KNOWLEDGE DISCOVERY FROM TREE DATABASES USING BALANCED OPTIMAL SEARCH

Israt Jahan Chowdhury

Bachelor of Science (Hons.) in Engineering

Bangladesh University of Engineering and Technology (BUET), 2009

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy (PhD)

School of Electrical Engineering and Computer Science

Science and Engineering Faculty

Queensland University of Technology

2016

Keywords

Trees, Rooted Unordered Trees, Free Trees, Root Node, Sibling Order, Ancestor-descendant Relationship, Tree Traversing Algorithm, Canonical Form, Adjacency Matrix, Optimisation, Frequent Subtree Mining, Induced Subtree, Embedded Subtree, Isomorphism, Candidate Generation, Tree Structure Guided Enumeration Tree, Enumeration Operations, Support, Frequency Counting, Tree Matching, Pair-wise Similarity Measure, Similarity Metric, Computational Complexity, Accuracy, Clustering, Bill of Materials (BOMs), Web Log Data (CSLOGS).

Abstract

Various domains such as Web, XML, Bioinformatics, Computer Networks and Manufacturing commonly represent their data in tree structures. Trees have become a formal means of studying link-based structures present in these domains. The tree structured data can appear in many forms such as rooted labelled ordered trees, unordered trees and free trees due to enforced structural constraints. The structural flexibility of unordered tree data allows extracting additional interesting information with practical significance, but at the same time, enforces complexities like exponential increase of runtime and memory usage.

An important problem in the knowledge discovery of labelled unordered trees is to find frequently occurring subtrees, thus facilitating data understanding. Another important problem is pairwise tree matching–a fundamental core operation of many data manipulation tasks such as clustering, data integration, and data querying. This thesis proposes efficient methods for solving these two problems. The main contributions of this thesis are three-fold.

An efficient tree representation serves as a basic block for further tree manipulation. Firstly, the Balanced-Optimal-Search (BOS) traversal, a novel traversing algorithm for trees, which can define an optimal order for any rooted labelled trees, is introduced. Utilising this optimal order, canonical forms, named **B**alanced **O**ptimal **C**anonical **F**orms (BOCFs) for labelled rooted unordered trees and free trees are defined. BOCF uniquely represents a rooted unordered tree or a free tree, which helps deal with isomorphic trees in tree processing. Two matrix representations of unordered trees are proposed—**A**ugmented **A**djacency **M**atrix (AAM) and **E**xtended **A**ugmented **A**djacency **M**atrix (EAAM); these will capture more structural information than the traditional adjacency matrix. These matrix representations ensure the unique identity of an unordered tree (one-to-one mapping).

Secondly, a tree matching algorithm is proposed for measuring similarity between rooted unordered tree pairs with two variations, one based on the AAM representation and another on the EAAM representation. This algorithm ensures faster similarity computation by comparing the matrices using a cosine similarity measure without compromising the accuracy. The similarity information can be embedded further in a clustering algorithm for grouping the tree datasets.

Thirdly, based on the BOCFs, frequent tree mining algorithms are developed that can effectively deal with the isomorphism problem–a pressing issue in frequent subtree mining. The **B**alanced **O**ptimal **S**earch **T**ree min**ER** algorithm (BOSTER) proposes a tree structure guided scheme-based enumeration to generate only valid candidate subtrees for mining frequent induced unordered subtrees. The **B**alanced optimal search **E**mbedded **S**ub**T**ree mining algorithm (BEST) generates candidate subtrees through the tree structure guided scheme-based enumeration approach with modified enumerate operation to find frequent embedded unordered trees. The Frequent **Free S**ubtree algorithm (FreeS) mines all frequent free induced subtrees using the tree structure guided scheme-based enumeration approach subject to constraint on supporting the generation of candidate trees in the canonical form of free trees.

Empirical analysis for the tree matching algorithm shows that the runtime reduces drastically without compromising the accuracy of output. The baseline algorithms show exponential complexity after reaching a tree size in the range of 60~65 nodes while the proposed method yields the runtime of less than a second. The performance of each frequent subtree mining algorithm is also evaluated using extensive empirical analysis and is compared with the state-of-the-art algorithms using both synthetic and real life data. In general, the runtime and memory usage of each algorithm has reduced a few orders of magnitude than the benchmarks without missing any frequent subtree.

This thesis contributes towards the process of knowledge discovery from tree databases by focusing on alleviating the hurdles of existing tree representation methods. The BOS-based representation plays an important role in significantly improving the scalability performance of tree matching and frequent subtree mining algorithms.

Dedication

To my parents Md. Nazmul Hossain Chowdhury and Chowdhury Khaleda Akter, my husband Dr. Md Nayim Kabir, my sister Dr. Fatama Akter Chowdhury, my parents in law Md Humayun Kabir and Farida Yeasmin, as well as to my other family members.

Table of Contents

List of Figures

List of Tables

List of Publications

Chowdhury, Israt J. & Nayak, Richi (2013) A novel method for finding similarities between unordered trees using matrix data model. Lecture Notes in Computer Science: The 14th International Conference on Web Information Systems Engineering – WISE 2013, LNAI 8180, pp. 421-430. (Will form part of Chapter 4)

Chowdhury, Israt J. & Nayak, Richi (2014) Identifying product families using data mining techniques in manufacturing paradigm. In Nayak, Richi, Li, Xue, Liu, Lin, Ong, Kok-Leong, Zhao, Yanchang, & Kennedy, Paul (Eds.) Australasian Data Mining Conference (AusDM), 2014, Brisbane, Australia (Will form part of Chapter 4)

Chowdhury, Israt J. & Nayak, Richi (2014) BOSTER: an efficient algorithm for mining frequent unordered induced subtrees. Lecture Notes in Computer Science: The 15th International Conference on Web Information Systems Engineering – WISE 2014, LNAI 8786, pp. 146-155 (Will form part of Chapter 5)

Chowdhury, Israt Jahan & Nayak, Richi (2014) BEST: an efficient algorithm for mining frequent unordered embedded subtrees. Lecture Notes in Computer Science: The 13th Pacific Rim International Conference on Artificial Intelligence– PRICAI 2014, LNAI 8862, pp. 459-471 (Will form part of Chapter 5)

Chowdhury, Israt J. & Nayak, Richi (2015) FreeS: Fast Algorithm to Discover Frequent Free Subtrees Using a Novel Canonical Form. Lecture Notes in Computer Science: The 16th International Conference on Web Information Systems Engineering – WISE 2015, LNAI 9418, pp. 123–137 (Will form part of Chapter 5)

Chowdhury, Israt J. & Nayak, Richi, "Measuring Similarity between Unordered Trees with the Balanced-Optimal-Search Traversal Algorithm", Knowledge and Information Systems, KAIS (Under Review) (Will form part of Chapter 4)

List of Abbreviations

AAM	Augmented Adjacency Matrix
BOS	Balanced Optimal Search
BFS	Breadth-first Search
BOCF	Balanced Optimal Canonical Form
BFCF	Breadth-first Canonical Form
BOM	Bill of Material
BOSTER	Balanced Optimal Search Tree Miner
BEST	Balanced Optimal Search Embedded Subtree Miner
BPM	Business Process Management
B	Breadth
CF	Canonical Form
CE	CliqueEdit
DFS	Depth-first Search
DFCF	Depth-first Canonical Form
D	Depth
DCE	DpCliqueEdit
EAAM	Extended Augmented Adjacency Matrix
FreeS	Frequent Free Subtree
FP	False Positive
FN	False Negative
F	FScore
GT	Group Technology
GBOM	Generic Bill of Material
HBT	HybridTreeMiner
LOGML	Log Markup Language
NMI	Normalised Mutual Information
OR	Operations Research
OL	Occurrence List
PTAS	Polynomial Time Approximation Scheme
P	Precision

R	Recall
RHS	Right Hand Side
SALB	Simple Assembly Line Balancing
SNA	Social Network Analysis
TSM	Tensor Space Model
TP	True Positive
UB	Upper Bound
UCE	UwCliqueEdit
VSM	Vector Space Model
XML	eXtensible Markup Language

Statement of Contribution of Co-Authors for Thesis by Published Paper

The following is the format for the required declaration provided at the start of any thesis chapter which includes a co-authored publication.

The authors listed below have certified* that:

1. They meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;
2. They take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;
3. There are no other authors of the publication according to these criteria;
4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit, and

5. They agree to the use of the publication in the student's thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this thesis, Chapter 4 and Chapter 5 are formed based on the publications. The publication title that are used in the thesis chapters are detailed in the list of publications with their statuses:

Contributor	Statement of contribution*
Israt Jahan Chowdhury Signature QUT Verified Signature Date:15/09/2015	Developed the methods, wrote the manuscripts, conducted experimental design and implementation, carried out experiments, and data analysis.
A/Prof Richi Nayak	Supervised in ideation phase for developing the methods, aided the experimental design, data analysis and manuscript writing.

Principal Supervisor Confirmation

I have sighted email or other correspondence from all Co-authors confirming their certifying authorship.

QUT Verified Signature

A/Prof Richi Nayak_		____15/09/2015__
Name	Signature	Date

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

QUT Verified Signature

Signature:

Date: _02/01/2016

Acknowledgements

I like to start the acknowledgement list by expressing my gratitude towards the Almighty, the Lord of everything.

My PhD journey in Queensland University of Technology (QUT) would never be possible without the immense support and guidance of various personalities. At first, I gratefully acknowledge the financial support provided by QUT (QUT PRA) during my whole candidature to make the carried research a success. My foremost gratitude goes towards my principle supervisor Associate Professor Richi Nayak for the continuous support, motivation and technical contribution throughout my candidature. She really knows how to nurture a young researcher like me. The time spent working with my principal supervisor has greatly shaped my professional identity and effectively made me the researcher I am today. I also like to express my gratitude towards associate supervisor Dr. Azharul Karim for his support throughout my PhD journey.

I am thankful for my fellow research mates from data science discipline for their support in creating a productive research environment. My special thanks to Dr. Sangeetha Kutty and Paul Westall for always giving valuable advice and time to help me when I was in need. Also I want to name some of my friends Ifa, Jun, Daniel, Gavin and Reza for supporting me in many circumstances. Special gratitude goes to QUT High Performance Computing (HPC) team, (especially Adam) for their support while trouble shooting any problem with HPC service.

Further, I acknowledge the staff members from EECS school especially Ms. Ellainne Steele, Ms. Joanne Reaves and Ms. Jo Kelly for their administrative support in managing travel applications and so on. Finally I like to thank all academic and non-academic staff for the support given to me in a countless number of ways.

I want to express my appreciations towards Bangladesh University of Engineering and Technology (BUET) for giving me high quality education which gave me a solid background of knowledge for perusing higher study.

My husband for his love, support and patience, who not only stayed beside me as a shelter but also worked as a mentor in various phases of my PhD journey.

Furthermore, I can’t express enough how much grateful I am to my family for their unconditional love in every possible aspects of my life.

Last, but by no means the least, I pay my respect to the valiant freedom fighters, 3 million martyrs for their sacrifices and 200 thousand women for sacrificing their innocence during the liberation war of Bangladesh—my motherland. Through their supreme sacrifices Bangladesh earned her desired victory.

Chapter 1: Introduction

Chapter 1 provides an introduction to the field of knowledge discovery from tree databases and describes the motivation behind the research. Following on from this, specific research questions are presented to address the research aims and guide the investigation through certain objectives. Given a separate list of research contributions, a high level overview of the thesis is shown through a relational map, which illustrates how the research progress has been carried out through linking the contributions. This is a thesis by publication, where the peer reviewed publications are directly used as chapters in comprising the greater part of the thesis. Details of the publications are provided, along with a brief summary of how each paper contributes to the thesis. A comprehensive introduction (i.e., preamble) of each paper is presented at the start of each chapter.

1.1 BACKGROUND

Knowledge discovery is a nontrivial process for extracting implicit, unknown and potentially useful information like patterns, rules, constraints, regularities and various relationships from a large set of data [1, 2]. Some other terms like data mining, data archaeology, data dredging, and data analytics have been used interchangeably in various reports, and have a similar or slightly different meaning. A general architecture of the knowledge discovery process is provided in Figure 1.1, where it can be seen that the journey of this process starts from the relevant data in databases and ends by extracting interesting knowledge and high level information as it passes through several stages. This process is considered as a rich and authentic way to generate and confirm knowledge, and therefore, has been recognised as a key research topic by many researchers from database systems, artificial intelligence, knowledge-based systems, knowledge acquisition and machine learning [3-5]. Moreover, an increasing interest has developed in the fields of business analysis, marketing management and industrial companies, where knowledge discovery is treated as an important area, which can potentially create opportunities for major revenues [6-8].

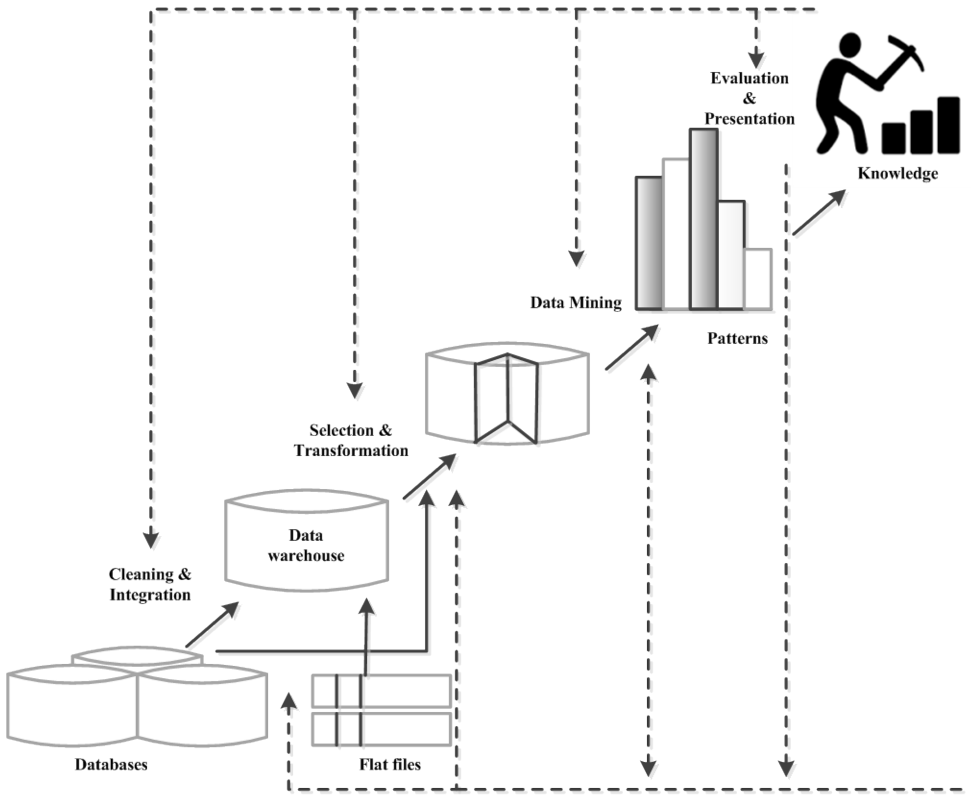


Figure 1.1: Architecture of knowledge discovery [5, 9]

In this era, the popularity of knowledge discovery has risen with the explosive growth in data, due to an easy access of internet and advanced storage capacity. Now, the knowledge discovery is no longer a random process but it has become a necessity for gaining insights and extracting information from the vast amount of data. Besides its enormous volume, data has become more complex in structure, with many interconnections and hierarchical dependencies [10, 11]. There is a need for developing new techniques that can deal with intricacy and volume of data to advance the current progress in the area of semi-structured data [10-15].

Trees are one of the most common data with complex structures [11, 12, 15-17]. Tree data have strong representational and expressive power for naturally capturing topological and relational characteristics embedded within a dataset. Tree structures therefore have become the de-facto standard for representing information with hierarchical dependencies [11, 18]. The dominance of tree data is noticeable in various applications, such as, XML and Weblogs in Web intelligence [19, 20]; DNA and Glycan in bioinformatics [21, 22]; Bill of Material (BOM) documents in manufacturing [23, 24]; and Phylogenetic trees in evolutionary science [25, 26].

