p)$ that has n vertices and with probability p an edge is drawn independently at random between every two vertices. Recently, Agarwal et al. (2015) showed that when G is sparse, $\mathbf{E}(\delta_{i\bullet}(v)) \approx \mu c(i,d-1)(1+\mu c(i,d))$, where $d = d(i,v)$, $c(i,d)$ is the fraction of the expected number of vertices that have a distance larger than d from i and μ is the average degree of G. For two vertices i and i' in G such that $d(v,i) < d(v,i')$, $c(i,d(v,i))$ is greater than $c(i',d(v,i'))$ and hence, $\mathbf{E}(\delta_{i\bullet}(v)) \gtrapprox \mathbf{E}(\delta_{i'\bullet}(v))$. Note that in Equation 5.9, $d(v,i) < d(v,i')$ yields $p_i > p_{i'}$.

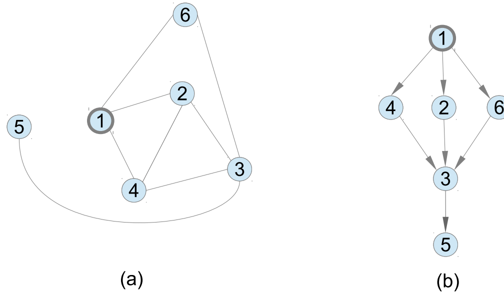
Now, lets investigate the *quality* of the proposed sampling method and compare it with the uniform sampling. Consider the network of Figure 5.1(d) and suppose that we want to estimate betweenness centrality of vertex 1. For convenience, we present the graph again in Figure 5.2 with its SPD rooted at vertex 1.

Dependency scores of other vertices on vertex 1 are:

vertex	dependency score
vertex 2	$\frac{1}{2}$
vertex 3	0
vertex 4	$\frac{1}{2}$
vertex 5	0
vertex 6	1

Probabilities p_k calculated for the vertices of the network using Equation 5.9 are

Figure 5.2: A graph (left) and its SPD rooted at vertex 1 (right).



vertex	probability
vertex 2	$\frac{6}{23}$
vertex 3	$\frac{3}{23}$
vertex 4	$\frac{6}{23}$
vertex 5	$\frac{2}{23}$
vertex 6	$\frac{6}{23}$

All pairs of vertices, except $\{6, 2\}$ and $\{6, 4\}$, satisfy the property of Equation 5.8. Therefore, the *quality* of our proposed sampling technique for this network is $\frac{8}{10}$. However, the uniform sampling assigns equal probabilities to all vertices. In this sampling, only pairs $\{2, 4\}$ and $\{3, 5\}$ satisfy the property of Equation 5.8. Hence, its *quality* is $\frac{2}{10}$.

A nice property of our proposed sampling technique is that it only requires to compute the distance between the vertex v and all other vertices in the graph: the *single-source shortest path*, or *SSSP* in short, problem. For unweighted graphs, this problem can be solved in $O(m)$ time and for weighted graphs with positive weights, it is solvable in $O(m+n \log n)$ time (Fredman and Tarjan, 1987). This means that using our proposed sampling technique will not increase time complexity of Algorithm 7. Therefore, with probabilities p_i defined in Equation 5.9, a vertex i is selected and dependency score of i on v is computed, and the result is scaled. For unweighted graphs, it gives an $O(Tm)$ time algorithm for approximate betweenness centrality computation. For weighted graphs (with positive weights), time complexity of the algorithm will be $O(Tm + Tn \log n)$.

5.5 Experimental results

We performed extensive experiments on both synthetic datasets and real-world networks to assess the quantitative and qualitative behavior of the proposed sampling technique. The experiments were done on a AMD Processor with 8 GB main memory and 2×1 MB L2 cache.

We compared our proposed method with the algorithm of Bader et al. (2007). As mentioned earlier, methods such as Brandes and Pich (2007) and Geisberger et al. (2008) aim to rank vertices based on betweenness scores (and the betweenness score of an individual vertex is not of high importance for them). Therefore, they are not suitable for our comparisons. We refer to the algorithm of Bader et al. (2007) as the *uniform sampling*, since it chooses source vertices uniformly at random, and to our proposed method as the *distance-based sampling*. We also compared the algorithms against the exact algorithm of Brandes (2001).

5.5.1 Datasets

For synthetic data, using the Barabasi-Albert (BA) model (Barabasi and Albert, 1999), we generated power-law graphs with degree distribution $p(k) \propto k^{-3}$. We generated networks of size $n \in \{10^3, 10^4\}$. We refer to the network of size 10^3 as BA10^3, and to the network of size 10^4 as BA10^4.

For real-world data, we used the DBLP co-authorship network, the Wiki-Vote social network, the Enron-Email communication network, the CA-CondMat collaboration network, and the CA-HepTh collaboration network.

DBLP This dataset is constructed from a snapshot of DBLP[1], that has yearly time granularity. Vertices represent authors and edges represent co-authorship relations. Two graph snapshots were extracted from two different periods: dblp0305 (from 2003 to 2005) and dblp0507 (from 2005 to 2007) (Berlingerio et al., 2009).

Wiki-Vote network This dataset contains all administrator elections and vote history data in Wikipedia[2], using the latest complete dump of Wikipedia page edit history (from January 3, 2008). It contains 2,794 elections, 103,663 votes and 7,066 users. A user either casts a vote or gets a vote. About half of the votes in the dataset are by the existing admins and the rest comes from Wikipedia users. Vertices of the

[1] Digital Bibliography and Library Project `http://www.informatik.uni-trier.de/~ley/db/`.
[2] `http://en.wikipedia.org/wiki/Main_Page`

Table 5.1: Summary of real-world networks.