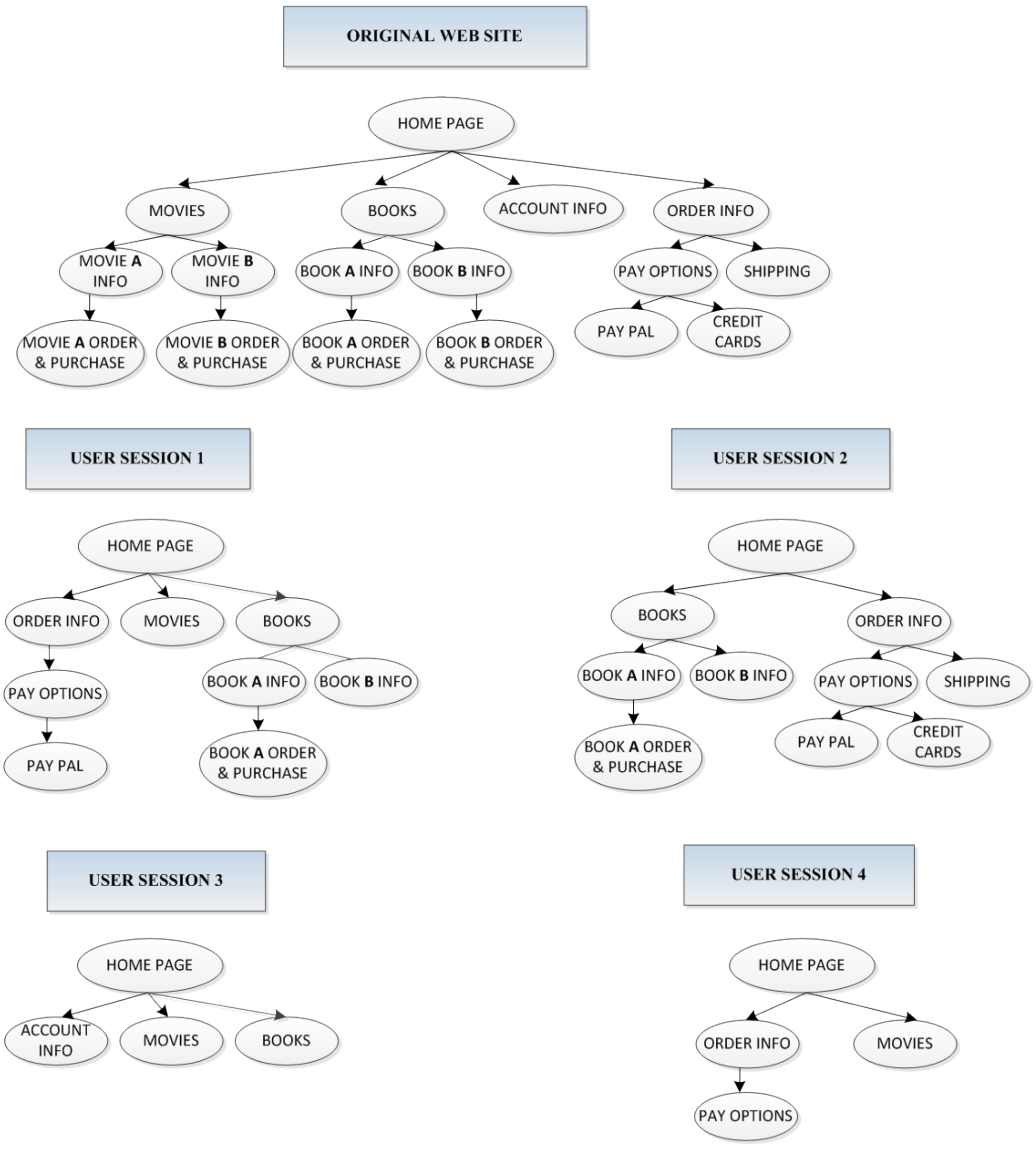


Figure 1.2: Example of a simple Web site and a fragment of Web usage patterns

Tree data can appear in several forms such as free tree, rooted unordered tree and rooted ordered tree, based on how nodes are represented in a tree. A rooted ordered tree preserves specific left-to-right order among the sibling nodes, whereas, a rooted unordered tree does not have any fixed order among the nodes except the ancestor-descendant order/relations. A free tree is unordered as well as unrooted, i.e., no root node is specified and has no sibling order. All these trees are usually found as labelled in real-life application, where the labels are attached to their nodes and edges (the formal definitions of all of these terms such as sibling, ancestor, and descendant nodes are provided in Chapter 2).

Ordered trees are well-studied in the area of knowledge discovery. On the contrary, the area of unordered and free tree mining has been understudied due to the complexities involved with the flexible structures. Therefore, knowledge discovery from unordered and free tree databases has become of interest and is the focus in this thesis. This thesis presents novel methods for representing rooted unordered and free trees that are utilised to propose novel frequent subtree mining and tree matching algorithms. These algorithms discover knowledge in the forms of frequent patterns and similarity information by ensuring less processing complexity.

1.2 MOTIVATION

With computing storage getting cheaper, heterogeneous data sources are rising. A tree model, especially of unordered nature, is robust to the data inconsistency and irregularity that a heterogeneous data source usually possesses [27, 28]. Consequently, it becomes enticing to use unordered tree structure models to represent this type of data. Moreover the progression of Web technology causes swift changes in online information that is better portrayed through an unordered tree model [29-31]. The common tree data (e.g., XML, Weblog\ Log Markup Language) for representing and exchanging information are treated as unordered in various database applications as a more reliable information transfer in comparison to the form of ordered trees [29, 32]. The following two examples are used to show the superiority of unordered tree models in data representation compared to their counterpart, ordered trees.

Figure 1.2, presents a simplified structure of a Web site that sells movies and books. The Web content is represented through LOGML, which is a Web log representation in XML template [33]. The LOGML documents can be modelled as trees. Each node in the tree corresponds to a Web page in the Web site. The interactions with the Web site are illustrated through user sessions following the Web site structure where the sample trees show the visit of Web pages from left to right. An interesting and useful information for site managers will be knowing how many times a set of Web pages (in the sub-tree form) have been accessed under the home Web page. This information can be useful in improving the site design. In this scenario, the order in which the set of Web pages were visited is irrelevant. The only information of interest is the set of Web pages, not the ones that are visited in the same order. Imposing the order may treat a frequently visited set of Web pages as non-frequent. For a given threshold in a frequent subtree mining process, only the Web pages that are visited in the same order will be extracted as frequent if the trees are considered as ordered. The same set of Webpages, browsed through different sessions in different order, will not be treated as the same during the process of ordered subtree mining. Hence, in order to extract the frequent subtrees, trees should be represented as unordered. For example, in Figure 1.2, the ‘Books’ and ‘Order Info’ Web pages were accessed in both user sessions 1 and 2, but in different order. This user behaviour should be shown as frequent information regardless of the visiting order.

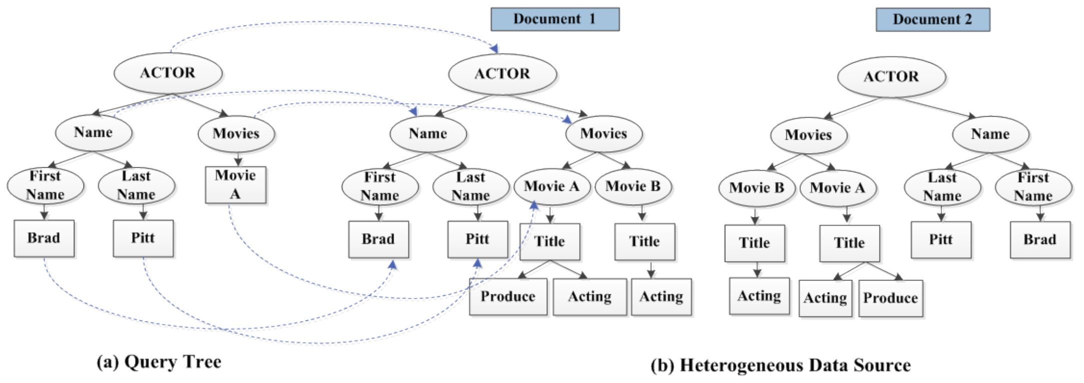


Figure 1.3: Example of a subtree query system using a heterogeneous collection of documents. Here the dotted lines are showing the exact matching between a query tree (a) and the available documents (b)

Consider a heterogeneous collection of documents (Figure 1.3(b)) that are modelled as trees. Quite often, these documents contain the same information with different structures (i.e., different order among sibling nodes). Modelling these documents as unordered trees is more appropriate for similarity computation, since a user will not be aware of document structures. As output, all matching subtrees containing the same information will be retrieved without considering the difference in the sibling node orders. Suppose these trees are organised in a database and a query system is designed to get useful information. The user may have partial knowledge of the data structure and specifies a query that meets his/her information need (Figure 1.3(a)). Due to the enforcement of ordering, only the Document 1 will be returned, despite the fact that Documents 2 also matched the user’s information need (only the sibling nodes are reversed in Document 2). Now, if the sibling order is not used as a grouping criterion in the system, then the query subtree would be treated as unordered and both the documents would be retrieved. The latter result would be more favourable.

The omnipresence of unordered tree data in various applications has sparked interest from the data mining community [23, 32, 34-38]. Discovered information that is manifested in structures with rich semantics can portray many inherent relationships and finding them is significant [27, 35, 39-41]. Learning knowledge from unordered trees includes abstracting useful pattern information, finding similarity information and finding many other relationships embedded within the repository. But extracting knowledge from the data with increased granularity, such as unordered trees, possesses additional processing complexity which cannot be dealt with by simple tree mining techniques. Discovering this knowledge would require developing specialised methods such as similarity measures, frequent pattern mining, and clustering. Compared to the fruitful achievements in ordered tree mining, the field of unordered tree mining yet requires more maturity and in-depth study. Unordered tree mining requires new algorithms to be developed that can deal with the underlying structural flexibility and uncertainty.

Similarity measure methods like tree edit distance, alignment distance, and tree inclusion have been successfully used for comparing various tree data [42-44]. However, these standard edit distance-based methods do not produce desirable results when applied on unordered tree data [42, 45, 46]. Research has shown that when computing the symmetric difference between unordered trees, overstating and double counting problems often arise that result in less accurate measures [23]. Nodes with distinctive parents are counted more than once in various calculations. A variety of methods based on tree edit string operations have been proposed to solve the unordered tree similarity measure problem, but the majority of these methods have provided an intractable solution [42, 45, 47, 48].

Frequent pattern mining is a popular method to discover knowledge from tree-structured data in the form of subtrees. It is a basic step for performing association mining; this is a commonly used data mining technique for finding the association between data entities that can potentially reveal novel and useful relationships. The knowledge driven by frequent pattern mining also has some other important applications such as in database indexing and access method design, classification, clustering, and query system [49-52]. Mining frequent subtrees is non-trivial since it contains hierarchical relationships among the data entities through rich semantics. A subtree mining problem becomes more complex in dealing with unordered and free trees due to the flexibility in structural constraints. For an unordered tree, the subtrees that only differ in permutations of the ordering of siblings are to be considered the same. This causes the “repeated exploration” problem that will result in the generation of a huge number of candidates, where subtrees with similar structure will be included. This eventually creates the “repeated counting” problem [38, 49, 53]. This problem is referred to as an “isomorphism problem” in the literature and the frequency counting step often needs subtree isomorphism checking, which is computationally hard, even known as an NP-complete problem in graph mining algorithms [54]. Exponential candidate generation is another problem wherein a lot of candidate trees, including invalid subtrees, are generated during enumeration [27, 55]. Moreover, it is hard to find a good growth strategy during enumeration. Most of the frequent pattern mining algorithms for the unordered tree type are computationally expensive in terms of both memory usage and run time because of these challenges [11, 12, 49]. Some work has been done to overcome the challenges, but this field still requires improvisation to make the methods efficient and scalable.

The challenges in various tasks of knowledge discovery are in general associated with the structural complexity of tree data [56]. An increasing structural complexity in tree data involves a higher processing cost in various tree manipulation algorithms [11, 42]. Hence, for rooted unordered trees, the knowledge discovery tasks are computationally harder than those of the ordered trees, and also for the free tree, the processing is harder than that of the rooted unordered tree [57-59]. When a tree structure becomes less constrained; it poses additional complexities during processing. If the complex structure of trees can be represented in a way that will ease the processing, the computational intricacy involved in manipulating algorithms can be resolved. Moreover, it is also evident that for developing an algorithm to process any data, one of the essential parts is data representation, and representation has a close relation with the efficiency and scalability of an algorithm [56, 60, 61]. Because of having complex semantics and additional hierarchical information embedded in a tree structure, the efficient encoding of all tree information often requires more memory. Sometimes, the representation does not even reflect the fundamental properties of a tree, especially when the tree is unordered, which may affect the accuracy performance of an algorithm.

For example, the traditional Breadth First Search (BFS) and Depth First Search (DFS) traversal algorithms [62] are largely used for encoding trees during various tree manipulating algorithms. The BFS and DFS algorithms traverse a tree either following the breadth wise or the depth wise direction, where the sibling node are visited from left to right order. Using the traversing order, the tree is encoded. For unordered trees, the order among sibling is not important; therefore similar unordered trees may have different structures varied in sibling orders. The structural dependant traversing strategy of BFS or DFS will provide a different traversing order for each tree, and will result in different encoding for similar unordered trees. This encoding has direct relation with other tree representation methods like canonical form and adjacency matrix, which eventually causes pressing issues such as scalability and accuracy in knowledge discovery methods. Therefore, it is essential to utilise an appropriate data representation scheme for unordered trees.

All of these issues appear more intensely during the processing of free tree since they have a root node as well as no ordered sibling nodes. Mining free tree databases has significant importance in the area of knowledge discovery as modelling trees as free trees offers richer expressivity and a good compromise between graphs and sequences. A graph is a richer representation of tree data, but mining graph data is known as very hard problem in the literature [12]. Sequential mining does not have processing issues but sequences fail to express structural characteristic inherent in the data. Therefore free tree mining often gets priority over graph data mining and sequential mining [54, 63, 64].

This dissertation will explore the mining tasks from a database of labelled unordered trees, with an emphasis on tree similarity measure and frequent subtree mining methods. Firstly, it looks into the scope of using a novel representation for both rooted unordered and free trees that will efficiently capture the embedded relationships and dependencies. It is assumed that this will lead towards achieving less manipulating cost during knowledge processing. The concept of optimisation is utilised to overcome the existing barriers in representation methods. Secondly, it works on the similarity measure method of trees by using the new data representations as well as by utilising the frequent pattern information. Thirdly, it focuses on frequent pattern mining to alleviate the existing research problems by incorporating the new tree representation as well as by improvising the candidate enumeration and frequency counting steps. Besides using new representation, an optimised enumeration approach has been explored for generating candidate trees that will only generate valid subtrees without hampering the completeness property (i.e., will not miss any candidate patterns). All the proposed works and findings are evaluated with state-of-the-art methods using multiple datasets with diverse characteristics.

1.3 RESEARCH OBJECTIVES AND GOALS

The objective of this research is to provide efficient and scalable methods for discovering knowledge from databases of labelled unordered trees. To achieve this objective, the research emphasises tree representation, as it is usually a mandatory step in tree manipulation methods. For knowledge discovery, the research focuses on two important tree mining problems, tree matching and frequent subtree mining.

This research is guided by the following goals to achieve the above mentioned objectives:

– Utilising an optimisation technique for representing unordered tree data in a structure independent manner since it represents more complex and less constrained structures. Based on this representation, the canonical form and matrix form representations can be developed that will allow more appropriate encoding and efficient manipulation of rooted unordered trees and free trees.

– Proposing a tree matching algorithm that will provide tractable solution to the similarity measure problem of unordered trees. This algorithm should avoid complex mapping between unordered tree pairs by using an appropriate data model. This similarity measure should ensure fast computation without compromising accuracy.

– Developing fast and effective frequent subtree (e.g., induced and embedded) mining algorithms by using the introduced canonical form that will ensure efficient indexing of rooted unordered and free trees during frequency counting and candidate generation steps. In order to make the frequent subtree mining algorithms computationally efficient an optimal and non-redundant candidate enumeration technique needs to be developed. Also, the frequency counting step needs to be explored to boost its performance.

1.4 RESEARCH QUESTIONS

This research is structured to answer the following research questions:

1. How can the labelled unordered tree be represented in a more appropriate and efficient manner?
    a. Can the existing traversal techniques (Breadth-first Search and Depth-first Search) encode the unordered trees without breaching their structural flexibility?
    b. Can an optimisation technique be utilised for representing unordered trees?
    c. Can an unordered tree be represented through the traditional adjacency matrix?
    d. Which canonisation will ensure unique identity of both unordered and free trees regardless of the structural flexibility?
2. What is the appropriate method for addressing the tree matching problem from a database of labelled unordered trees?
    a. Is a better accuracy and scalability possible with the proposed method in comparison to the tree edit distance-based methods?
    b. Can representation play a role in reducing the computation complexity of the unordered tree matching algorithm?
    e. Can the knowledge of frequent subtrees be helpful in finding similarity between trees pairs?
3. How can the frequent subtree mining algorithms be designed for mining frequent rooted unordered and free subtrees through ensuring less run time and memory usage?
    a. Can a canonical form provide unique identity of unordered trees in the presence of isomorphism and automorphism?

b. How can the candidate generation be restricted without hampering the completeness property?

c. How can an enumeration approach be optimised by generating only the valid candidate trees?

d. What is a more suitable approach for executing the frequency counting?

1.5 RESEARCH CONTRIBUTIONS

This thesis has developed the following contributions in the field of knowledge discovery from the databases of labelled unordered trees:

– A novel tree traversal algorithm, named as **B**alanced **O**ptimal **S**earch (BOS), is proposed by reducing the tree traversal problem to the SALB (Simple Assembly Line Balancing) problem, a well-known optimisation problem in Operations Research (OR) paradigm [65]. An optimisation model is formulated for solving the traversing problem, which consists of feasibility constraints and an objective function for minimising the computation time of traversal. BOS traversal gives an optimal traversing sequence for a rooted unordered tree without relying on a fixed left-to-right order among sibling nodes, unlike existing traversal algorithms [62]. In order to enhance the effectiveness of frequent subtree mining algorithms, new canonical forms called **B**alanced **O**ptimal **C**anonical **F**orms (BOCF) are proposed based on BOS traversal for effectively representing rooted unordered trees and free trees.

– A new data structure-based tree matching algorithm for unordered trees is introduced. The traditional adjacency matrix representation of trees uses a BFS or DFS traversal driven encoding in its construction. BFS and DFS traverse a tree following breadth- and depth- wise movements respectively. Their encodings preserve the structural flexibilities such as sibling order variations. Even if the unordered trees are similar they have different encodings because of the different sibling orders. This leads to having different matrix representations for similar unordered trees. Instead of a structure dependent traversal strategy, the BOS traversal is used to provide optimal encoding of trees that are independent to the structural variations. By using this encoding and additional tree structural information, an approximate numerical matrix representation called **A**ugmented **A**djacency **M**atrix (AAM) is presented, which ensures similar matrix representations for similar trees. Finally, the vector cosine similarity metric is modified to make it compatible with matrix computation for calculating the similarity between tree pairs. The similarity information is further used in clustering to show an application of this method. Necessary empirical analysis has been conducted to establish the findings.