Dataset	# vertices	# edges	Avg. degree
dblp-0305	$109,044$	$233,961$	4.29
dblp-0507	$135,116$	$290,363$	4.28
Enron-Email	$36,692$	$367,662$	20.04
Wiki-Vote	$7,115$	$103,689$	29.14
CA-CondMat	$23,133$	$93,497$	8.08
CA-HepTh	$9,877$	$25,998$	5.26

network represent wikipedia users and an edge from vertex v to vertex u represents that user v voted on user u (Leskovec et al., 2010).

Enron-Email network This email communication network contains all email communications within a dataset of email addresses. Vertices of the network are email addresses and if an address u sent at least one email to address v, the graph contains an undirected edge between u and v (Leskovec et al., 2009).

CA-CondMat network This network contains scientific collaborations among authors of papers submitted to the Condense Matter category. If an author v co-authored a paper with author u, the graph contains an undirected edge between v and u. The network covers papers in the period from January 1993 to April 2003 (124 months) (Leskovec et al., 2007).

CA-HepTh network This network contains scientific collaborations among authors of papers submitted to the High Energy Physics - Theory category. If an author v co-authored a paper with author u, the graph contains an undirected edge between v and u. The network covers papers in the period from January 1993 to April 2003 (Leskovec et al., 2007).

Table 5.1 summarizes specifications of our real-world networks.

5.5.2 Empirical results

For a vertex v, the empirical approximation error, reported in our experiments, is defined as follows:

$$err(v) = \frac{|App(v) - BC(v)|}{BC(v)} \times 100 \tag{5.12}$$

where $App(v)$ is the calculated approximate betweenness score.

In our experiments, we consider several vertices of a dataset and for every vertex, we compute distance-based probabilities, exact betweenness centrality and approximate betweenness scores using distance-based and uniform samplings. Table 5.2 summarizes the average results (i.e. the sum of the results obtained for different vertices divided by their number) obtained for different datasets.

Figure 5.3 plots approximation errors of the uniform and distance-based samplings for different vertices in the $\text{BA}10^3$ dataset. In the plots of this section, we order vertices by the difference in the error rate of their distance based sampling and uniform sampling. For most vertices, distance-based sampling gives a better approximation. As depicted in Table 5.2, the average approximation error for distance-based sampling is 41.77%, while it is 56.13% for the uniform sampling. The extra time needed by the distance-based sampling to compute required shortest path distances is quite tiny and ignorable compared to the running time of the whole process. For example, for different vertices of $\text{BA}10^3$ it is always less than 0.2. In all experiments, for both uniform and distance-based samplings, the number of samples is 10% of the number of vertices in the network. Therefore, the running time of the approximate methods is around 10% of the running time of the exact method.

Table 5.2: Comparing average approximation error and average running time of uniform sampling, distance-based sampling, and the Brandes exact method, for various single vertices in different datasets.

Database	Exact BC		Approximate BC				
	Avg. BC score	Avg. time (sec)	Distance-based sampling		Uniform sampling	Avg. time (sec)	# iterations
			Avg. error (%)	Avg. dist. comp. time (sec)	Avg. error (%)		
$BA10^3$	3503.83	7.87	**41.77%**	≤ 0.2	**56.13%**	0.79	100
$BA10^4$	358789.85	743.49	**16.54%**	≤ 0.5	**29.79%**	78.33	1000
Wiki Vote	76056.85	515.09	**37.0%**	≤ 0.5	**41.13%**	46.05	711
Email-Enron	2775100.8	9033.11	**15.75%**	≤ 0.5	**25.28%**	925.80	3,669
dblp0305	564246.41	19149.8	**7.59%**	≤ 1.5	**64.73%**	1747.15	10,904
dblp0507	798125.00	35140	**7.19%**	≤ 1.5	**50.17%**	2863.82	13,511
CA-CondMat	691667	3026.9	**10.8%**	≤ 1	**20.81%**	315.3	2,313
CA-HepTh	23747.85	341.8	**26.99%**	≤ 0.4	**32.18%**	35.14	987

Figure 5.3: A comparison between approximation errors of uniform sampling and distance-based sampling for 57 different vertices in the BA10^3 dataset.

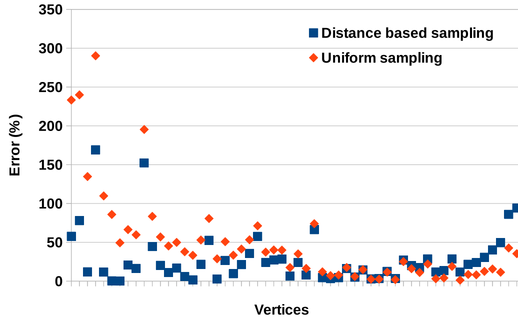


Figure 5.4 compares the methods on the BA10^4 dataset. Over this dataset, the average error of distance-based sampling is 16.54%, while it is 29.79% for the uniform sampling. We note that since the number of iterations is a fixed ratio (10%) of the network size, we have more iterations over larger datasets. This increase in the number of iterations might reduce the approximation error over large datasets, as we see for BA10^3 vs. BA10^4.

To further study the quality of approximations, we test the methods on real-world datasets. Figure 5.5 reports the results obtained for Wiki-Vote. It is a very dense dataset (its average degree is 29.14). For most vertices of the Wiki-Vote network, distance-based sampling gives a better approximation. The next real-world dataset is Email-Enron. It is less dense than Wiki-Vote, but still a dense graph. As reported

Figure 5.4: A comparison between approximation errors of uniform sampling and distance-based sampling for 155 different vertices in the BA10^4 dataset.

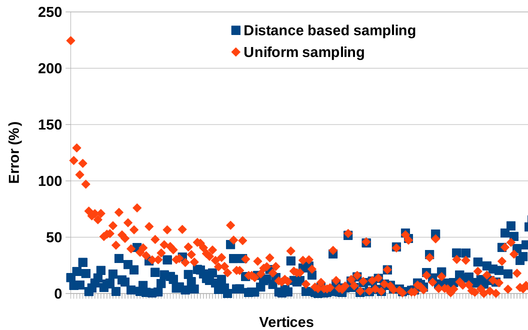


Figure 5.5: A comparison between approximation errors of uniform sampling and distance-based sampling for 89 different vertices in the Wiki-Vote dataset.

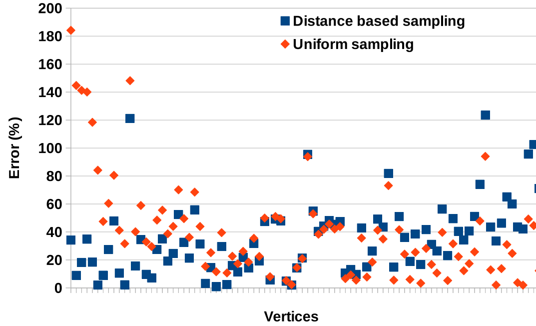
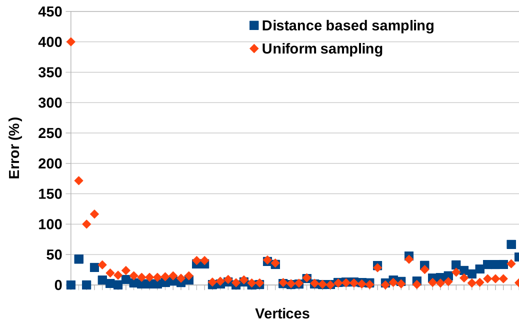


Figure 5.6: A comparison between approximation errors of uniform sampling and distance-based sampling for 118 different vertices in the Email-Enron dataset.

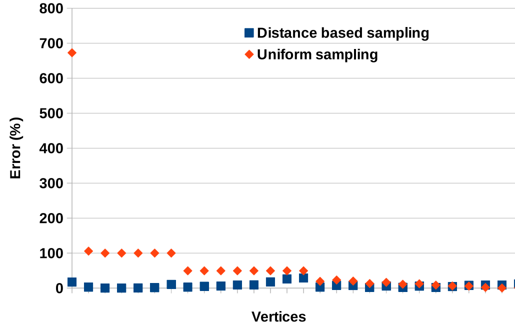


in Figure 5.6, for most vertices of Email-Enron, the approximation error of distance-based sampling is better than the uniform sampling.

Dblp0305 and dblp0507 are large and relatively sparse datasets. As reflected in Figures 5.7 and 5.8 and Table 5.2, over these datasets, distance-based sampling works much better than uniform sampling. This means that on sparse networks, the difference between quality of two methods is more substantial. It has several reasons. The first reason is that in very dense datasets, many vertices have the same (and small) distance from vertex v (v is the vertex whose betweenness centrality is estimated). Therefore, distance-based sampling becomes closer to the uniform sampling.

The second reason is that in sparse networks, in the SPD rooted at i, the probability that a vertex v' has only one ancestor at some level k is lower than this probability in dense

Figure 5.7: A comparison between approximation errors of uniform sampling and distance-based sampling for 28 different vertices in the dblp0305 dataset.



graphs. Figure 5.11 compares these two situations. It means that in sparse networks, distance-based sampling is closer to the optimal sampling, because by distance-based sampling, a larger number of vertices will satisfy the condition expressed in Equation 5.5. As a result, over sparse networks, distance-based sampling becomes much more effective than uniform sampling. Fortunately, most of real-world networks are sparse.

Then, the methods are compared on the CA-CondMat dataset which contains scientific collaborations between authors of papers submitted to Condense Matter category (Leskovec et al., 2007). The average degree in this dataset is 8.08. It is denser than dblp0305 and dblp0507, but less dense than Wiki-Vote and Email-Enron. Over this dataset, the approximation error of uniform sampling is almost twice the approximation error of distance-based sampling.

Finally, to study the behavior of the methods on small datasets, we use the CA-HepTh network. It has only 9877 vertices and its average degree is 5.26. As depicted in Figure 5.10, for most vertices of this network, distance-base sampling gives a better approximation than the uniform sampling. The average approximation error of uniform sampling is higher than the average approximation error of distance-based sampling and the time required to compute distance-based probabilities is always less than 0.4.

5.6 Conclusion

In this chapter, we presented a randomized algorithm for unbiased approximation of betweenness centrality. In the proposed algorithm, a source vertex i is selected by some strategy, single-source betweenness scores of all vertices on i are computed, and

Figure 5.8: A comparison between approximation errors of uniform sampling and distance-based sampling for 9 different vertices in the dblp0507 dataset.

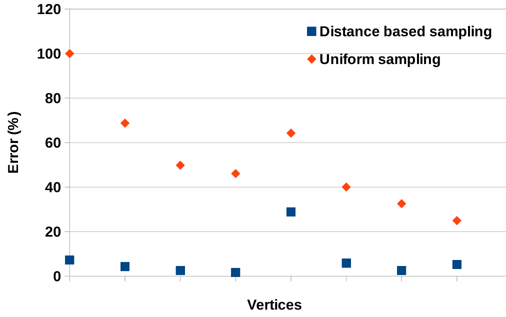


Figure 5.9: A comparison between approximation errors of uniform sampling and distance-based sampling for different vertices in the CA-CondMat dataset.