- Another new data structure-based tree matching algorithm is proposed, which is utilising not only the tree information but also the database specific knowledge for measuring similarities between unordered trees. By applying the frequent pattern mining algorithm, the common structures present in a database can be discovered, which often aids in understanding a database, especially a new one [24, 49]. Using additional information, in the form of frequent structural dependencies like parent-child, for representing a tree, will emphasise the characteristics of the database during finding similarities between its trees. In this work, a novel **E**xtended **A**ugmented **A**djacency **M**atrix (EAAM) representation is introduced, that consists of the frequent subtree information of a particular database along with other important information of an individual tree. The EAAM representation also uses BOS encoding to ensure unique identity of a rooted unordered tree. The unordered trees represented in EAAM are compared to calculate the similarity between a tree pair, and used as a clustering input to group the trees of a database. This work is empirically evaluated against relevant benchmarking works.
- An efficient **B**alanced **O**ptimal **S**earch **T**ree min**ER** (BOSTER) algorithm is developed to mine frequent induced unordered subtrees from a database of labelled rooted unordered trees. BOCF is used to generate candidate subtrees using a tree structure guided scheme based- enumeration approach. Representing the rooted unordered trees has been always an issue due to the flexible order among sibling nodes which causes the isomorphism problem. It is important to represent trees uniquely during candidate generation to ensure accurate frequency counting through correct indexing. BOCF handles the isomorphism and automorphism

problems efficiently. Exponential candidate generation is another pressing problem in frequent unordered tree mining that BOSTER mitigates using the tree structure guided scheme-based enumeration by generating the valid candidate subtrees only. A catching technique is used to boost-up the frequency counting step. BOSTER is evaluated and compared against relevant benchmark algorithms.

- Another important frequent mining algorithm, **B**alanced optimal search **E**mbedded **S**ub**T**ree miner (BEST) that finds the set of frequent embedded unordered subtrees from a database of labelled rooted unordered trees is proposed. Mining embedded subtrees can be seen as a generalisation task of mining induced subtrees that mines interesting relational information inherent within deeply embedded data objects in the tree database. It is a more difficult problem than induced subtree mining as it requires examining several levels within a tree to identify an embedded subtree. Both the extension and join operations are defined using a level constraint to enumerate only the valid candidate subtrees. BEST is compared with several benchmarks using both real and synthetic datasets.
- The problem of mining frequent free subtrees in a database of labelled free trees is considered and a fast algorithm called FreeS (Frequent **Free** **S**ubtree) is proposed. Free trees can be considered as a good compromise between graph and sequence data, and as a stepping stone towards solving the graph mining problem [66]. The BOCF canonical form of free trees requires an additional step for normalising the root node. Using this BOCF of free trees, a tree structure guided scheme based enumeration approach is introduced that avoids generating false positive in the candidate generation step, one of the key issues in frequent pattern mining. A lemma is proved that satisfies the conditions to grow the enumeration tree using extension and join operations using the proposed canonical form of free trees. FreeS is compared with several benchmarks using both real and synthetic datasets.

1.6 ACCOUNT OF RESEARCH PUBLICATIONS

This is a thesis by publication, and the body of the thesis is comprised of peer-reviewed journal and conference papers. Each paper listed in Table 1.1 is either published, accepted, or submitted for review. In this table, the thesis chapter number is also mentioned where the full paper has appeared.

Publications	**Details of papers included in thesis**
– Paper- 1 (*TRA*) {Citations: 3} {Chapter 4}	– *Chowdhury, Israt J.* & Nayak, Richi (2013) A novel method for finding similarities between unordered trees using matrix data model. Lecture Notes in Computer Science: WISE 2013, 8180, pp. 421-430
– Paper- 2 {Chapter 4}	– *Chowdhury, Israt J.* & Nayak, Richi, "Measuring Similarity between Unordered Trees with the Balanced-Optimal-Search Traversal Algorithm", *Knowledge and Information Systems* (Under Review)
– Paper- 3 (*TRB*) {Chapter 4}	– *Chowdhury, Israt J.* & Nayak, Richi (2014) Identifying product families *using* data mining techniques in manufacturing paradigm. In Nayak, Richi, Li, Xue, Liu, Lin, Ong, Kok-Leong, Zhao, Yanchang, & Kennedy, Paul (Eds.) Australasian Data Mining Conference (AusDM), Australia
– Paper- 4 (*TRA*) {Citations: 2 Chapter 5}	– *Chowdhury, Israt J.* & Nayak, Richi (2014) BOSTER: an efficient *algorithm* for mining frequent unordered induced subtrees. Lecture Notes in Computer Science: WISE 2014, 8786, pp. 146-155
– Paper- 5 (*TRB*) {Citations: 1 Chapter 5}	– *Chowdhury, Israt Jahan* & Nayak, Richi (2014) BEST: an efficient *algorithm* for mining frequent unordered embedded subtrees. Lecture Notes in Computer Science: PRICAI 2014, 8862, pp. 459-471
– Paper- 6 (*TRA*) {Chapter 5}	– *Chowdhury, Israt J.* & Nayak, Richi (2015) FreeS: Fast Algorithm to *Discover* Frequent Free Subtrees Using a Novel Canonical Form. Lecture Notes in Computer Science: WISE 2015, 9418, pp. 123–137

Table 1.1: List of peer reviewed papers forming chapter in this thesis

STATEMENT OF THE CONTRIBUTION

All of the published papers that are included in this thesis are co-authored by the PhD candidate, Israt Jahan Chowdhury and the candidate’s principle supervisor, Associate Professor Richi Nayak. Apart from that no one else has contributed in these published papers. Both authors have agreed to use these publications as a part of this thesis.

1.7 HIGH LEVEL OVERVIEW

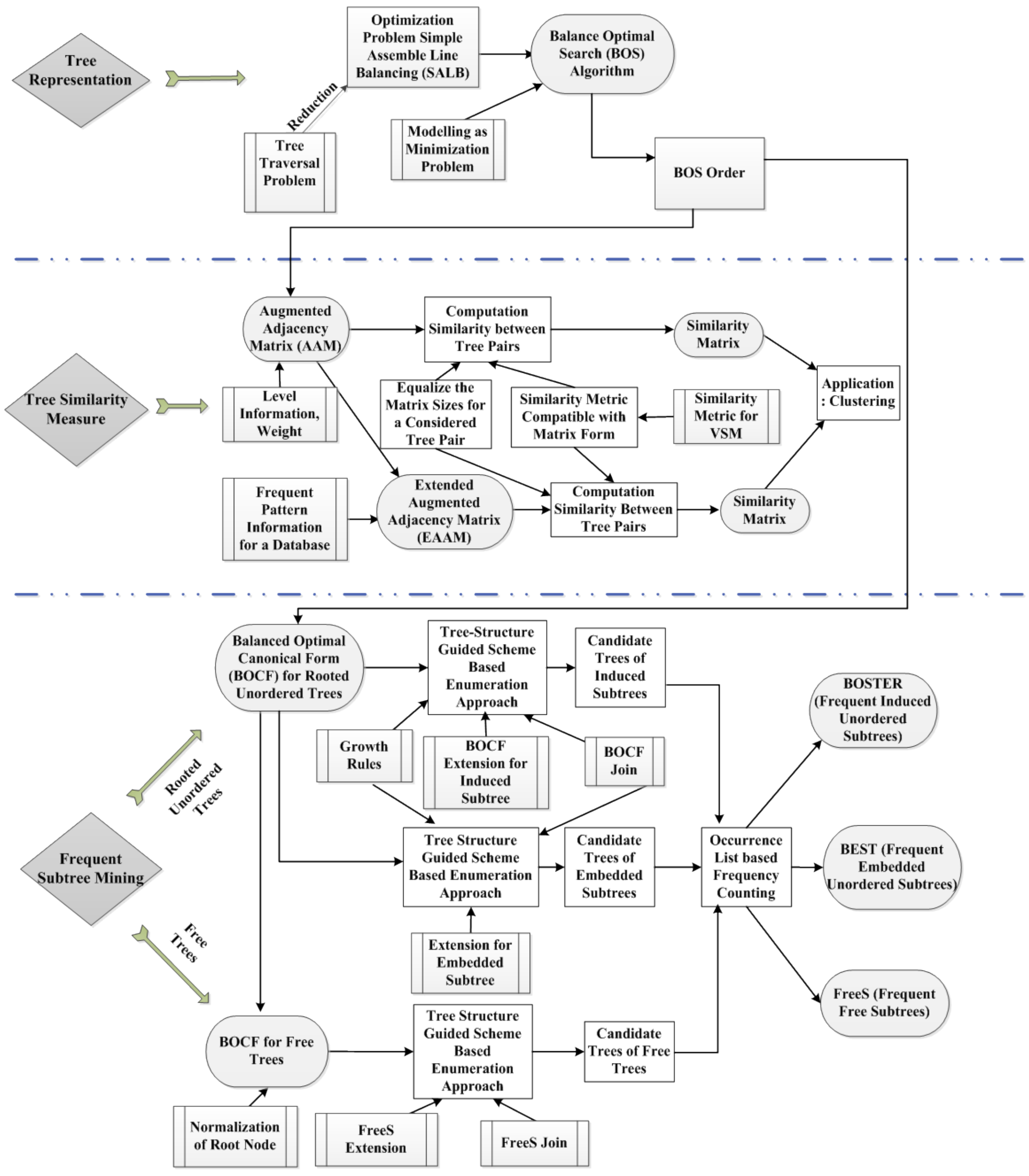
The aim of this section is to present a high level overview of all the contributions made in the thesis and show how they fit together. A novel tree traversing scheme based on optimisation and the novel tree representations using this scheme are developed to facilitate the effective knowledge discovery from labelled unordered tree databases. A tree matching algorithm is developed based on the new matrix form. Novel frequent pattern mining algorithms are developed using the proposed representation and a new enumeration approach with specific growth rules. The matrix representation is further extended utilising the results of frequent pattern mining algorithms to incorporate more domain specific insights into similarity calculation. The similarity measure results are evaluated through a clustering algorithm. A map of the contributions of this thesis is presented in Figure 1.4. Each arrow indicates that the following contribution is built upon the results in the previous contribution.

The backbone of this thesis is the novel Balanced Optimal Search (BOS) traversal algorithm, which is proposed by reducing the tree traversal problem to the Simple Assembly Line Balancing (SALB) problem – an optimisation problem from an Operations Research (OR) paradigm. The BOS traversal derives an optimal encoding of an unordered tree that ensures a total unique order for all available similar unordered trees in a database. A novel tree representation named as Balance Optimal Canonical Form (BOCF) is defined using BOS traversal, which can represent a rooted unordered tree uniquely. The BOCF is extended to define canonical form for representing free trees.

The optimal BOS encoding is further used to overcome the limitations of traditional adjacency matrix representations of unordered trees. This optimal order gives birth to a novel adjacency matrix representation called Augmented Adjacency Matrix (AAM), which allows capturing more tree information in matrix form by including adjacency information, level information and weight along with BOS encoding. The AAM facilitates comparing tree pairs with using the cosine similarity metric adopted for matrix. The proposed tree similarity measure method is found efficient and scalable in comparison to the traditional tree similarity measure methods based on empirical analysis. The tree edit distance problem is commonly used for finding similarity between unordered trees is known to be computationally hard (no known tractable solution without restricting tree parameters), whereas the proposed AAM based similarity measure method offers a radical reduction in the computational complexity without an accuracy compromise. The result of this method is further used as input to a clustering algorithm — an important application of this contribution.

Another matrix representation of unordered tree, Extended Augmented Adjacency Matrix (EAAM) is defined by incorporating the knowledge of frequent subtrees of a database in the basic AAM construction. This provides a domain specific insight of tree data as the frequent pattern mining allows initial analysis of an unexplored database. The EAAM is used for calculating similarity between tree pairs and used as input to a clustering algorithm. All of these results are evaluated and compared with relevant benchmark methods.

The BOCF representation of a rooted unordered tree is used to propose two scalable frequent pattern mining algorithms, BOSTER and BEST for unordered trees that can mine frequent induced and embedded subtrees respectively. BOCF resolves the isomorphism and automorphism problems quite naturally. Hence, the processing time is reduced by skipping an additional isomorphism checking test unlike the state-of-the art methods. Moreover, a tree structure guided scheme- based enumeration is used that alleviates generation of the false positive candidates and boosts the frequency counting step. The enumeration process consists of two operations, BOCF extension and BOCF join, that are defined according to the proposed canonical form and enumeration approach. Growth rules are specified in these algorithms for restricting the number of potential nodes for having an extension in enumeration process. The algorithm for mining induced subtrees can be considered as a generalised algorithm for mining embedded subtrees. An additional level constraint is introduced while mining embedded subtrees to make the corresponding algorithm scalable. Both the algorithms are evaluated using the relevant and state-of-the-art algorithms from the literature. Empirical analysis shows the superior performance of the proposed algorithms over the benchmarking algorithms. These algorithms are found computationally efficient in term of both memory and runtime in comparison to the state-of-the-art algorithm.



Tree Representation
Optimization Problem Simple Assemble Line Balancing (SALB)
Balance Optimal Search (BOS) Algorithm
Reduction
Tree Traversal Problem
Modelling as Minimization Problem
BOS Order
Augmented Adjacency Matrix (AAM)
Computation Similarity between Tree Pairs
Similarity Matrix
Application : Clustering
Tree Similarity Measure
Level Information, Weight
Equalize the Matrix Sizes for a Considered Tree Pair
Similarity Metric Compatible with Matrix Form
Similarity Metric for VSM
Frequent Pattern Information for a Database
Extended Augmented Adjacency Matrix (EAAM)
Computation Similarity Between Tree Pairs
Similarity Matrix
Balanced Optimal Canonical Form (BOCF) for Rooted Unordered Trees
Tree-Structure Guided Scheme Based Enumeration Approach
Candidate Trees of Induced Subtrees
BOSTER (Frequent Induced Unordered Subtrees)
Rooted Unordered Trees
Growth Rules
BOCF Extension for Induced Subtree
BOCF Join
Frequent Subtree Mining
Tree Structure Guided Scheme Based Enumeration Approach
Candidate Trees of Embedded Subtrees
Occurrence List based Frequency Counting
BEST (Frequent Embedded Unordered Subtrees)
Free Trees
Extension for Embedded Subtree
FreeS (Frequent Free Subtrees)
BOCF for Free Trees
Tree Structure Guided Scheme Based Enumeration Approach
Candidate Trees of Free Trees
Normalization of Root Node
FreeS Extension
FreeS Join

Figure 1.4: A research map to provide the high level overview of the thesis

Research Phase	Phase 1			Phase 2		Phase 3		
Research Activity	Tree Representation and Data Structure			Tree Matching		Frequent Subtree Mining		
Research Question	Q 1			Q 2		Q 3		
Corresponding Chapters	Chapter 3			Chapter 4		Chapter 5		
Contributions	BOS	BOCF	Adjacency Matrices	AAM-based Method	EAAM - based Method	BOSTER	BEST	FreeS
Corresponding Papers	Paper 1, 2	Paper 4, 5, 6	Paper 1, 2, 3	Paper 1, 2	Paper 3	Paper 4	Paper 5	Paper 6
BOS Traversal								
Reduction from SALB to Tree Traversal	The Optimisation Model Formulation		Pseudocode of the Algorithm		Properties of BOS along with complexity analysis			
Paper 2 (Sub-section 3.1 & 3.2)	Paper 2 (Sub-section 3.3)		Paper 2 (Sub-section 3.3)		Paper 2 (Sub-section 3.3)			
Balance Optimal Search Canonical Forms (BOCFs)								
BOCF for Rooted Unordered Tree (definition and properties)				BOCF for Free Tree (definition and properties)				
Paper 4 (Sub-section 3.1); Paper 5 (Sub-section 3.2)				Paper 6 (Sub-section 3.1)				
Adjacency Matrices								
AAM definition	AAM Properties		EAAM definition					
Paper 1 (Sub-section 2.2); Paper 2 (Sub-section 4.2)	Paper 2 (Sub-section 4.2)		Paper 3 (Sub-section 4.3)					

Table 1.2: A detailed sketch of the major contributions made in the thesis

The BOCF representation of rooted unordered trees is used to define canonical form of free trees by using tree normalisation. A scalable algorithm, FreeS for mining frequent free trees is then proposed. This algorithm uses additional conditions for enumerating free candidate trees with the support of tree structure scheme; accordingly the FreeS-extension and FreeS-join operations for growing the enumeration tree are defined. Evaluation is done by comparing it with relevant state-of-the-art methods. It provides an improved result by a few orders of magnitude for the computational complexity. Currently, this algorithm works on a database of labelled free trees and can be considered as a first step towards mining frequent subgraphs in the future.

From the above discussion it is ascertained that the research carried out in this thesis has three major phases. In the first phase, the BOS traversal is proposed. Based on BOS traversal, some novel tree representation methods – BOCF canonical forms and adjacency matrices – are introduced. In the second phase, tree matching algorithms are developed and in the third phase, frequent subtree mining algorithms are proposed. Table 1.2 shows the corresponding references in this thesis where the necessary descriptions of the proposed methods under each research phase has been made.

1.8 RESEARCH SIGNIFICANCE

This thesis advances the field of knowledge discovery from tree databases with a focus on alleviating the hurdles of existing tree representation methods. The BOS based representation plays an important role in significantly improving the scalability performance of frequent subtree mining and tree matching algorithms. The area of unordered tree mining is under researched; this makes the significance of this research unquestionable.

The research carried out in this thesis has practical significance, since all of these contributions have a relationship with many real life applications.

– The research focus is on mining unordered and free trees, which are often used in modelling various common and popular domain data such as Weblog, XML, BOM, Glycan and many more. This is an era of “big data”, where data are coming from many sources and are stored in a common platform for future manipulation for knowledge discovery. Data coming from various sources are likely to have inconsistency, where unordered tree modelling is more suitable to support the overall knowledge discovery process. In general, the findings in this area of research are going to benefit various domains, which are currently lacking in the process of discovering knowledge from such less constrained and complex data models.

- The balance optimal traversal search based representations (i.e., BOS encoding and BOCFs) can ensure one-to-one mapping of unordered and free trees regardless of the presence of isomorphic trees in a database. This will greatly benefit the indexing of a database of trees. Finding frequently occurring subtrees can also help the database indexing system. Moreover, knowledge in the form of frequent subtrees improves a user's understanding about a data source. The frequent subtree mining algorithm also serves as the first step in classifying and clustering tree-structured data.
- The frequent free tree mining algorithm can be helpful in solving some graph and network data problems, which are a very common data format in social network and business intelligence systems.
- The similarity information of a tree database may facilitate building a query system. This similarity information can feed to a clustering algorithm for grouping trees without any class information. Based on similarity measures, a nearest-neighbour classification, data integration and data cleaning methods can be built upon.
- The proposed methods of knowledge discovery focus on scalability and less complex processing which will be beneficial for processing big data.