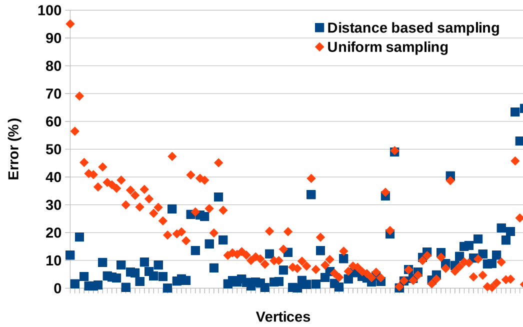


Figure 5.10: A comparison between approximation errors of uniform sampling and distance-based sampling for different vertices in the CA-HepTh dataset.

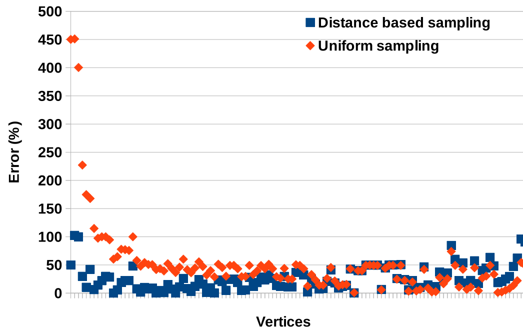
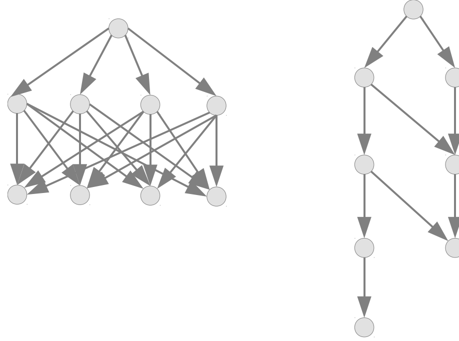


Figure 5.11: In sparse graphs, distance-based sampling is closer to optimal sampling. The graph in the left side shows an SPD in a dense graph, and the graph in the right side shows an SPD in a sparse graph.



the scores are *scaled* as estimations of betweenness scores. Our proposed algorithm can be adapted with different sampling techniques to give diverse betweenness estimation methods. Some existing methods can be seen as special cases of the proposed algorithm adapted with particular samplings.

We discussed the conditions that a promising sampling technique should satisfy to minimize the approximation error and proposed a sampling technique that fits better with the conditions. We performed extensive experiments on synthetic networks as well as networks from real-world and showed the high efficiency and quality of our proposed method.

Chapter 6

Conclusion and Future Work

In this chapter, we present a summary of our key contributions and provide directions for future work.

6.1 Summary of main contributions

The main goal of this dissertation was to improve state of the art algorithms in graph mining. The contributions can be summarized as follows.

Single network mining under subgraph homomorphism. In this work, we studied the problem of single network mining under subgraph homomorphism for a class of patterns more general than rooted trees. We introduced the class of *rooted graph patterns* and presented a method for complete generation of rooted graphs. We defined the notion of *height* of a rooted pattern and proposed an algorithm for generating height-k-bounded core rooted patterns (for some given integer k). We also introduced a new data structure for compact representation of all frequent rooted patterns and showed that it gives a closure operator for the rooted patterns that are frequent under subgraph homomorphism. Finally, we presented the HoPa algorithm for finding frequent core rooted patterns. We empirically evaluated HoPa on different synthetic and real-world networks and showed its high efficiency. In particular, by restricting our patterns to rooted trees, we compared HoPa against htreeminer (Dries and Nijssen, 2012) and showed that there are several cases where htreeminer fails (due to lack of memory) or it does not terminate within a reasonable time (e.g., 3-4 days), but HoPa finds frequent patterns effectively.

Mining rooted ordered trees under subtree homeomorphism. In this work, we introduced a novel algorithm for subtree homeomorphism of rooted ordered trees. It uses a more compact data-structure, called **occ**, for representing occurrences, and a more efficient subtree homeomorphism algorithm based on Dietz's numbering scheme Dietz (1982). An **occ** data-structure can represent/encode all occurrences that have the rightmost path in common. The number of such occurrences can be exponential, even though the size of the **occ** is only $O(d)$, where d is the length of the rightmost path of the tree pattern. We presented efficient join operations on **occ** that help us to efficiently calculate the occurrence count of tree patterns from the occurrence count of their proper subtrees.

We observed that in most of widely used real-world databases, while many vertices of a database tree have the same label, no two vertices on the same path are identically labeled. For this class of database trees, worst case space complexity of our algorithm is linear; a result comparable to the best existing results for per-tree frequency. For such databases worst case space complexity of the well-known existing algorithms for per-occurrence frequency, such as VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008), is still exponential. Based on the proposed subtree homeomorphism method, we developed an efficient pattern mining algorithm, called TPMiner. To evaluate the efficiency of TPMiner, we performed extensive experiments on both real-world and synthetic datasets. Our results showed that TPMiner always outperforms most efficient existing algorithms such as VTreeMiner (Zaki, 2005b) and MB3Miner (Tan et al., 2008). Furthermore, there were several cases where the improvement of TPMiner with respect to existing algorithms was significant.

Approximate Betweenness Centrality Computation. In this work, we proposed a randomized algorithm for unbiased approximation of betweenness centrality. In the proposed framework, a source vertex i is selected by some strategy, single-source betweenness scores of all vertices on i are computed, and the scores are *scaled* as estimations of betweenness centralities. Our proposed algorithm can be adapted with different sampling techniques to give diverse methods for approximating betweenness centrality. We discussed the conditions that a promising sampling technique should satisfy to minimize the approximation error for a single vertex. Then, we proposed a sampling technique that fits better with the conditions. Finally, we performed extensive experiments on synthetic networks as well as networks from real-world, and showed that compared to existing algorithms, our proposed method works with a higher accuracy.