1.9 THESIS OUTLINE

A detailed introduction of the thesis topic is provided in Chapter 1 with specific research questions and objectives. A brief relational map of contributions is added to provide a clear idea of research tasks. Since the thesis is presented as a *thesis by publication*, the reader may notice some repetition of materials between published articles. Each article should be self-contained and therefore, has been published with relevant material for completeness. In this way, a reader does not have to refer to several different references to get the whole picture for the results presented. Therefore, the author's suggestion to the reader is to skip the repeated parts unless otherwise you have not read them already. The outline of other chapters is given below:

Outline of Chapter 2

A concise review of existing peer-reviewed literature and the necessary prerequisites to understand the thesis contents are presented in Chapter 2. Critical studies on existing literature are performed based on the research questions and objectives. These studies mainly present a review of the literature on tree representation, tree similarity measure for unordered tree pairs, and frequent subtree mining, especially the mining of frequent rooted unordered and free subtrees.

Outline of Chapter 3

After the initial literature review, the research questions and objectives are addressed gradually. This chapter mainly focuses on tree representation, which is a primary contribution in this thesis. The novel tree traversal approach, canonical forms and matrix representations are introduced briefly as the attached subsequent publications include the details of these concepts. Although the technical detail of each of the representations is discussed under the published articles with the corresponding algorithms, Chapter 3 is presented as a hub for other contributing chapters to increase the thesis readability and to avoid abrupt discussion.

Outline of Chapter 4

Chapter 4 presents the detailed contribution on tree matching algorithms based on Paper 1, Paper 2 and Paper 3. Paper 1 is published in a Tier A conference, which includes preliminary information on the BOS traversal approach and AAM representation. Paper 2 details the overall BOS traversal algorithm, including mathematical modelling and heuristics. It includes the detailed empirical analysis of the tree matching algorithm. This is currently under review in a high impact factor journal. Paper 3 has been published in a popular Tier B conference, and utilises the tree matching algorithm with the EAAM matrix representation. Before presenting the paper's contents, a preamble is added to explain its contents.

Outline of Chapter 5

Chapter 5 is formed from Paper 4, Paper 5 and Paper 6 and describes the contributions to frequent subtree mining from the databases of trees and free trees. Paper 4 was published in a Tier A conference, and explains the algorithm of mining frequent rooted unordered induced subtrees. Paper 5 is published in a well-known

Tier B conference that describes the algorithm of mining frequent rooted unordered embedded subtrees. Paper 6 is accepted in a Tier A conference that is about the algorithm of mining frequent free subtrees. Before presenting the paper contents, a preamble is included to explain the context of these papers in the thesis.

Outline of Chapter 6

Chapter 6 summarises the outcomes obtained from the research work in Chapters 3, 4 and 5. The significant research findings are specified, and also mentioned are how these findings have answered the considered research questions. Finally, recommendations for future research directions are suggested.

Chapter 2: Literature Review and Background

This section will give a review of the tree structured data and various tree mining techniques. The background of labelled unordered trees with basic tree concepts and tree mining terminologies is detailed first. The methods of tree representation are discussed next, guiding the discussion on two major data mining tasks tree matching and frequent subtree mining. This research is focused on unordered and free trees; therefore the state-of-the-art research of these types of trees will mostly be discussed here. Moreover, the limitations of various tree mining methods will be highlighted to support the research hypothesis of this thesis. Figure 2.1 outlines the main areas to which this thesis is related. It provides the relationships between various fields of tree mining research, as viewed in this thesis. The middle area, where clustering is shown as a common part, is a real life application, which can be fitted to both of tree matching and frequent subtree mining algorithms. Clustering is briefly discussed under the preamble of Chapter 4.

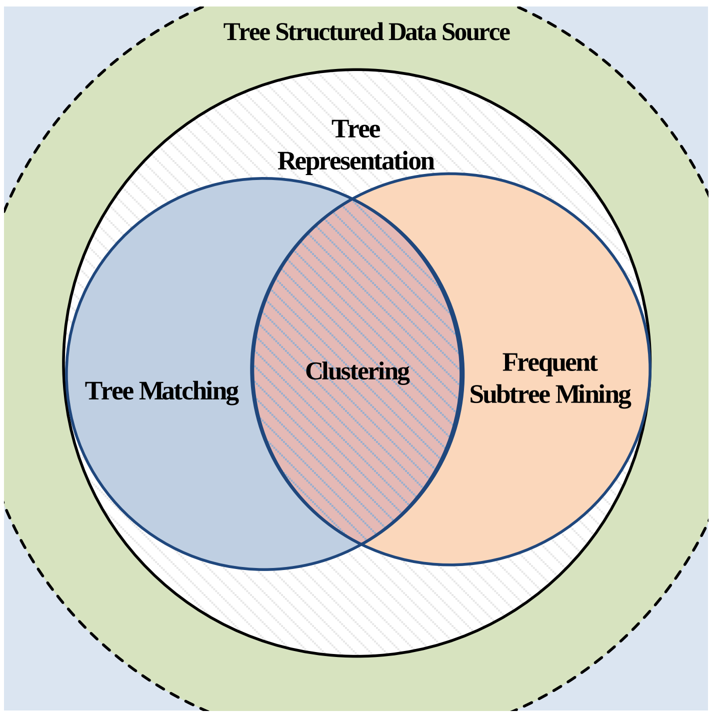


Figure 2.1: Related research areas and coverage of literature review

2.1 TREE STRUCTURED DATA SOURCE

Semi-structured data can portray the two-dimensional relationships among data entities that are manifested through structural relationships among entities. Hence, the analysis of semi-structured data objects can often reveal valuable information [11]. Trees are the most common data format used to represent semi-structured data [12, 19, 67]. Due to the usefulness of semi-structured data, the research field of tree mining has gained a considerable amount of interest in applications such as XML document management, Web intelligence, Bioinformatics, Manufacturing and Product Design [38, 49, 68]. This section presents some of the significant domains that use tree data to express their domain information. Data originated from some of these domains have been used in this thesis for evaluating the designed methods. Table 2.1 provides a summary of these domains.

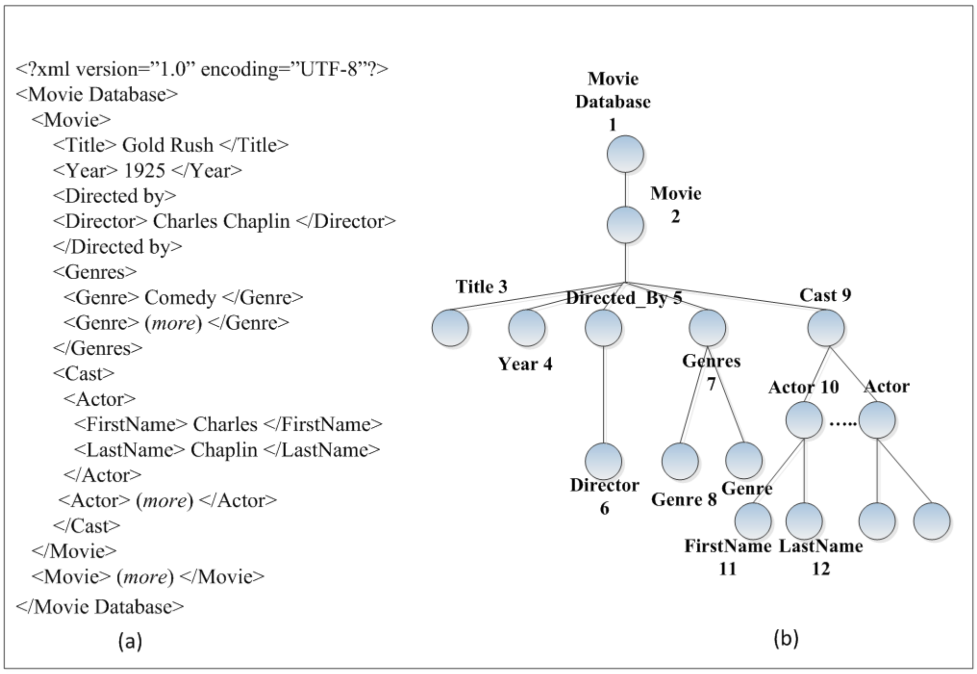
DOMAINS	**EXAMPLE OF DATA**	**BRIEF DESCRIPTION**
Internet/ Intranet	XML or HTML	Quite often the online information is stored and exchanged in HTML or XML format. These data on Internet / Intranet can be represented as trees [67].
Web Intelligence	Web log	The Web log data represented with tree format, can provide useful insight on user behaviour [20, 38].
Production or Manufacturing Industry	Bill of Material (BOM) [69]	Similarity information among Bill of Materials (BOMs) of various products can help in accelerating the design phase of a new product. Based on the similarity, often BOM of an existing product is reused and modified to design a new product. A BOM document can naturally be depicted as rooted unordered tree [23, 24].
Bioinformatics	RNA secondary structures, Phylogenetic trees, Glycan, etc.	RNA structures represented as trees can be compared for finding important information of a newly sequenced RNA based on the common topological patterns of a known RNA [25, 70]. This is useful for obtaining some important clues about the function of the RNA.

Table 2.1: The example of various tree data domains

2.1.1 XML (eXtensible Markup Language)

XML (eXtensible Markup Language) is a markup language defined by the World Wide Web Consortium (W3C) that consists of a set of rules for encoding documents [71]. XML is advantageous in comparison to other markup languages like Hypertext Markup Language (HTML), because it describes the content in a way that is readable to both human and machine. Moreover, XML allows for user-defined tags that makes it more flexible than HTML. XML data is application and platform independent. Figure 2.2(a) shows a simple example of an XML document.

An XML document can be naturally represented as a tree [67]. For deriving a tree structure from an XML document various XML parsers (e.g., Document Object Model (DOM) and, the Simple API for XML (SAX)) are used which treat the element in an XML document as a node in a tree representation. To show the hierarchical relationships between elements, tree branches are used. For instance a tree-based model for the XML document in Figure 2.2(a) can be derived as the one shown in Figure 2.2(b). These trees are often modelled as unordered [29, 41] as there is no order in appearances of multiple instances. The unordered representation also assists in dealing with the irregularities and inconsistency that may present in an ill-formed XML document due to it originating from heterogeneous sources.



```
<?xml version="1.0" encoding="UTF-8"?>
<Movie Database>
  <Movie>
     <Title> Gold Rush </Title>
     <Year> 1925 </Year>
     <Directed by>
     <Director> Charles Chaplin </Director>
     </Directed by>
     <Genres>
       <Genre> Comedy </Genre>
       <Genre> (more) </Genre>
     </Genres>
     <Cast>
       <Actor>
         <FirstName> Charles </FirstName>
         <LastName> Chaplin </LastName>
       </Actor>
      <Actor> (more) </Actor>
     </Cast>
  </Movie>
  <Movie> (more) </Movie>
</Movie Database>
```

Figure 2.2: Tree modelling from XML data [72]

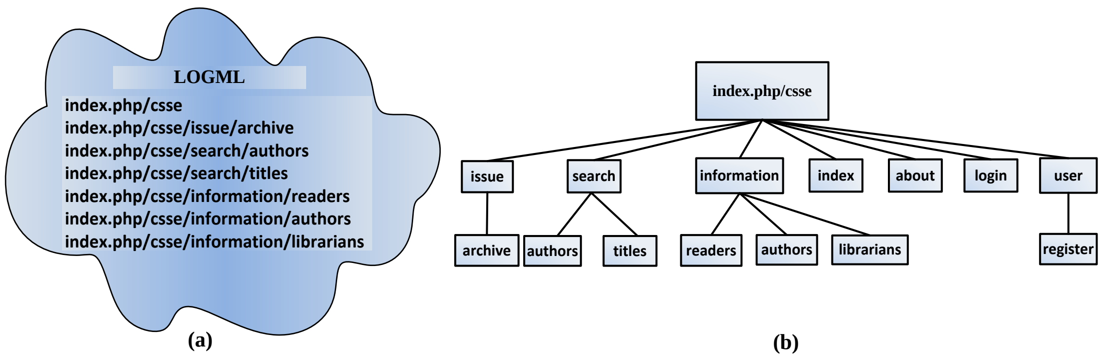

LOGML
index.php/csse
index.php/csse/issue/archive
index.php/csse/search/authors
index.php/csse/search/titles
index.php/csse/information/readers
index.php/csse/information/authors
index.php/csse/information/librarians
(a)
index.php/csse
issue
search
information
index
about
login
user
archive
authors
titles
readers
authors
librarians
register
(b)

Figure 2.3: Example of Web log data in tree format [11]

2.1.2 Web log data

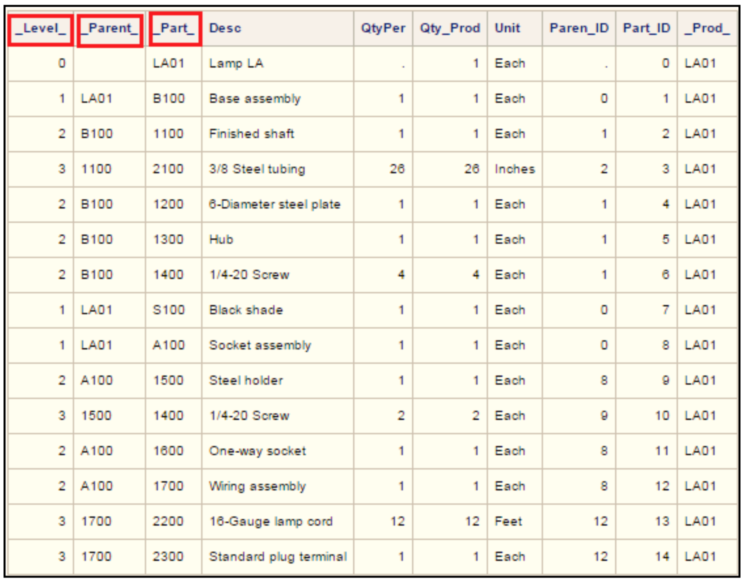
Web log data contains information on Web users' browsing behaviour during a visit to a Web site. Analysis of user browsing behaviour can result in obtaining user browsing patterns and frequent usage paths [73, 74]. These useful insights inform site managers for improvement of the site as well as creation of business opportunities.

Recent research advancement in Web mining encourages a more complex structural representation of Web log data, which will allow the capturing of deep information on structure of the site and navigational patterns. A popular representation language of Web log data is LOGML [33, 75], which uses XML templates to detail the user activities. LOGML data can easily be represented as trees where the set of requested Web pages refer to the tree nodes and the traversed hyperlinks in a Web log file refer to the edges or links between tree nodes.

An example of tree representation of Web log data in LOGML format is shown in Figure 2.3. From the sequence of logs in Figure 2.3(a), the 'index.php/csse' is considered as the home page which leads to the tree representation as shown in Figure 2.3(b). The unordered tree representation of Web log data allows finding more detailed insights of a Web domain [38], as discussed in Chapter 1.

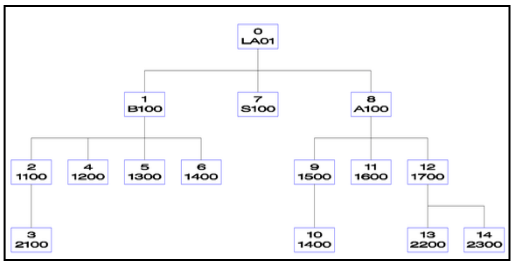
2.1.3 Bill of Material (BOM)

Bill of Material (BOM) is a common data type used in various engineering domains such as mechanical, civil or infrastructure, electrical and electronic. It is a structured or hierarchical portrayal of an end product comprising information about part or components, raw materials, quantity and manufacturing instructions [76]. BOM data is usually produced in tabular form that represents the overall description of particular product manufacturing. By considering the parent and part name, the BOM data can be easily represented as a tree, whereby the underlying product will be the root node and the tree model will maintains the parent-child relationship by using the level information, parent and part name [23, 77]. For BOM data only the ancestral or parent-child relationship is significant; the order among the parts under the same parent is unimportant. That is why the unordered tree modelling of BOM data will result in meaningful analysis.



Level	_Parent_	_Part_	Desc	QtyPer	Qty_Prod	Unit	Paren_ID	Part_ID	_Prod_
0		LA01	Lamp LA	.	1	Each	.	0	LA01
1	LA01	B100	Base assembly	1	1	Each	0	1	LA01
2	B100	1100	Finished shaft	1	1	Each	1	2	LA01
3	1100	2100	3/8 Steel tubing	26	26	Inches	2	3	LA01
2	B100	1200	6-Diameter steel plate	1	1	Each	1	4	LA01
2	B100	1300	Hub	1	1	Each	1	5	LA01
2	B100	1400	1/4-20 Screw	4	4	Each	1	6	LA01
1	LA01	S100	Black shade	1	1	Each	0	7	LA01
1	LA01	A100	Socket assembly	1	1	Each	0	8	LA01
2	A100	1500	Steel holder	1	1	Each	8	9	LA01
3	1500	1400	1/4-20 Screw	2	2	Each	9	10	LA01
2	A100	1600	One-way socket	1	1	Each	8	11	LA01
2	A100	1700	Wiring assembly	1	1	Each	8	12	LA01
3	1700	2200	16-Gauge lamp cord	12	12	Feet	12	13	LA01
3	1700	2300	Standard plug terminal	1	1	Each	12	14	LA01

(a)



(b)

Figure 2.4: Tree modelling from BOM data (Collected from SAS Bill of Material Processing[1])

1 http://support.sas.com/documentation/cdl/en/orbomug/63972/HTML/default/viewer.htm#orbomug_bom_sect002.htm#orbomug.bom.gs3_

Figure 2.4 gives an example of a raw BOM data and its corresponding tree modelling. This is the Bill of Material of a product named table lamp, ‘LA01. The table in Figure 2.4(a) contains the information on various parts of the lamp, such as part level positions, parent name, part name, quantity per parent, quantity per product, etc. For building a tree only the red marked information is used, which includes the level information, the parent item and the part number of the component under each parent item. If a component is used in more than one parent item, it appears in multiple records. For example, the part number '1400' is used in both ‘B100’ and ‘1500’; this item occurs in records identified by the values 6 and 10 in Figure 2.4(b).

2.1.4 Glycan

In bioinformatics, after DNA and proteins, the third major class of biomolecules is carbohydrate sugar chains knows as glycans [78]. Glycan carries important genomic information, and is extremely vital in functioning multicellular organisms. Gaining insight from this data structure has practical significance. The general structure of glycan is very complex and contains many branching monosaccharides, starting from a single monosaccharide, which allows it be represented through a rooted tree structure. Since siblings do not have order precedence, Glycan is a good example of real-life rooted unordered tree data. Researchers have treated glycans as rooted unordered trees and have applied tree mining techniques for discovering useful knowledge from them [21, 40, 79]. In Figure 2.5, a sample glycan structure is shown; similar examples can be found in the KEGG database [80], one of the famous repositories for glycan data.

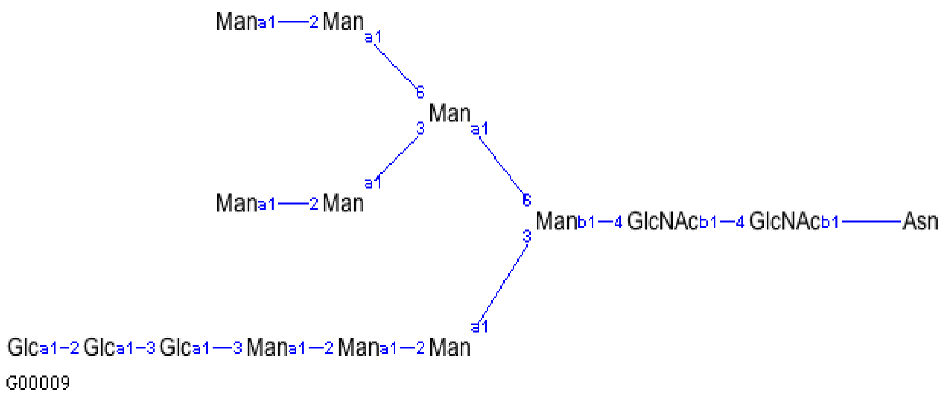

Mana1—2 Man
Man
Mana1—2 Man
Manb1—4 GlcNAcb1—4 GlcNAcb1—Asn
Glca1−2 Glca1−3 Glca1—3 Mana1−2 Mana1−2 Man
G00009

Figure 2.5: A sample glycan structure

Discussion: The omnipresence of trees is noticeable in various domains such as web intelligence, bioinformatics, production process and many others. Quite often, treating these data as rooted unordered trees allows the discovery of more useful knowledge and insights. Some of these data are naturally structured as rooted unordered trees (i.e. Glycan, BOM) and treating these domain data as rooted ordered trees will violate the fundamental properties embedded in their structure. On the other hand, some of the domain data are preferred to be modelled as rooted unordered trees (i.e. Web log data, XML) for supporting some specific applications, such as in some applications; it is preferable to regard input trees as unordered trees to allow more flexible matching. Hence, modelling these data as rooted ordered trees may cause the loss of some interesting patterns and information because of enforcing the grouping constraint. In fact, any data that exhibits a hierarchical relationship can be represented as trees, and can further be analysed through various tree mining techniques for insight in the domain. Moreover, patterns in the forms of sub-trees are found to be more descriptive and informative than itemsets or sequence patterns [11]. So, developing methods for mining tree data has great value and conducting research in the area of rooted unordered trees is essential, as this field is still in need of developing some efficient methods.

2.2 BASIC TREE CONCEPTS

Tree data is an interesting compromise between the structural representation such as graphs, and the linear representation such as vectors and matrixes. This can be considered as a natural representation of rules and hypotheses which expresses hierarchical dependencies with implicitly defined semantics [81].

The following definitions are adopted from [82-84], which are the necessary basics of a tree structure data and its various formalisations.

A labelled tree can be formally denoted as $T = (V, E, L, Ø)$, where (1) the set of nodes is $V(T) = \{v_0, v_1, v_2, …, v_n\}$, v_0 = root, (2) the set of edges is E, defined as $E = \{(v_i, v_j) \mid v_i, v_j \in V\} = \{e_1, e_2, …, e_n\}$; (3) L is the set of node labels and (4) $Ø$ is a labelling function that maps nodes to the set of labels and a label can be shared among many nodes, $Ø : V \rightarrow L$. This thesis does not consider any edge label in formalising a tree structured data.

A tree has a distinguished *root* node v_0, and for any other node v_i, there is a unique path from v_0 to v_i. A tree contains no cycle. A *cycle* is a path in which the first and the last node of the path are the same.

A *path* is a sequence of consecutive edges between two nodes in the tree. The length of this path is defined by the number of edges. Each node v_i of a tree has a unique path from its position to root v_0. The *size* of a tree denoted as $|T|$ is the total number of nodes present in tree T.

A tree structure poses several hierarchical relationships - parent-child, ancestor-descendant and sibling relationships - among its nodes, as shown in Figure 2.6.

The *parent* of v_i (and $v_i \neq v_0$), is the adjacent node of v_i in that unique path to v_0.

The *ancestors* of v_i, are all the other nodes in that unique path except v_i itself.

The *children* of v_i is the immediate follower nodes of v_i, the number of the children is also known as *fan-out*, denoted by f_i.

The *descendants* of v_i are the list of all follower nodes of v_i.

Sibling nodes share the same parent, so a sibling relationship exists between nodes that originate from the same parent node.

Definition 2.1 (Depth, Height, Level): For node v_i, the length of the unique path is called the depth of that node in tree T, denoted by $d(T, v_i)$. The height $h(v_i)$ of a node v_i in a tree is the longest path from that node to a leaf. The height $H(T)$ of a tree is the height of root $h(v_0)$. The level of a node v_i in a tree T is, $Lv(T, v_i) = H(T) - d(T, v_i)$.

According to this definition, the root node of a tree is positioned at the highest level.

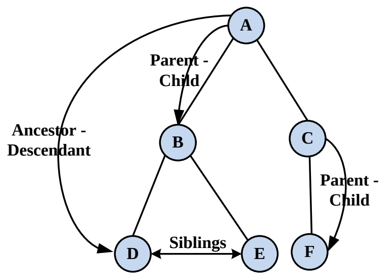


Figure 2.6: Hierarchical relationships amongst tree nodes

Definition 2.2 (Tree Isomorphism) Let two trees denoted by $T_1 = (V_1, E_1, L_1)$ *and* $T_2 = (V_2, E_2, L_2)$ respectively. A tree isomorphism is a bijective function f: $V_1 \rightarrow V_2$ satisfying

(1) $L_1 (V_1) = L_2 (f(V_1))$ *for all nodes* $v_i \in V_1$

(2) *for each edge* $e_1 = (v_i, v_j) \in E_1$, *there exists an edge* $e_2 = (f(v_i), \mathrm{f}(v_j)) \in E_2$

If a bijective mapping exists between the set of nodes of two trees T_1 and T_2, which preserves and reflects the tree structures, then these trees are isomorphic to each other, denoted as $T_1 \cong T_2$. The automorphism corresponds to isomorphism of a tree to itself.

2.2.1 Types of trees

There are many types of trees. Based on the topology, three types of trees are listed below:

Definition 2.3 (Free Tree) A *free tree* is connected, acyclic and undirected whose edges have no direction. Therefore, it has no designated root node.

Definition 2.4 (Rooted Unordered Tree) A *rooted unordered tree* is connected, acyclic and directed, which has a distinguished root node from which all other nodes can be reached. A root node does not have any incoming edge. For a rooted unordered tree, there is no predefined or fixed left-to-right order among siblings; only ancestor-descendant and parent-child order are defined.

Definition 2.5 (Rooted Ordered Tree) **A** *rooted ordered tree* is connected, acyclic and directed and also has a designated root node. In this tree type, the predefined order among siblings exists along with ancestor-descendant/ parent-child relations.

This research emphasises using rooted unordered trees and free trees for mining useful information from them, but the rooted ordered tree type is also discussed to provide a general tree type concept.

2.2.2 Types of subtrees

Subtrees play an important role in tree mining. They are a portion of a tree data structure that can be considered as a tree itself. Formally, the tree T' with node set V'

and edge set *E´* is a subtree of a rooted tree *T* with node set *V* and edge set *E* iff (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) the labelling of *V′* and *E′* is preserved in *T′* according to *T*. There are different types of subtrees that are also well known for their wide usage in various tree mining algorithms, but the following discussion is provided based on the research focus of this thesis.

Definition 2.6 (Induced Subtree) For a rooted labelled tree *T* with node set *V* and edge set *E*, a tree *T′* with node set *V′* and edge set *E′* is called an *induced subtree* of *T* iff (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) the labelling of *V′* is preserved in *T′* according to *T*, and (4) $(v_1, v_2) \in E'$ if and only if v_1 is a parent of v_2 in *T*. In other words, the induced subtree *T′* is a subtree that keeps the parent-child relationship among the vertices of the tree, *T*. In the case of defining it for a rooted ordered tree, on top of the above mentioned conditions, the left-to-right ordering among the siblings in *T′* should also be preserved.

Definition 2.7 (Embedded Subtree) For a rooted labelled tree *T* with node set *V* and edge set *E*, a tree *T′* with node set *V′* and edge set *E′* is called an *embedded subtree* of *T* iff (1) $V' \subseteq V$ (2) the labelling of *V′* is preserved in *T′* according to *T* (3) $(v_1, v_2) \in E'$ where v_1 is the parent of v_2 in *T′* only if v_1 is an ancestor of v_2 in *T* and the set of ancestors of $(v_2 \in V') \cap$ the set of ancestors of $(v_2 \in V)) \neq \varphi$. In simple words, an embedded subtree *T´* preserves an ancestor-descendant relationship among the nodes of the tree, *T*. If it is an ordered embedded subtree, besides other conditions, the left-to-right ordering among the siblings in *T′* should also be preserved. Examples of induced and embedded subtrees for a tree *T* are given in Figure 2.7.

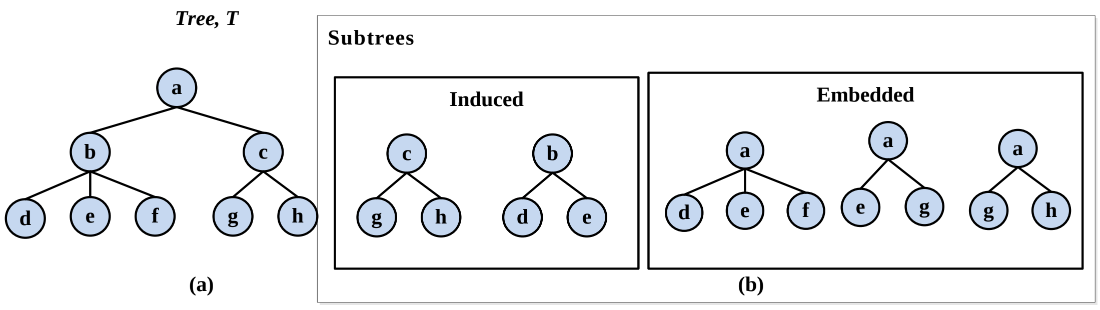


Figure 2.7: Examples of induced and embedded subtrees (b) for a tree, *T* (a)

Discussion: From the above discussion, it is certain that all induced subtrees are embedded subtrees but vice-versa is not true. Embedded subtrees can be considered as a generalised form of induced subtrees. Based on a tree type, properties of the trees can be defined. For example, the order among siblings does not need to be preserved for an unordered tree. Trees with the different permutation among siblings' order will still be considered the same. This property leads to the concept of isomorphic trees. According to the nature of desired information, different subtrees need to be mined. If the parent-child relationships are the main focus in the tree data, induced subtree mining should be performed. Mining of embedded subtrees would result in undesired outcomes in those situations. For example, if one is interested in characterising a particular disease then induced subtrees are essential to mine, due to the fact that some features of the dataset may have a similar set of values, and it is necessary to indicate which value belongs to which particular feature. On the other hand, if the captured relationships are to be generalised to those of ancestor-descendant nodes in the trees, then the focus should be shifted towards mining embedded subtrees that allow detection of information embedded deeply within the tree structure. In summary, both induced and embedded subtrees carry important information and hence, proposing algorithms to mine these subtrees is significant.

2.3 TREE REPRESENTATION

Semi-structured data, as known as tree data, has no fixed schema or class. It is implicit, irregular, nested and heterogeneous [85] which makes it more complex to be mined in comparison to the flat-representation data [86]. Mining tree structured data requires a rigorous pre-processing to get it prepared for further processing or manipulation. The data should be cleaned, transformed, and formatted before using it as input to a data mining task [87]. The pre-processing step takes a lot of time, but it is essential for discovering meaningful information. This thesis focuses on efficient representation of the tree data in order to apply mining techniques directly. This section covers the state-of-the-art methods of tree representation. Some of the most popular tree representations, such as canonical form, adjacency list and adjacency matrix that have been used in the algorithms of frequent subtree mining and tree matching, are discussed below.

2.3.1 Tree Traversal

Tree traversal refers to the approach of visiting all nodes of a tree in a systematic way [62, 83]. This allows the tree structured data to be represented as list data in order to facilitate knowledge discovery. Two basic traversal schemes for ordered trees are *preorder* and *postorder* traversals [83]. In a *left-to-right* preorder traversal, the root of a tree is visited first, and then the subtrees rooted at its children are visited recursively from left to right. (The children are visited from right to left recursively in a *right-to-left* preorder traversal)

On the other hand, in a *left-to-right* postorder traversal, before visiting root node, first all of the subtrees rooted at its children are visited recursively from left to right. (In a *right-to-left* postorder traversal, these children are visited from right to left recursively.) In the literature often the *left-to-right* preorder or postorder is simply referred to as preorder or postorder [83, 88].

In tree mining algorithms mainly preorder traversal are used to encode trees. *Depth-First Search* (*DFS*) and *Breadth-First Search* traversals [83] are the most popular pre-order schemes, which have been widely used in encoding both ordered and unordered trees [89, 90]. According to [62], these traversals can be defined as follows:

Definition 2.8 (Depth-First Search) *Depth-First Search* (*DFS*) is a preorder traversal that visits tree nodes following its depth.

In Figure 2.8 (a) the traversal order using DFS traversal for the given tree will be A-B-A-C-B-C-C-B-A.

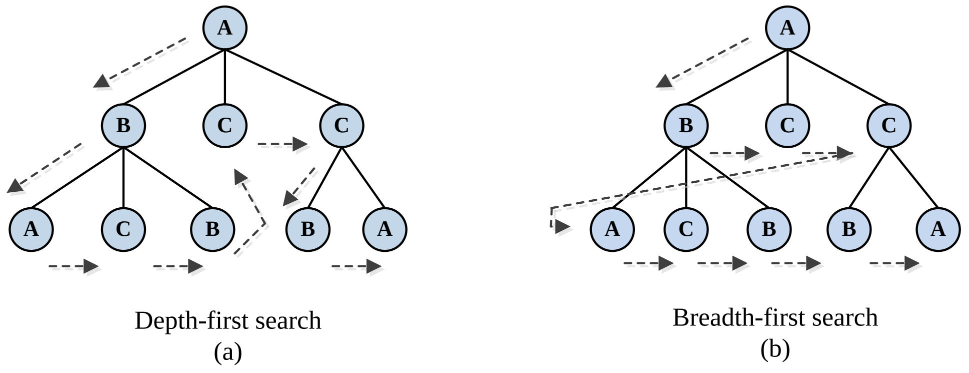
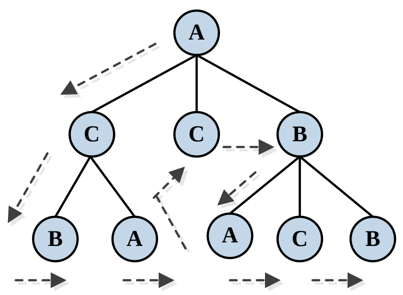
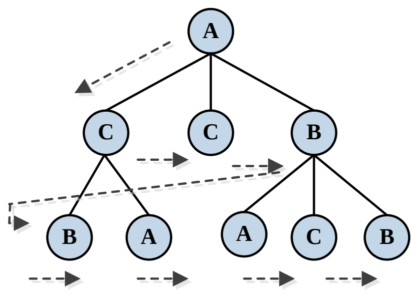


Figure 2.8: Examples of the depth-first search (a), and the breadth-first search (b). The dotted arrow lines show the traversing directions



Depth-first search
(a)



Breadth-first search
(b)

Figure 2.9: The Depth-first search (a), and the Breadth-first search (b) traversing orders for the same example tree of Figure 2.8, but with a different sibling order

Definition 2.9 (Breadth-First Search) *Breadth-First Search* (*BFS*) is a preorder traversal that visits the tree nodes following breadth or "level by level" after the root traversal; all its children are processed next, then all of their children, etc. down to the bottom level.

For the given tree in Figure 2.8(b), BFS traversal of nodes will be in this order: A-B-C-C-A-C-B-B-A.

Being left-to-right preorder traversal in both DFS and BFS schemes, the tree nodes are visited iteratively from left-to-right order following depth and breadth respectively. Both DFS and BFS are used widely to encode ordered and unordered trees. The traversing order of a tree should be unique so that it can be used to encode a tree distinctively. It is considered as a first step to define a canonical form of the tree. In order to maintain an accurate tree indexing in various tree mining algorithms, the traversing order and encoding play an important role.