6.2 Future work

Graph mining is a hot and challenging research area that has many interesting directions for future research. In this section, we discuss some future work related to the content of this dissertation.

Applications of frequent patterns Finding frequent patterns is useful; but only a first step in graph mining. A next step can be to investigate usefulness of frequent patterns in different applications, such as social and information networks, the world wide web and XML documents; as well as in other data mining tasks such as clustering and classification. For example, it is interesting to investigate if rooted patterns extracted by the HoPa algorithm (Chapter 3) can be used to characterize very large real-world networks. Furthermore, it is also useful to study the utilization of the found patterns in predictive models. For example, frequent patterns can be used in rule-based predictive models to classify structured data such as graphs (Deshpande et al., 2005) and trees (Zaki and Aggarwal, 2006). Here the arising questions are: what patterns should be used to form predictive rules? how these rules should be ordered? etc.

Interestingness of frequent patterns Defining proper interestingness predicates for patterns is a challenging and non-trivial problem. It is well-known that using only support as the interestingness measure is not proper (Kontonasios and Bie, 2010) and several alternatives have been proposed such as the product of the size with the support (Geerts et al., 2004), the ability to compress a database (Siebes et al., 2006), partial support (Poernomo and Gopalkrishnan, 2009) and non-drivable pattern (Calders and Goethals, 2007). However, these measures are mostly restricted to simple classes such as itemsets and in some cases, they are extensible to sequences. Hence, it is an interesting research direction to develop interestingness measures for complex pattern classes such as trees and graphs. These measures can help us to find richer patterns that reflect more aspects of the database. Afterwards, a next step would be to develop efficient algorithms for finding patterns that are interesting based on the proposed measures.

Bibliography

Agarwal, M., Singh, R. R., Chaudhary, S., and Iyengar, S. R. S. (2015). An efficient estimation of a node's betweenness. In Mangioni, G., Simini, F., Uzzo, S. M., and Wang, D., editors, *Complex Networks VI - Proceedings of the 6th Workshop on Complex Networks CompleNet 2015, New York City, USA, March 25-27, 2015*, volume 597 of *Studies in Computational Intelligence*, pages 111–121. Springer.

Arnborg, S., Corneil, D. G., and Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *SIAM J. Alo. Disc. Meth.*, 8(2):277–284.

Asai, T., Abe, K., Kawasoe, S., Arimura, H., Satamoto, H., and Arikawa, S. (2002). Efficient substructure discovery from large semi-structured data. In *Proceedings of the Second SIAM International Conference on Data Mining (SDM)*, pages 158–174. SIAM.

Asai, T., Arimura, H., Uno, T., and Nakano, S. (2003). Discovering frequent substructures in large unordered trees. In *Proceedings of the 6th International Conference on Discovery Science (DS)*, pages 47–61.

Bader, D. A., Kintali, S., Madduri, K., and Mihail, M. (2007). Approximating betweenness centrality. In *Proceedings of 5th International Conference on Algorithms and Models for the Web-Graph (WAW)*, pages 124–137.

Barabasi, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286:509–512.

Barthelemy, M. (2004). Betweenness centrality in large complex networks. *The Europ. Phys. J. B - Condensed Matter*, 38(2):163–168.

Berlingerio, M., Bonchi, F., Bringmann, B., and Gionis, A. (2009). Mining graph evolution rules. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 115–130.

Bille, P. and Gortz, I. (2011). The tree inclusion problem: In linear space and faster. *ACM Transactions on Algorithms*, 7(3):1–47.

Bonacich, P. and Lloyd, P. (2001). Eigenvector-like measures of centrality for asymmetric relations. *Social Networks*, 23(3):191–201.

Borgatti, S. P. (2005). Centrality and network flow. *Social Networks*, 27(1):55–71.

Borgelt, C. and Berthold, M. R. (2002). Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. of IEEE International Conference on Data Mining (ICDM)*, pages 51–58.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177.

Brandes, U. and Pich, C. (2007). Centrality estimation in large networks. *Intl. Journal of Bifurcation and Chaos*, 17(7):303–318.

Brin, S., Motwani, R., Page, L., and Winograd, T. (1998). What can you do with a web in your pocket? *IEEE Bulletin of the Technical Committee on Data Engineering*, 21(2):37–47.

Bringmann, B. and Nijssen, S. (2007). What is frequent in a single graph? In *Proceedings of 5th International Workshop on Mining and Learning with Graphs (MLG)*, pages 1–4.

Calders, T. and Goethals, B. (2007). Non-derivable itemset mining. *Data Min. Knowl. Discov.*, 14(1):171–206.

Calders, T., Ramon, J., and Dyck, D. V. (2011). All normalized anti-monotonic overlap graph measures are bounded. *Data Min. Knowl. Discov.*, 23(3):503–548.

Chalmers, R. and Almeroth, K. (2001). Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM)*, pages 449–458.

Chalmers, R. C., Member, S., and Almeroth, K. C. (2003). On the topology of multicast trees. *IEEE/ACM Transactions on Networking*, 11:153–165.

Chehreghani, M. H. (2011). Efficiently mining unordered trees. In *Proceedings of the 11th IEEE International Conference on Data Mining (ICDM)*, pages 111–120.

Chehreghani, M. H. (2014). Effective co-betweenness centrality computation. In *Seventh ACM International Conference on Web Search and Data Mining (WSDM)*, pages 423–432.

Chehreghani, M. H., Chehreghani, M. H., Lucas, C., and Rahgozar, M. (2011). OInduced: an efficient algorithm for mining induced patterns from rooted ordered trees. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 41(5):1013–1025.

Chi, Y., Muntz, R. R., Nijssen, S., and Kok, J. N. (2005a). Frequent subtree mining - an overview. *Fundamenta Informaticae*, 66(1-2):161–198.