Discussion: From the above description, it is clearly understandable that both the DFS and BFS traversal visit the sibling nodes by preserving an order from left-to-right, which implicitly forces the properties of an ordered tree. Using the BFS and DFS traversing orders for encoding ordered trees will not raise any issue, however, for unordered trees these two schemes encode two similar unordered trees (only varied in sibling order) differently, which causes various issues like isomorphism in frequent mining and false similarity measure in tree matching. The example tree in Figure 2.9 is the same tree as Figure 2.8 with the only difference of position of sibling nodes. For an unordered tree, the position of sibling nodes can be exchanged and the trees with varied sibling orders will still be considered the same. The DFS and BFS orders for the trees in Figure 2.9(a) and (b) are A-C-B-A-C-B-A-C-B and A-C-C-B-B-A-A-C-B respectively, which are different orders than those listed before for the example tree in Figure 2.8. Since these are the same unordered trees, their encodings should be the same, but the DFS and BFS traversal orders will lead different encodings for them due to enforcing the left-to-right order. During tree manipulation, these trees will be treated differently and may result in incorrect answers. This prompts the need for developing an alternative traversing approach as well as an unordered tree encoding scheme without relying on the left-to-right sibling order.

2.3.2 Canonical Form

The canonical form (CF) of an entity (or tree) is a representative form that can consistently represent many equivalent variations of that entity into one standard [83, 90]. It can be considered as a bijective mapping function that maps a tree along with all of its equivalent variant trees in a database into a unique identity, which ensures efficient processing of many tree mining algorithms.

In the literature, various canonical forms for representing trees have been proposed [63, 90-92]. A canonical representation is normally referred to as string encoding, which is a compact and memory efficient way of representing the tree data [83, 90]. Besides, the string encoding provides an efficient data access mechanism. Often, canonical form and canonical form string encoding are used interchangeably. To build a canonical form, the nodes of a tree are stored in the string encoding following a traversing order. Based on the DFS or BFS traversing order, the state-of-the-art canonical forms can be classified as follows.

Depth-first Canonical Form* (*DFCF*) *String Encoding

The DFCF string encoding utilises the DFS order of a tree. It is usually built by adding the label of the tree nodes in a depth-first order with a special backtrack symbol that is not in the label alphabet. The backtrack symbol is used whenever, in accordance with the traversing order, the encoding needs to come back from a child node to its parent node. Different backtrack symbols such as ‘$’, ‘/’, ‘↑’ or -1 have been used by researchers [70, 90, 93, 94]. Another symbol “#” is commonly used to indicate the end of the string encoding. All of these special symbols should not be in the node labels set. The DFCF using ‘$’ for backtrack for the example tree in Figure 2.8(a) would be ABA$C$B$$C$CB$A$$. Asai et. al. [92] and Nijssen & Kok [91] independently defined a similar string encoding for rooted ordered trees using depth sequences, where they explicitly store the depth of each node within the tree. For example, the depth sequence for the tree used in the previous examples will be “(0;A); (1;B); (2;A); (2;C); (2;D); (1;C); (1;C); (2;B); (2;A)” or equivalently “0A1B2A2C2D1C1C2B2A”.

Breadth-first Canonical Form* (*BFCF*) *String Encoding

The BFCF string encoding is obtained by storing the label of each node in accordance with the BFS traversing order, level by level. Additional symbols “$” and “#” are added that should not be in the label alphabet. “$” is used to separate the families of siblings and “#” is used to indicate the end of the string encoding. The breath-first encoding for the previous example in Figure 2.8(b) will be “A$BCC$ACB$$BA#”.

The BFCF representations have been also utilised by many researchers, especially in various frequent subtree mining algorithms [49].

2.3.3 Canonical Representation for Unordered and Free Trees

Canonical Representation for Rooted Labelled Unordered Tree

Defining the canonical form for unordered trees is not as simple as for the ordered trees. For an unordered tree, many possible ordered tree variations are available. All of these ordered trees will actually map the same unordered tree, therefore they should be treated as the same unordered tree for doing further manipulation like frequent subtree mining or clustering. Therefore, the canonical form of unordered trees should be defined in a way that will ensure unique identity to all of its *isomorphic* trees.

Figure 2.10 shows the example of a group of isomorphic trees which hold an exact bijective map to each other and preserve the same tree structure. These trees represent the same unordered tree. The concept of the presence of isomorphic trees in a database is known as isomorphism [62, 82].

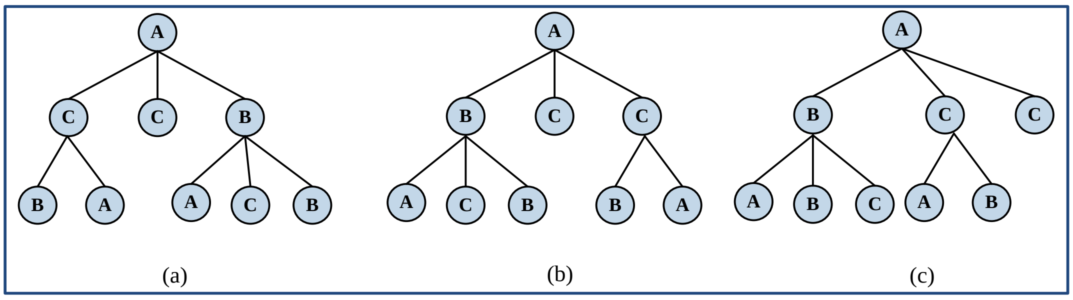


Figure 2.10: Example of isomorphic trees

To deal with this representational issue of unordered trees, many researchers proposed to choose a representative of the isomorphic trees and then use the canonical form of the representative tree for all isomorphic trees [49, 92, 95]. To describe this canonical form, the breadth-first canonical string (BFCS) is used here; the encoding proposed by Chi et al. is used as an example [90, 96]. First all possible rooted ordered trees and the corresponding breadth-first string encodings are obtained by assigning different orders among the sibling nodes. Then, according to the lexicographic order, the minimum breadth-first strings of the ordered trees is defined as the breadth-first canonical form of the rooted unordered tree. Consider the example in Figure 2.10, where for three different rooted ordered trees, the breadth-first string encodings are:

(a) "A$CCB$BA$$ACB#",

(b) "A$BCC$ACB$$BA#",

(c) "A$BCC$ABC$AB#".

According to the minimum lexicographic order, the BFCF string encoding "A$BCC$ABC$AB#" will become the canonical form of these trees and other isomorphic trees.

Any of the breadth-first search and depth-first search-driven preorder scheme can be used to define the canonical form of an unordered tree in similar manner. Chi et al [90] defined a depth-first search canonical form (DFCF) of unordered trees. In another work, Chi et al have defined the canonical form based on breadth-first search order [96]. Asai et al. [92], Nijssen & Kok [91] and Zaki [70] also proposed similar canonical form in their works. Hadzic et al. also utilised similar canonical forms in [97, 98]. The most recent contribution to encoding process of unordered tree is found in [53], which also used the preorder to encode the tree nodes.

Canonical Representation for Free Trees

A free tree is unrooted and unordered in nature, which make its representation even harder than the rooted unordered tree, as it can possibly be represented in multiple ways due to having different choices for the root. To define a canonical form for a free tree, the root node is defined uniquely at first by repeatedly removing the leaf node at a time along with its incident edge until one or two nodes remain [54, 63, 64, 96]. If a single node remains then the free tree is called centred, whereas if two nodes remain then the free tree is called bicentred [10]. A free tree is either centred or bicentred.

Ruckert et al. [54], Nijssen et al. [66], and Chi et al. [63, 96] have shown for a centred free tree, the centre can be designated as the root, and the tree becomes a rooted unordered tree. The canonical form for the rooted unordered tree then can be used to define the canonical form of the transformed tree (centred free tree). If a free tree is bicentred, the tree can be imagined as two pieces of a free trees, each of which is rooted in one of the bicentres, and therefore a canonical string can be obtained by comparing the string encodings of two subtrees based on the lexicographic order [54, 96].

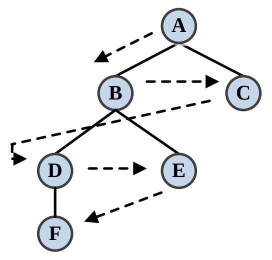
Discussion: From the above description, it can be ascertained that the canonical forms of trees have been developed based on the BFS and DFS traversal approaches. For the rooted ordered trees, there is no issue in the CF representation due to the order dependency. An ordered tree cannot have any other variations or isomorphic trees. A BFCF or DFCF string encoding can represent an ordered tree identically. However, for unordered trees this is not true. An unordered tree can have several isomorphic trees. Researchers have proposed various solutions to choose a representative isomorphic tree and use its CF for all. However these processes depend upon costly operation such as sorting. A method is yet to be proposed that ensures a unique identity of a rooted unordered tree without performing an expensive operation to find the lexicographically minimum BFCF or DFCF string encodings from the available ordered variations of an unordered tree. The same applies to free trees, since the CF of a free tree uses the CF of a rooted unordered tree as its canonical form after deciding the root node. For a bicentred free tree, if an approach can be proposed to define the root node or to make it centred then again the sorting can be avoided to define the CF of a bicentred free tree.

In general, the majority of tree mining algorithms (e.g., frequent subtree mining) use a canonical form for representing trees and then processing to obtain patterns. A novel breakthrough in tree representation will save the cost of the overall process.

2.3.4 Adjacency List and Adjacency Matrix

Adjacency List and Adjacency Matrix are two common forms of tree representation for pairwise comparison. Generally, an *Adjacency List* representation of a tree consists of each node along with its collection of adjacent nodes and edges. This basic idea may vary, depending upon how the association between a node and its adjacent collection is detailed [62]. On the other hand, an *Adjacency Matrix* used a matrix form to represent the adjacency information of each node of a tree. From this representation, it can be understood which nodes of a tree are adjacent to which other nodes.

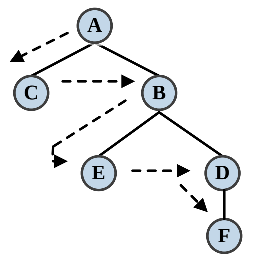
The adjacency list is more space efficient than the adjacency matrix, but can be cumbersome when a tree node has lots of adjacent edges. Usually, when the data is sparse, then an adjacency list is preferred over adjacency matrix, but it is vice versa when the data is dense. An adjacency matrix allows fast computation in case of checking or comparing trees; more specifically, when it is needed to check whether two nodes are adjacent to each other or not. An adjacency matrix can be even used as canonical form while doing frequent pattern mining [99, 100], but due to the compact size, the string encoding representation has become popular, since the frequent mining process includes some complex steps like frequency counting and candidate generation. The adjacency matrix representation can be considered useful for finding approximate similarity scores between trees. Therefore, the adjacency matrix representation can be considered as more appropriate for the tree mining tasks in which the similarity calculation is required.



BFS
A-B-C-D-E-F

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	1	1	0
C	0	0	0	0	0	0
D	0	0	0	0	0	1
E	0	0	0	0	0	0
F	0	0	0	0	0	0

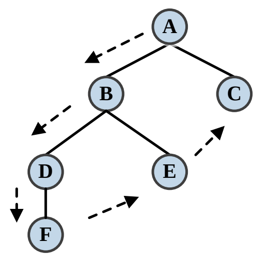
(a)



BFS
A-C-B-E-D-F

	A	C	B	E	D	F
A	0	1	1	0	0	0
C	0	0	0	0	0	0
B	0	0	0	1	1	0
E	0	0	0	0	0	0
D	0	0	0	0	0	1
F	0	0	0	0	0	0

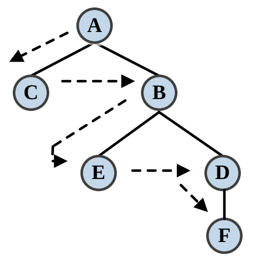
(b)



DFS
A-B-D-F-E-C

	A	B	D	F	E	C
A	0	1	0	0	0	1
B	0	0	1	0	1	0
D	0	0	0	1	0	0
F	0	0	0	0	0	0
E	0	0	0	0	0	0
C	0	0	0	0	0	0

(c)



DFS
A-C-B-E-D-F

	A	C	B	E	D	F
A	0	1	1	0	0	0
C	0	0	0	0	0	0
B	0	0	0	1	1	0
E	0	0	0	0	0	0
D	0	0	0	0	0	1
F	0	0	0	0	0	0

(d)

Figure 2.11: Adjacency matrix representation using BFS and DFS order

Discussion: The traditional adjacency matrix representation has some issues, since it uses a pre-order traversal for encoding the tree nodes and then populates the adjacency information in a matrix form, therefore the same issues highlighted for traversing unordered trees also held true during the node encoding of the adjacency matrix. It is well understood now that an unordered tree can form many ordered variations, but during the knowledge discovery process, all these variations should correspond to the same unordered tree. The representation of all these ordered tree variations should be the same. Adjacency matrix representation uses a pre-order to arrange the rows and column which contains the adjacency information. The usual practice is to use either a depth-first search or breadth-first search traversal to get that pre-order. Since both DFS and BFS preserve a left-to-right order among sibling nodes, the adjacency matrix is not unique for all variations of an unordered tree. To elaborate on this, consider the example in Figure 2.11 where, two trees (a) and (b) are two ordered variations of the same unordered tree due to the variations in sibling nodes only. The BFS orders are different: for (a) "ABCDEF" and for (b) "ACBEDF", which eventually build two different adjacency matrices for the same unordered tree. Similar results are obtained while using DFS orders for constructing adjacency matrices as shown in Figure 2.11(c) and (d).

Other limitations of Adjacency matrix representation exist. Semantic information of nodes in the tree cannot be represented in an adjacency matrix. Moreover, for a tree structure, an adjacency matrix just shows the relationship of parent-child. The information of ancestor-child cannot be represented. It cannot precisely depict the difference of positions of different nodes. Value 1 is used to merely indicate that there is a link between two nodes; it is not able to distinguish different situations. For example, the level importance of a node is not equal, which it fails to express. This representation can be improved by inserting the ancestor-descendant relation; the information about existence of a node, etc. This thesis proposes an improvised adjacency matrix that includes more hierarchical and semantic information.

2.4 TREE MATCHING

Tree matching is fundamental to the core operation of many data manipulation tasks such as clustering analysis, nearest-neighbour classification, data integration, data cleansing and data querying [60, 81, 101]. The tree matching problem refers to the problem of finding a similarity (or distance) score between tree pairs by means of some comparison [60]. The concept of similarity or distance can be expressed using a *distance function* (*dist*). Let a tree database T_{db} contains trees $\{T_i, T_j, T_k\}$. *dist*: $T_{db} \times T_{db} \rightarrow R^+$ be a mapping function that defines a distance between each pair of trees of a database. For example, the similarity between trees T_i and T_j can be expressed by *dist*(T_i, T_j), which will give a distance or similarity score between these two trees. This distance function is treated as distance *metric* if it satisfies the following conditions:

1. *dist*(T_i, T_j) ≥ 0 (non-negativity)
2. *dist*(T_i, T_j) = 0 iff $T_i = T_j$ (coincidence axiom)
3. *dist*(T_i, T_j) = *dist*(T_j, T_i) (symmetry)
4. *dist*(T_i, T_k) ≤ *dist*(T_i, T_j) + *dist*(T_j, T_k) (triangle inequality)

A myriad of tree mining methods have been developed for finding similarity between tree pairs. The majority of them are applicable for ordered trees, and very few are available for unordered trees due to the complexities involved with unordered tree processing [42]. These methods are developed based on nodes, paths, subtree representations, higher order model and many more [42, 102, 103]. Amongst these varieties, the tree edit distance is the most widely used method for tree matching. Some other methods are also available based on level similarity, frequent pattern, matrix computation, etc. This section mainly details the available methods for unordered tree matching, including their pros and cons in general. The discussion on various tree matching algorithms spans across two major areas: the tree edit distance-based methods; and the other methods not using an edit distance operation. The tree edit distance methods use string representation and other methods use a non-string representation such as vector, matrix, and tensor etc. Figure 2.12 provides an overview of the tree matching approaches used for unordered tree comparison.

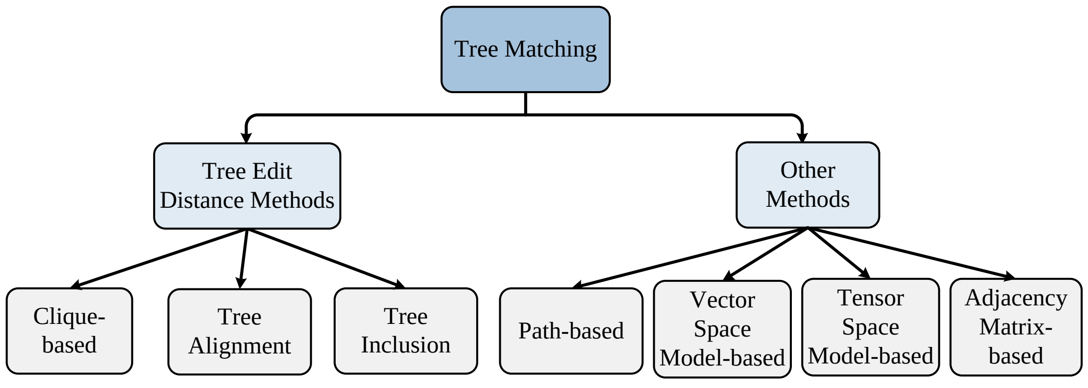


Figure 2.12: An overview of various tree matching approaches

2.4.1 Tree Edit Distance based Methods

Amongst these method varieties, tree edit distance is the most widely used for tree matching [42, 104, 105]. Tree edit distance methods utilises the string representation of trees and, for the strings of characters; a particular syntax of string is used in many programming languages to represent regular expression. Researchers found the string edit operation-based tree edit distance methods convenient. Moreover, for ordered trees, the string representation usually consumes less memory [42, 106]. This method measures the distance between two trees by the minimum cost to transform one tree into another tree by applying a sequence of edit operations, which are constrained to be metric, such as deletion, insertion and substitution of nodes. The *tree alignment distance* problem is a special case of the tree editing problem, which can be considered as a restricted edit distance where all insertions must be performed before any deletions [42]. It only uses insertion and deletion as edit string operations. The *tree inclusion* problem is another special case of the tree edit distance problem, which only uses deletion as an edit string operation to calculate the distance [42]. T_i is included in T_j iff deleting nodes from T_j gives T_i. In *clique based approach*, a tree edit distance is reduced to a clique problem, and then a clique solver is used to solve the problem. [107, 108].

Many tree matching algorithms have been developed based on these problems. For an ordered tree, the edit distance-based algorithms are known to exhibit $O(n^2)$ complexity [109, 110] (where n is the maximum size of the two input trees), whereas for an unordered tree, the tree edit distance problem is found NP-hard [42, 48, 111]. The tree edit distance and the alignment problems for unordered trees have even been shown as MAX SNP-hard in literature [45, 47].

To avoid this computational intractability, researchers have developed algorithms constrained to conditions such as tree size and other tree properties; however they result in compromising on accuracy [42]. Akutsu et al [112] introduced an algorithm under fixed parameters, which exhibited improved complexity of $O(2.62^k.poly(n))$ (where k is the maximum allowed edit distance), however, it performs poorly for comparing non-similar trees. Horesh et al. [113] developed an A* algorithm which can efficiently compare unordered trees of moderate size but only under the unit cost distance (i.e., the cost of each edit operation is 1).

Variant	Type	Time	Reference
Tree edit distance			
General	O	$O(n^3)$	[114]
General	O	$O(n^2)$	[109, 110]
General	U	Max SNP-hard	[45, 47]
Constrained	O	$O(\|T_i\|\|T_j\|)$	[115]
Constrained (bounded height, *h*)	U	$O(h)$	[116]
Less-constrained	O	$O(\|T_i\|\|T_j\|I_i^3I_j^3\ (I_i+I_j))$	[104]
Less-constrained	U	Max SNP-hard	[104]
Unit-cost	O	$O(u^2\min(\|T_i\|, \|T_j\|)\ \min(L_i, L_j))$	[117]
Unit-cost	U	$O(2.62^k\cdot\text{poly}(n))$	[112]
Bounded degree trees	U	$O((1 + \varepsilon)^{\|Ti\|+\|Tj\|})$, for any fixed $\varepsilon > 0$	[118]
1-degree	O	$O(\|T_i\|\|T_j\|)$	[119]
Tree alignment distance			
General	O	$O(\|T_i\|\|T_j\|(I_i+I_j)^2)$	[43]
General	U	Max SNP-hard	[43, 45]
Tree inclusion			
General	O	$O(\|T_i\|\|T_j\|)$	[120]
General	O	$O(\|\sum T_i\|\|T_j\|+ m_{Ti,Tj}\ D_2)$	[121]
General	O	$O(L_i\|T_j\|)$	[122]
General	U	NP-hard	[44, 123]
Clique-based			
General	U	Not defined/ calculated	[40, 124]
Others (Pattern Matching)			
Tree contraction pattern-based	U	NP complete	[125, 126]
Largest common subtree (constrained)	U	Polynomial	[112]

Table 2.2: Time complexity of various tree edit distance-based methods, here O = ordered tree and U = unordered tree, adopted from [42]

In most recent times, some methods have been developed by reducing the tree edit distance problem to a clique problem [40, 79, 108, 124]. For example, Fukagawa et al [40] proposed a method of computing maximum clique, in which an instance of tree edit distance is directly transformed into an instance of the maximum vertex weighted clique problem, and then it is solved using a clique solver [127]. This method can work efficiently on moderate sized trees, but it will be slow for the large sized trees. This method is further improved with using dynamic programming that repeatedly solves instances of the maximum vertex weighted clique problem as subproblems [124]. However, this method still suffers from high complexity for large tree structures with many leaves. Some similar reductions [128, 129] and methods of variants of the tree edit distance problem [107] have been proposed, however none of them exactly solves the formal tree edit distance problem for unordered trees. Some of the available tree edit distance-based methods may work efficiently for some particular tree shapes (i.e., by constraining height, size, etc.) but will degenerate for others by arising unpredictable, even infeasible runtime [46, 110, 116].

Apart from the tree edit distance, some other string representation based tree matching methods are proposed using pattern matching [102], maximum agreement subtree [26, 130], smallest common super tree and largest common subtree [131], tree contraction pattern [125, 126]. Unfortunately, these methods also provide unfavourable results for unordered trees by exhibiting high computational complexity [42]. In Table 2.2, some of the available tree edit distance based algorithms for both ordered and unordered trees are listed including their complexities.

2.4.2 Other Methods

Due to the high complexity involved in tree edit distance methods, researchers have attempted to calculate the approximate similarity score between tree pairs using the similarity function on Vector Space Model (VSM), Adjacency Matrix (AM) and Tensor Space Model (TSM) of tree representation. Trees represented as VSM can be compared using distance measures such as Cosine, Euclidean, Manhattan, Jaccard, Dice, etc. [127, 132]. A comprehensive survey on various distance measures can be found in [133]. Though these methods have reported as computationally efficient, VSM representation has its own limitations. It is a feature vector that contains information about tree content only, the structural detail in a tree such as hierarchical relationships cannot be captured through this representation.

In response to this need, researchers have developed methods based on AM representation for doing the tree computation. Romanowski et al. [23] proposed a method for matching unordered trees by employing the minimum weighted symmetric difference metric. Authors in [77] attempted to calculate the similarity between unordered trees by considering the shape or geometrical structure, where a Orthogonal Procrustes method was used to calculate the similarity score. But again, the AM representation also has some limitation; it only contains the adjacency information of nodes, whereas for representing tree structure, some other pieces of information like ancestor-descendant relationship, fan-out and level are also required, especially when these equivalent representations of trees are going to be used further for calculating the similarity score between tree pairs.

Recently, the TSM representation is used that can capture both the tree content and structure information for tree matching, however, it faces high computational complexity due to high dimensionality and sparsity [134]. Another family of algorithms (path-based method) uses the level similarity concept by counting the common nodes in the corresponding levels of two trees where each level has different weight assigned. These methods fail to preserve the child-parent relationship among tree nodes [51, 72, 135-137], which is an important differentiating factor for trees. Besides, these methods have been proposed for ordered trees only.

Discussion: Much research has been conducted in the area of ordered trees, but the methods for unordered tree matching are still underway due to immense computational complexities. Many important problems in the research fields of genetics, bioinformatics and web intelligence emphasise the need for developing efficient methods of manipulating unordered trees [21, 29]. The nature of an unordered tree mining problem is harder than that of an ordered tree due to its less constrained structure, which results in complex tree mapping.

The structure of a tree plays an important role in differentiating the data; therefore, the dependencies inherent in a structure need to be captured efficiently. The representation format of a tree heavily affects the performance and complexity of the algorithm [46, 138]. Due to having a less constrained expression of hierarchical dependencies, the representation of an unordered tree for further manipulation is trickier and challenging. Apparently the lack of efficient equivalent representation raises the complexity in tree mapping as well as increasing the computational complexity of executing tree manipulation algorithms [42].

Researchers have tried to solve the tree matching problem using tree edit distance and have built polynomial algorithms for ordered trees. Unfortunately, for unordered trees, the tree edit distance problem have been shown as NP-complete, even MAX SNP-hard, which means unless P = NP there is no polynomial time approximation scheme [45, 47], therefore no tractable solution is available following this approach. Because of the high complexity yielded by tree edit distance-based methods, measuring similarities of unordered trees is still an open problem. In comparison to tree edit distance-based algorithms, the other methods that use vector and matrix comparisons seem to be more promising as they allow faster computation than the edit string operations. However, the majority of these methods have proposed solutions for ordered tree matching only.

This thesis conjectures that an efficient equivalent representation of the exact tree structure may propose the effective solution. In comparison to the rooted ordered trees, the unordered tree representation is way more challenging and the existing representation methods are lacking in efficient representation due to their structure or order dependent scheme. This causes an accuracy issue in tree matching with the presence of isomorphic trees. If the representation problem can be solved then, in comparison to the edit string-based method, the other methods may find a faster solution.

In this thesis only the database of rooted unordered trees is considered for addressing the tree matching problem. Since a free tree is very similar to graph data, it is usually discussed under the main stream of graph matching [12, 139], which is another vast area of research, therefore no separate study on free tree matching is carried out here.

2.5 FREQUENT PATTERN MINING

With explosive growth in structured data that presses the need for insight information, frequent pattern mining has generated much interest in the data mining community. It is a basic step in *association mining* [3, 11, 140] and a pre-requisite in many other data mining tasks such as *sequence mining* [85-87, 141]; *multi-dimensional patterns* [83, 107]; *maximal pattern mining* [36, 142]; *emerging pattern mining* [110]; *clustering* [111, 130, 132] and *classification* [50]. Generally the problem of frequent pattern (or, subtree) mining can be stated as identifying the common patterns based on a user-specified support which is called minimum support, denoted by (*min_sup*). The terms “frequent pattern mining”, “frequent tree mining” and “frequent subtree mining” are interchangeably used in this thesis. Formally the frequent subtree mining problem can be defined as:

Given a tree database $T_{db} = \{T_1, T_2, ..., T_n\}$, find a list of frequent subtrees $S = \{t_1, t_2, . . . , t_r\}$, such that for every $t_r \in S$, $support(t_r)$ >= *min_sup*, where $support(t_r)$

is the percentage of Trees in T_{db} that contain t_r. The support definition may vary, which is discussed later.

Mining frequent patterns is significant and the overall process requires several non-trivial steps. Candidate generation and frequency counting are two main steps in a frequent pattern mining algorithm, which are in general very expensive in terms of memory and time [49, 143]. Because of the complex nature of the frequent pattern mining problem, many efforts have been made to propose different approaches for solving this problem. The available works in the literature on frequent pattern mining can be classified based on several factors as shown in Figure 2.13. Most of them will be covered in the following discussion but with a stronger focus on the unordered and free tree mining algorithms.

Most of the frequent tree mining algorithms (including the proposed one in this thesis) adopt the basic ideas from frequent itemset mining algorithms which mainly consist of two steps:

- Candidate generation step
- Frequency counting step

2.5.1 Candidate Generation Step

This step generates candidate trees so that their frequencies can be analysed and a list of frequent patterns can be generated. Given a database, all trees are represented in their canonical form such as BFCF, DFCF, adjacency matrix or adjacency list [90, 96, 144, 145]. The candidate generation step can be performed using various algorithms such as the *apriori* algorithm [140], vertical mining algorithm [49], hybrid or combination of apriori and vertical mining [96], and many others. The vertical mining algorithms have recently become popular due to their relatively small memory footprint as compared to apriori algorithms – the most widely used algorithm for candidate generation step in frequent subtree mining.

In vertical mining algorithms, the concept of an enumeration tree is used. All of the candidate trees will be generated into this tree following a traversal strategy, which can be breadth-first, depth-first, or a combination of the two. In the breadth-first search approach, the search for an appropriate candidate is performed level-wise. First, all size 1 trees are generated and counted, which are basically frequent labels; then from the frequent 1 trees, candidate 2 trees are constructed and verified to be frequent and the process continues. In the depth-first search, the enumeration tree is traversed following depth. In this case, first from a single frequent 1 tree, all possible candidate trees will be generated and validated; then it will start processing another frequent 1 tree. The third approach is to use a combination of depth-first search and breadth-first search traversal, which means that the candidate trees will be generated following both breadth and depth.

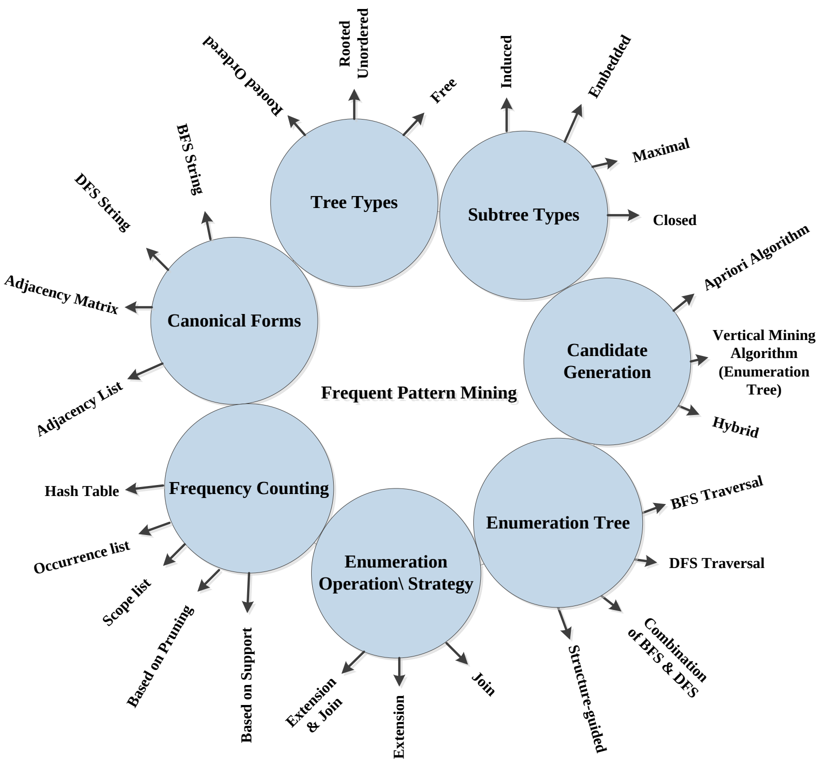


Figure 2.13: Overview of various frequent pattern mining approaches

The BFS traversal requires more space since at each enumeration the generated subtree would not have the final frequency count yet, whereas, the DFS traversal is space efficient, even for processing a long pattern, because every enumeration will compute a frequency count of each generated subtree completely [38, 49]. Depending upon the type of enumeration process, various operations or strategies can be adopted to grow the enumeration tree by generating candidates. These are:

- Enumeration by Extension
- Enumeration by Join
- Structure Guided Enumeration

The extension approach, also known as right-most-path extension, is a commonly used technique for growing the enumeration tree for both ordered and unordered trees. For any type of subtree, the right-most path extension method is reported to be complete and non-redundant (i.e., all valid candidates are enumerated at most once) [38, 91]. By following the extension operation, adding a frequent label at the right most path of the existing frequent *K*-tree will generate a new candidate *K*+1-tree. Usually this operation is used in DFS traversal or vertical mining.

The join operation, also referred as the guided extension process, is mainly used in the enumeration tree where a combination of BFS and DFS traversal is employed [38, 70, 96, 100]. When the cardinality of the node label is very high, using an only extension operation can be exhaustive and inefficient. Given frequent *K*-trees, candidate *K*+1-trees are formed by joining a pair of *K*-trees that have a common *K* - 1 prefix (node along with tree structure). The BFS traversal and the combined DFS and BFS traversal usually adopt this operation for their candidate generation.

Both extension and join operations result in a huge number of candidates and not all of them are valid or, frequent. Therefore, to reduce the number of candidates generation, the apriori heuristic [140] has been applied, i.e., “if length *K* pattern is not frequent in the database, its length (*K* + 1) super-pattern will not be frequent”. As the process generates a lot of candidates and then requires adopting a full pruning process, the overall complexity of the step to enumerate and generate candidates is very high. An improved candidate enumeration technique is desirable and will be considered as an important contribution in this research field.

An idea of utilising a structural model for efficient enumeration proposed in [91, 146, 147], suggests generating only valid candidates by guiding the candidate generation process using the available information on XML Schema. The candidates that confirm the available schema are only considered valid. This idea can be utilised by considering the tree structured data information as the guidance scheme. In Figure 2.14, an example is given for the task of mining frequently occurring rooted induced unordered subtrees. Now based on the underlying structure (e.g., available hierarchical relationships, leaf node, root node etc.) of the database, the candidate generation is guided for obtaining only the valid subtrees. Here, the valid subtrees are only those that confirm their existence according to the tree structure scheme of the considered database. By following this enumeration technique, a large amount of memory and time can be saved, as it will allow skipping the record of invalid subtrees, which are not going to be frequent anyway and therefore, are needed to be pruned in between the process. Besides, this approach can complement the join approach by overcoming its existing limitation, i.e., avoiding generation of invalid subtrees. Depending upon the tree representation process, this scheme information will vary.

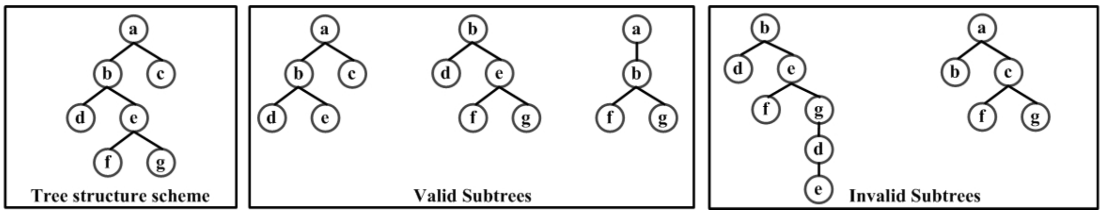


Figure 2.14: An example of valid and invalid subtree, considering the underlying information of the sample database while mining frequent induced subtrees

Besides the above mentioned enumeration process to find out the frequent subtrees, another technique is also reported in the literature, which can obtain frequent subtrees without candidate generation. This is called pattern growth [148], based on FP-tree [149]. A pattern-growth approach does not perform level-by-level candidate enumeration; rather, it works by constructing a compact database utilising the *FP-tree structure,* which is an extended prefix-tree structure for storing compressed and significant information about frequent patterns. Although the FP-tree based method avoids costly and repeated database scans by giving a compact representation of a large database, it comes with its own limitations. This process can end up having a lot of projected databases in accordance to each of the frequent prefix substructures, which causes huge expense because of the recursion process to reach the different node and FP-growth [5, 150]. Apart from this common problem, another problem with mining frequent unordered trees is that the FP-tree can't avoid the expensive task of sorting canonical forms to avoid the isomorphism. The projected database can also become large and as well, the number of pseudo projection steps can be bigger in comparison to that of the ordered trees, which causes thrashing of memory.

In general, the excessive candidate generation, large memory foot print, memory thrashing issue, and costly I/O processing are the shortcomings of the candidate generation step [143, 151].