Chi, Y., Xia, Y., Yang, Y., and Muntz, R. R. (2005b). Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):190–202.

Chi, Y., Yang, Y., and Muntz, R. (2004a). HybridTreeMiner: an efficient algorihtm for mining frequent rooted trees and free trees using canonical forms. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 11–20.

Chi, Y., Yang, Y., and Muntz, R. R. (2003). Indexing and mining free trees. In *Proceedings of the Third IEEE International Conference on Data Mining (ICDM)*, pages 509–512.

Chi, Y., Yang, Y., Xia, Y., and Muntz, R. R. (2004b). CMTreeMiner: Mining both closed and maximal frequent subtrees. In *Proceedings of the 8th Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 63–73.

Cook, D. J., Holder, L. B., and Djoko, S. (1995). Knowledge discovery from structural data. *J. Intell. Inf. Syst.*, 5(3):229–248.

Cordella, L., Foggia, P., Sansone, C., and Vento, M. (2001). An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159.

Cordella, L., Foggia, P., Sansone, C., and Vento, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1367–1372.

Cui, J., Kim, J., Maggiorini, D., Boussetta, K., and Gerla, M. (2002). Aggregated multicast - a comparative study. In *Proceedings of the Second International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications (NETWORKING)*, pages 1032–1044.

Daenen, J. (2009). Isomorfvrije generatie van een contextvrije, geconnecteerde graaftaal met begrensde graad. In *Technical report, Universiteit Hasselt, In Dutch, available from the author*.

Dehaspe, L. and Toivonen, H. (1999). Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36.

Deshpande, M., Kuramochi, M., Wale, N., , and Karypis, G. (2005). Frequent sub-structure based approaches for classifying chemical compounds. *IEEE Trans. Knowl. Data Eng.*, 17(8):1036–1050.

Diestel, R. (2005). *Graph Theory*. Springer.

Diestel, R. (2010). *Graph Theory, 4th ed.* Springer-Verlag, Heidelberg.

Dietz, P. F. (1982). Maintaining order in a linked list. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 122–127.

Dolev, S., Elovici, Y., and Puzis, R. (2010). Routing betweenness centrality. *J. ACM*, 57(4):1–27.

Dries, A. and Nijssen, S. (2012). Mining patterns in networks using homomorphism. In *Proceedings of the Twelfth SIAM International Conference on Data Mining (SDM)*, pages 260–271.

Everett, M. and Borgatti, S. (1999). The centrality of groups and classes. *Journal of Mathematical Sociology*, 23(3):181–201.

Fannes, T., Kibriya, A., Grave, K. D., and Ramon, J. (2015). MIPS: A graph mining library. In *ICML Workshop on Machine Learning Open Source Software (MLOSS)*.

Fomin, F. V., Lokshtanov, D., Raman, V., Saurabh, S., and Rao, B. V. R. (2012). Faster algorithms for finding and counting subgraphs. *J. Comput. Syst. Sci.*, 78(3):698–706.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.

Freeman, L. C. (1977). A set of measures of centrality based upon betweenness, sociometry. *Social Networks*, 40:35–41.

Freeman, L. C. (1979). Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239.

Friedkin, N. E. (1991). Theoretical foundations for centrality measures. *American Journal of Sociology*, 96(6):1478–1504.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Co.

Garriga, G. C., Khardon, R., and De Raedt, L. (2007). On mining closed sets in multi-relational data. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 804–809.

Geerts, F., Goethals, B., and Mielikainen, T. (2004). Tiling databases. In *Proceedings of the 7th International Conference on Discovery Science (DS)*, pages 278–289.

Geisberger, R., Sanders, P., and Schultes, D. (2008). Better approximation of betweenness centrality. In *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 90–100.

Ghazizadeh, S. and Chawathe, S. S. (2002). Seus: Structure extraction using summaries. In *Proceedings of the 5th International Conference on Discovery Science (DS)*, pages 71–85.

Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Natl. Acad. Sci. USA*, 99:7821–7826.

Gjoka, M., Kurant, M., Butts, C., and Markopoulou, A. (2010). Walking in facebook: a case study of unbiased sampling of osns. In *Proceedings of 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 2498–2506.

Goh, K. I., Kahng, B., and Kim, D. (2001). Universal behavior of load distribution in scale-free networks. *Phys. Rev. Lett.*, 87(27):278701.

Gould, V. R. and Fernandez, R. M. (1990). Structures of mediation: A formal approach to brokerage in transaction networks. *Sociological Methodology*, 19:89–126.

Hajiaghayi, M. and Nishimura, N. (2007). Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. *J. Comput. Syst. Sci.*, 73(5):755–768.

Hale, S. A. (2014). Multilinguals and wikipedia editing. *arXiv:1312.0976*.

Halin, R. (1976). S-functions for graphs. *Journal of Geometry*, 8:171–186.

Han, J. and Kamber, M. (2000). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Hasan, M. A. and Zaki, M. J. (2009). Output space sampling for graph patterns. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):730–741.

Holme, P. (2003). Congestion and centrality in traffic flow on complex networks. *Adv. Complex. Syst.*, 6(2):163–176.

Inokuchi, A., Washio, T., and Motoda, H. (2000). An Apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 13–23.

Inokuchi, A., Washio, T., Nishimura, Y., and Motoda, H. (2002). A fast algorithm for mining frequent connected subgraphs. *IBM Research Report*, RT0448.

Ivancsy, R. and Vajk, I. (2006). Frequent pattern mining in web log data. *Acta Polytechnica Hungarica*, 3(1):77–90.

Kibriya, A. M. and Ramon, J. (2012). Nearly exact mining of frequent trees in large networks. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 426–441.

Kilpelainen, P. and Mannila, H. (1995). Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24(2):340–356.

Kleinberg, J. M. (1999). Group-authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632.

Kolaczyk, E. D., Chua, D. B., and Barthelemy, M. (2009). Group-betweenness and co-betweenness: Inter-related notions of coalition centrality. *Social Networks*, 31(3):190–203.

Kontonasios, K.-N. and Bie, T. D. (2010). An information-theoretic approach to finding informative noisy tiles in binary databases. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, pages 153–164.

Koutis, I. and Williams, R. (2009). Limits and applications of group algebras for parameterized problems. In *36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 653–664.

Kuramochi, M. and Karypis, G. (2001). Frequent subgraph discovery. In *Proceedings of the First IEEE International Conference on Data Mining*, pages 313–320.

Kuramochi, M. and Karypis, G. (2004). Grew: A scalable frequent subgraph discovery algorithm. In *Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM)*, pages 439–442.

Kuramochi, M. and Karypis, G. (2005). Finding frequent patterns in a large sparse graph. *Data Min. Knowl. Discov.*, 11:243–271.

LaPaugh, A. S. and Rivest, R. L. (1980). The subgraph homeomorphism problem. *J. Comput. Syst. Sci.*, 20(2):133–149.

Lee, M. J., Lee, J., Park, J. Y., Choi, R. H., and Chung, C.-W. (2012). Qube: A quick algorithm for updating betweenness centrality. In *Proceedings of the 21st World Wide Web Conference (WWW)*, pages 351–360.

Leskovec, J., Huttenlocher, D., and Kleinberg, J. (2010). Signed networks in social media. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI)*, pages 1361–1370.

Leskovec, J., Kleinberg, J., and Faloutsos, C. (2007). Graph evolution: densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1).

Leskovec, J., Lang, K., Dasgupta, A., and Mahoney, M. (2009). Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123.

Matousek, J. and Thomas, R. (1992a). On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, 108:343–364.

Matousek, J. and Thomas, R. (1992b). On the complexity of finding iso-and other morphisms for partial k-trees. *Discrete Mathematics*, 108:343–364.

Miyahara, T., Shoudai, T., Uchida, T., Takahashi, K., Ueda, H., Cheung, D., Williams, J. G., and Qing, L. (2001). Discovery of frequent tree structured patterns in semistructured web documents. In *Proceedings of Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 47–52.

Miyahara, T., Suzuki, Y., Shoudai, T., Uchida, T., Takahashi, K., and Ueda, H. (2004). Discovery of maximally frequent tag tree patterns with contractible variables from semistructured documents. In *Proceedings of the 8th Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 133–144.

Moens, M., Li, J., and Chua, T., editors (2014). *Mining User Generated Content*. Chapman and Hall/CRC.

Muggleton, S. and Raedt, L. D. (1994). Inductive logic programming: theory and methods. *J. Log. Program.*, 19/20:629–679.

Newman, M. E. J. (2005). A measure of betweenness centrality based on random walks. *Social Networks*, 27(1):39–54.

Nienhuys-Cheng, S.-H. and De Wolf, R. (1997). *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*. Springer-Verlag, New York, NY, USA.

Nijssen, S. and Kok, J. (2010). There is no optimal, theta-subsumption based refinement operator. *Personal communication*.

Nijssen, S. and Kok, J. N. (2003). Efficient discovery of frequent unordered trees. In *Proceedings of the First International Workshop on Mining Graphs, Trees, and Sequences (MGTS)*, pages 55–64.

Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416.

Poernomo, A. K. and Gopalkrishnan, V. (2009). Efficient computation of partial-support for mining interesting itemsets. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, pages 1014–1025.

Puzis, R., Elovici, Y., and Dolev, S. (2007a). Fast algorithm for successive computation of group betweenness centrality. *Phys. Rev. E*, 76(5):056709.

Puzis, R., Elovici, Y., and Dolev, S. (2007b). Finding the most prominent group in complex networks. *AI Commun.*, 20(4):287–296.

Qin, L., Yu, J. X., and Ding, B. (2007). TwigList: Make twig pattern matching fast. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 850–862.

Raedt, L. D. and Dehaspe, L. (1997). Clausal discovery. *Machine Learning*, 26(2-3):99–146.

Ramon, J., Roy, S., and Jonny, D. (2011). Efficient homomorphism-free enumeration of conjunctive queries. In *International Conference on Inductive Logic Programming (ILP)-Preliminary Papers*.

Robertson, N. and Seymour, P. (1986). Graph minors. II: Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322.

Sabidussi, G. (1966). The centrality index of a graph. *Psychometrika*, 31:581–60.

Shimbel, A. (1953). Structural parameters of communication networks. *Bulletin of Mathematical Biophysics*, 15:501–507.

Sidhu, A. S., Dillon, T. S., and Chang, E. (2006). Protein ontology. In Ma, Z. and Chen, J. Y., editors, *Database Modeling in Biology: Practices and Challenges*, pages 39–60. Springer-Verlag.

Siebes, A., Vreeken, J., and Leeuwen, M. V. (2006). Item sets that compress. In *Proceedings of the Sixth SIAM International Conference on Data Mining (SDM)*, pages 395–406.

Simovici, D. A. and Djeraba, C. (2014). *Mathematical Tools for Data Mining - Set Theory, Partial Orders, Combinatorics. Second Edition.* Advanced Information and Knowledge Processing. Springer.

Tan, H., Hadzic, F., Dillon, T. S., Chang, E., and Feng, L. (2008). Tree model guided candidate generation for mining frequent subtrees from XML documents. *ACM Transaction on Knowledge Discovery from Data (TKDD)*, 2(2):43 pages.

Tatikonda, S. and Parthasarathy, S. (2009). Mining tree-structured data on multicore systems. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):694–705.

Tatikonda, S., Parthasarathy, S., and Kurc, T. M. (2006). TRIPS and TIDES: new algorithms for tree mining. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 455–464.

Ullmann, J. (1976). An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach. (JACM)*, 23(1):31–42.

Vanetik, N., Gudes, E., and Shimony, S. (2002). Computing frequent graph patterns from semistructured data. In *Proceedings of the Second IEEE International Conference on Data Mining (ICDM)*, pages 458–465.

Wang, C., Hong, M., Pei, J., Zhou, H., Wang, W., and Shi, B. (2004). Efficient pattern-growth methods for frequent tree pattern mining. In *Proceedings of the 8th Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 441–451.

Wang, K. and Liu, H. (1998). Discovering typical structures of documents: A road map approach. In *Proceedings of the 21st ACM SIGIR International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 146–154.

Wang, K. and Liu, H. (2000). Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):353–371.

Wang, Y., Di, Z., and Fan, Y. (2011). Identifying and characterizing nodes important to community structure using the spectrum of the graph. *PLoS ONE*, 6(11):e27418.

Wang, Y. and Ramon, J. (2012). An efficiently computable support measure for frequent subgraph pattern mining. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 362–377.

Wasserman, S. and Faust, K. (1994). *Network Analysis: Methods and Applications*. Cambridge University Press.

Xiao, Y., Yao, J. F., Li, Z., and Dunham, M. H. (2003). Efficient data mining for maximal frequent subtrees. In *Proceedings of the Third IEEE International Conference on Data Mining (ICDM)*, pages 379–386.

Xiao, Y., Yao, J. F., and Yang, G. (2005). Discovering frequent embedded subtree patterns from large databases of unordered labeled trees. *International Journal of Data Warehousing and Mining*, 1(2):70–92.

Yan, G., Zhou, T., Hu, B., Fu, Z. Q., and Wang, B.-H. (2006). Efficient routing on complex networks. *Phys. Rev. E*, 73:046108.

Yan, X. and Han, J. (2002). gSpan: Graph-based substructure pattern mining. In *Proceedings of the Second IEEE International Conference on Data Mining (ICDM)*, pages 721–724.

Yan, X. and Han, J. (2003). CloseGraph: Mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 286–295.

Zaki, M. J. (2002). Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71–80.

Zaki, M. J. (2005a). Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52.

Zaki, M. J. (2005b). Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transaction on Knowledge and Data Engineering*, 17(8):1021–1035.

Zaki, M. J. and Aggarwal, C. C. (2006). XRules: An effective algorithm for structural classification of XML data. *Machine Learning*, 62(1-2):137–170.

Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., and Yu, P. S. (2011). Mining top-k large structural patterns in a massive network. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):807–818.

Selected Papers

Book chapters

- Jan Ramon, Constantin Comendant, Mostafa Haghir Chehreghani and Yuyi Wang, *Graph and network pattern mining, Mining of user generated content and its applications*, Boca Raton, Florida: CRC Press, 2014.

Journal articles

- Mostafa Haghir Chehreghani and Maurice Bruynooghe, *Mining rooted ordered trees under subtree homeomorphism*, Data Mining and Knowledge Discovery journal (DMKD), in press, DOI: 10.1007/s10618-015-0439-5. This paper will also be presented in the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECML-PKDD), Riva del Garda, Italy, September 19-23, 2016.
- Mostafa Haghir Chehreghani, *An efficient algorithm for approximate betweenness centrality computation*, Computer Journal (Comp. J.), Oxford University press, 57(9), 1371-1382, 2014.
- Mostafa Haghir Chehreghani, Jan Ramon and Maurice Bruynooghe, *Single network mining under homomorphism*, under submission.

Conference papers

- Mostafa Haghir Chehreghani, *Effective co-betweenness centrality computation*, 7th ACM Conference on Web Search and Data Mining (WSDM), 2014.

- Mostafa Haghir Chehreghani, Jan Ramon and Thomas Fannes, *Mining large networks under homomorphism*, Dutch-Belgian Database Day (DBDBD), Rotterdam, the Netherlands, 2013.

- Mostafa Haghir Chehreghani, *An efficient algorithm for approximate betweenness centrality computation*, 22nd ACM International Conference on Information and Knowledge Management (CIKM), 2013.

- Mostafa Haghir Chehreghani, *Efficiently mining unordered trees*, IEEE International Conference on Data Mining (ICDM), 2011, Vancouver, Canada.

- Jan Ramon and Mostafa Haghir Chehreghani, *On the complexity of listing closed frequent subgraph patterns*, 8th French Conference on Combinatorics (FCC), Paris, France, 2010.

Curriculum

Mostafa H. Chehreghani obtained his Master degree in Computer Engineering - Software in 2007 from University of Tehran, Iran. From 2007 to 2010, he worked as a programmer and part-time researcher in Iran. Then, he started his PhD studies at Machine Learning group of DTAI lab of KU Leuven, under the supervision of Prof. Maurice Bruynooghe and Dr Jan Ramon. The title of his PhD studies was "a theory-based study of graph mining". His research interests include data mining, machine learning and networked data analysis. He has published papers in conferences such as ACM WSDM, IEEE ICDM, ACM CIKM and journals such as DMKD, Comp. J., IEEE TSMC, CI, DKE, DSS and IDA. He has served as the reviewer for scientific journals such as Physics Letters A (PLA) and Computer Journal (Comp. J.).

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
MACHINE LEARNING RESEARCH GROUP
Celestijnenlaan 200A box 2402
B-3001 Leuven
Mostafa.HaghirChehreghani@cs.kuleuven.be
https://dtai.cs.kuleuven.be/ml/