2.5.2 Frequency Counting Step

In this step, the occurrences or frequencies of candidate trees are counted to calculate their supports to determine if they are frequent, whilst the infrequent ones have to be pruned. This step needs to be very efficient since the number of candidates to be counted can be huge.

A conventional approach is direct checking, which generally uses a hash-tree data structure to count the frequency [140]. For each generated candidate, its frequency is increased by one if it exists in the transaction; FP-tree based frequent mining techniques also use conceptually similar hast-tree to count the frequency.

Another widely used approach is the occurrence list-based approach, which associates an occurrence list with each candidate subtree [38, 90, 146]. A vertical representation is used to store a list of Ids of the transactions that support the candidate subtree; therefore by simply checking the size of the occurrence list one can determine whether the corresponding candidate subtree is frequent or not. In the literature, this approach is found faster than direct checking [38, 90]. Another scope list-based frequency counting approach is also proposed by the researchers which is also computationally effective [70].

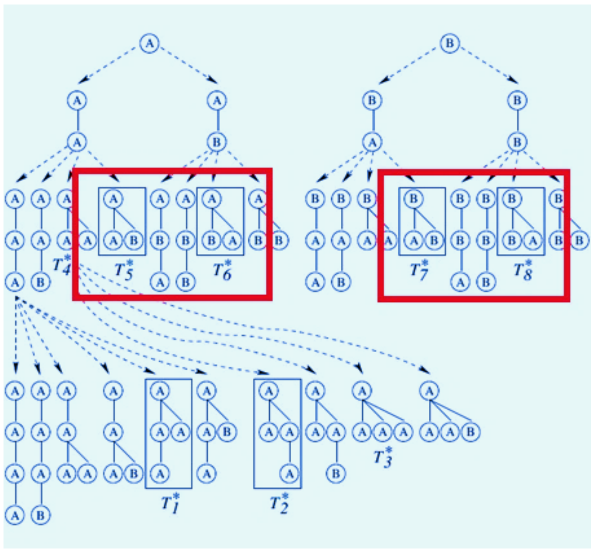


Figure 2.15: Isomorphism issue during candidate generation step of mining frequent unordered tree using enumeration tree [49]

Removing the infrequent subtrees or pruning is also a part of the frequency counting step. Based on the pruning techniques, a frequency counting step can give varied performance. The two most common pruning techniques are full pruning and opportunistic pruning [11, 49, 70]. Full pruning is time consuming, but will confirm the completeness, whereas opportunistic pruning can be beneficial when a dataset contains long patterns and can afford to miss out some of the frequent patterns.

Different support definitions are also found to be used for determining the frequent trees. The most commonly used one is a conventional support which is sometime referred to as transaction-based support [5]. The transaction-based support count of a subtree is defined as the total number of transactions that contain it; here a transaction is referring to a tree. Most of the existing subtree mining algorithms use this support definition [49]. On the contrary, occurrence-match or weighted support count of a subtree is defined as the total number of occurrences of that subtree in all of the transactions [70, 146]. Occurrence-match support can produce pseudo-frequent subtrees; a detailed discussion about this is provided in [55, 97, 98].

2.5.3 Algorithms for Mining Frequent Rooted Unordered and Free Trees

Generally, mining unordered subtrees is a more difficult problem than mining ordered subtrees. For mining ordered trees, only the ordered subtrees need to be enumerated, whereas for mining unordered subtrees one additional checking is required in candidate generation to avoid the isomorphism problem [70]. This extra computation is essential to determine which subtrees are isomorphic to each other. Otherwise, the many isomorphic trees will be generated, which makes the candidate generation process redundant and eventually leads toward counting incorrect frequency. In Figure 2.15, an example is shown, where the enumeration tree is generating candidate trees for a rooted unordered tree, and the red rectangles are used to show some of the isomorphic trees that should not be generated more than once as a candidate. Because these are the same subtrees, a checking mechanism is required for avoiding such generation, which is an expensive sorting or ordering process of canonical forms. The success of a frequent mining algorithm for unordered subtrees largely depends on efficient enumeration and canonical form transformations [98] as well as on avoiding expensive canonical sorting [142]. This thesis works toward achieving such goals. In Table 2.3, the list of available algorithms for mining rooted unordered and free trees are classified, based on their types and mining approaches.

Algorithm Type	Mining Approach	Algorithms
Tree Type	Rooted Unordered Tree	uFreqt, Unot, RootedTreeMiner, HybridTreeMiner, CMTreeMiner, UNI3, UITree, SLEUTH, TDU, U3, Treefinder
	Free Tree	Chi's FreeTreeMiner, HybridTreeMiner, Rückert's FreeTreeMiner, F3TM
Subtree Types	Induced Subtree	uFreqt, Unot, RootedTreeMiner, HybridTreeMiner, CMTreeMiner, UNI3, UITree, Chi's FreeTreeMiner, HybridTreeMiner, Rückert's FreeTreeMiner, F3TM
	Embedded Subtree	SLEUTH, TDU, U3, Treefinder
Canonical Form (Pre-order based String Representation)	DFS traversal	uFreqt, Unot, SLEUTH, UNI3, U3, UITree, Chi's FreeTreeMiner, Rückert's FreeTreeMiner
	BFS traversal	RootedTreeMiner, HybridTreeMiner, Rückert's FreeTreeMiner
	DFS or, BFS traversals	F3TM
Enumeration Tree	BFS traversal	RootedTreeMiner, Rückert's FreeTreeMiner
	DFS traversal	uFreqt, Unot, RootedTreeMiner, F3TM
	Combination of BFS & DFS traversals	PathJoin, HybridTreeMiner
	Structure Guided	UNI3, U3
Enumeration Operation	Extension	uFreqt, Unot, RootedTreeMiner, F3TM
	Join	PathJoin, Rückert's FreeTreeMiner, Chi's FreeTreeMiner
	Extension & Join	HybridTreeMiner, SLEUTH, UITree
Frequency Counting	Occurrence List	uFreqt, Unot, PathJoin, RootedTreeMiner,
	Based on Pruning	F3TM, Chi's FreeTreeMiner
	Scope list	SLEUTH

Table 2.3: A general classification of the available frequent subtree mining algorithms for rooted unordered and free trees

Algorithm for Mining Rooted Unordered Induced Subtrees

For finding unordered frequent tree patterns, most of the proposed algorithms use a canonical form and extend only candidates that are in the canonical form. A sorted pre-order string canonical form that can be obtained in linear time was first defined by [94] and the frequent subtree mining algorithm was developed accordingly. A few more similar canonical representations based on either depth-first traversal or breadth-first traversal have been defined [90-92, 96]. But, all these canonical forms need an additional isomorphism test for avoiding the redundancy problem during the frequency counting step, which results in more run time for processing the frequent subtree mining algorithm.

To deal with the computational complexity, some researchers played with the varied frequency counting approaches to improve the algorithmic efficiency. Asai et al. [92] proposed an algorithm, uNot that mines induced unordered subtrees by using a reverse search technique for incremental computation of unordered subtree occurrences. Another algorithm Ufreqt, proposed by Nijssen & Kok [91] is designed to mine induced subtrees based on a bottom-up strategy for determining the frequency. Both the uNot and Ufreqt algorithms use the concept of an occurrence list-based frequency count. In UITree algorithm [53], the authors use an early termination or early pruning technique for boosting up the algorithm performance while mining frequent induced subtrees.

Variations are also found in the candidate generation step, such as the Chi et al. proposed [90] RootedTreeMiner, which is a vertical mining algorithm and conceptually a re-implementation of uNot. Later, as an extension to their previous work, the authors proposed the HybridTreeMiner [96] algorithm that can systematically enumerates all induced subtrees; it uses a hybrid concept for candidate generation that utilises both the apriori and vertical mining algorithm. PathJoin [152] assumes that children of every node are labelled identically and finds maximal patterns using vertical mining algorithm-based candidate generation that utilises only a join operation to grow. Another algorithm, UNI3 [98] was proposed for mining unordered induced subtrees and for candidate generation; it uses structure guided enumeration that is associated with a right path extension operation to grow the enumeration tree, but this algorithm is designed for working on a database of labelled ordered trees. Recently, another algorithm was proposed based on a compression tree sequence but it is designed for mining frequent condensed subtree (i.e., maximal induced subtree) mining [15]. Some other similar works are also found in the literature, based on condensed representation of unordered trees [18, 28, 150].

HybridTreeMiner [96] and UNI3 [98] have been used in benchmarking the proposed BOSTER algorithm. The process of growing the enumeration tree in HybridTreeMiner is similar to BOSTER, but it is not structure guided. Whereas, UNI3 uses the structure guided enumeration tree but only utilising extension operation to grow it. Similar to BOSTER, both the HybridTreeMiner and UNI3 use the canonical form for storing the trees. Moreover, HybridTreeMiner is the most widely used method for benchmarking and UNI3 is a recent method.

Algorithm for Mining Rooted Unordered Embedded Subtrees

The majority of the existing unordered subtree mining methods work with induced subtrees and very few are available for mining unordered embedded subtrees. SLEUTH [70] was one of the first techniques to mine frequent embedded unordered subtrees and used a scope-list join via the descendant and cousin tests for growing the enumeration tree. Chehreghani et al. [142] developed the TDU algorithm to mine unordered embedded subtrees, which was reported as a faster algorithm because of avoiding isomorphism checking, but it only mines maximal subtrees, which are subsets of the all frequent embedded subtrees that SLEUTH discovers. Hadzic et al. also proposed an algorithm, U3 [97], based on the structure guided enumeration to mine frequent unordered embedded subtrees from a database of labelled ordered trees. Another algorithm for mining frequent embedded unordered subtrees is Treefinder [153], which uses an Inductive Logic Programming approach for mining, but this process does not guarantee completeness (can miss many frequent subtrees), especially at a lower support. Besides these approaches, another apriori based frequent mining algorithm FRESTM is proposed which has used a restricted tree edit distance technique to detect restrictedly rooted unordered embedded subtrees [36]. Since, the tree edit distance problem is already known for exhibiting high complexity for an unordered tree, the overall performance of this algorithm can be affected. Moreover, this algorithm yields low recall in comparison to other algorithms and misses some patterns, which is not desirable in many cases. Another algorithm, EvoMiner, is proposed, where the phylogenetic tree is considered as a rooted unordered embedded subtree but with some restricted properties; therefore, it is not exactly solving the general embedded subtree mining problem [37].

For comparing the proposed work – BEST [58] for frequent rooted unordered embedded subtree mining – U3 [97] and SLEUTH [70] are used as benchmarks in this thesis. U3 uses a structure guided enumeration similar to BEST, although BEST utilises different tree information for guiding the candidate generation. SLEUTH is commonly used benchmarking algorithm as well as it adopts the extension and join concepts for candidate generation. Moreover, both of these U3 and SLEUTH utilise the canonical form based representation. So it facilitates testing of BEST for performing against the existing canonical form based works.

Algorithm for Mining Free Subtrees

Compared to mining rooted unordered trees, mining free trees is more complex, since it has no root node specified. Many possible variations of the same free tree can exist, which need to be reduced during candidate enumeration. Because of the complexities involved, only a handful of free tree mining algorithms are available in the literature. Chi et al. have presented an apriori-like algorithm FreeTreeMiner [90] which uses apriori based algorithm for candidate generation. Then for reducing the memory usage, another algorithm, HybridTreeMiner, is used based on a combination of apriori and vertical mining algorithms for candidate generation [96]. Both of these algorithms are designed for working on databases of labelled free trees. Rückert et al. [54] and Zhao et al. [64] have proposed algorithms for mining frequent free trees from a graph database. These algorithms generate large number of false positives (i.e., invalid candidate subtrees) during enumeration, which need to be pruned in the frequency counting step. This results in high processing time. Moreover, the necessity of performing isomorphism checking to avoid redundant candidate tree and false frequency counting causes additional computational complexity.

For the benchmarking purpose, FreeTreeMiner [90] and HybridTreeMiner [96] algorithms are used due to their good performance record as well as for the relevancy with the proposed work, FreeS. HybridTreeMiner uses both the extension and join operations to grow the enumeration tree, as well as it uses the occurrence list-based frequency counting method. Both of them use canonical form for representing trees. They come closest to FreeS in terms of the algorithmic design and enable a fair comparison.

Discussion: Mining frequent unordered trees and mining free trees are advantageous in many cases over mining frequent ordered trees; however, in comparison to ordered tree mining these two fields require more maturity. Frequent ordered tree mining methods already face high computation and memory expense issues; for unordered and free trees the complexity turns even higher. Although some works have been done to mine frequent rooted unordered and free subtrees, the exponential candidate generation with redundancy and the isomorphism issue are there. The available algorithms lack a systematic enumeration process as well as an efficient frequency counting process. It is also critical to determine a good growth strategy, as there can be many possible ways to extend a candidate subtree due to not having the sibling order constraint. Therefore, an optimal enumeration strategy for a tree-structured pattern is highly sought after. There should be algorithms for mining both induced and embedded unordered trees, because each of them has different applications and needs. Besides, during mining frequent free subtrees, the whole candidate generation process becomes trickier. The confirmation of candidate generation in canonical form of free tree requirement is essential, which demands additional care. Since the free trees are more flexible than rooted unordered trees, the number of isomorphic trees can be huge. Clearly the frequent free tree mining process requires an efficient canonical form as well as candidate enumeration approach, which are missing in the existing state-of-the-art algorithms.

2.6 CONCLUDING REMARKS

Undoubtedly, mining frequent subtrees and finding tree similarity information as a course of knowledge discovery are significant. Any data mining task for unordered (both rooted and unrooted) tree databases faces additional challenges over the ordered tree databases, due to the flexibility of data representation; however, the need for developing techniques of knowledge discovery from unordered tree databases is inevitable.

From the literature review, it can be noticed that the representation of unordered or free trees is not as straight-forward as ordered trees because of its less constrained structure. The existing representation methods (i.e., tree traversal, canonical string representation, and adjacency matrix) lack in dealing with the isomorphism and automorphism problems, which are the most pressing issues in unordered (both rooted and unrooted) tree representation. The field of unordered trees calls for a novel representation that can overcome this issue. It would be beneficial if the canonical form for both rooted unordered trees and free trees can be proposed, which will help in avoiding the isomorphism/automorphism checking step during candidate generation. Moreover, the present enumeration processes are found to be memory and time inefficient. An optimal enumeration approach is therefore needed to accelerate the unordered tree mining process which can resolve the exponential candidate generation issue. The technique to boost up the traditional frequent counting approaches should also be explored.

Besides mining induced unordered subtrees, embedded subtrees also need to be mined, since they carry additional information that is interesting to some of the significant applications. Compared to induced unordered subtrees, not too many algorithms are available for mining embedded unordered subtrees, due to the complex nature of this problem. Serious attention should be directed to this topic. Similarly, the field of frequent free tree mining lacks efficient algorithms despite its importance in various domains. The canonical representation of a free tree faces additional challenge due to the fact of being unrooted, which also makes the enumeration process in free tree mining challenging.

For tree matching, most of the available methods provide unfavourable results in terms of time and space complexities for unordered trees. Most of string-edit based matching problem exhibit NP-hard complexities; some of them are even Max SNP-hard. Apart from tree edit distance based methods, some other approaches seem to be promising but yet require improvisation, especially in choosing the right data structure. Instead of using string representation for comparing trees, matrix-based representation can be considered for facilitating fast computation of similarity metrics. However, it is essential to investigate whether the available similarity metrics will support this representation while differentiating the trees.

In summary, the following research gaps can be highlighted after reviewing the literature:

- Lack of current tree representation methods including tree traversing, canonical form and adjacency matrix for rooted unordered and free trees.
- Lack of efficient and scalable tree matching algorithms for unordered trees.

– Lack of efficient frequent rooted unordered induced and embedded subtree mining algorithms.

– Lack of efficient frequent free subtree mining algorithms.

All the important achievements of the considered works to date have been highlighted, while some of the problems that remain outstanding are pointed out and will be addressed in this thesis. In particular, a number of development needs is evident:

– An efficient tree traversal approach that will encode all ordered variations of an unordered tree uniquely.

– An efficient tree representation, i.e., canonical form, adjacency matrix, which will resolve the isomorphism issue of unordered trees and will also capture some other important tree information.

– A faster and memory efficient tree matching approach for unordered tree that can resolve the current complexity issues.

– An optimal and measurable enumeration strategy for a tree-structured pattern that improves on the enumeration operations

Despite the present research progress in the field of tree mining, the persistent limitations in unordered and free tree mining algorithms are hard to be overlooked. The majority of the algorithms developed for unordered trees exhibit high complexity. Though restricting tree properties allows achieving polynomial algorithms, this raises the issues of non-completeness and compromising accuracy. Apparently the lack of efficient equivalent representations raises the complexity in tree mapping, which results in higher complexity in further tree manipulation. Conducting research in this direction to resolve the highlighted limitations is significant and much needed.

Chapter 3: Tree Representation and Data Structure

Tree structured data has become ubiquitous because of its capability to portray widely available information hierarchically. Much popular domain data (e.g., XML, Weblog, BOM, etc.) can simply be considered as a manifestation of tree structured data [19-21, 23, 24]. In previous chapters, it is noted that the problem of knowledge discovery from databases of unordered trees which are less constrained in structure is compelling and useful. This thesis will concentrate on developing mining techniques from databases of rooted unordered and free trees. Mining these tree types is challenging as highlighted in the literature review especially for the tasks of frequent subtree mining and tree matching. The current state-of-the-art algorithms are lacking in achieving optimal processing, which promotes the development of new efficient and scalable techniques.

Representation is a fundamental and essential component for conducting efficient manipulation of tree structured data [154]. The previous chapter detailed the different representation techniques utilised in the existing frequent subtree mining and tree matching algorithms. From that discussion, it is clear that the existing representation techniques are deficit in appropriate encoding of rooted unordered and free trees, which apparently hampers the efficiency performance of mining methods. An improved tree representation technique should be able to improve the performance of mining algorithms by offering appropriate encoding and optimal processing.

This chapter summarises the contribution of this thesis in the area of tree representation and shows how the different representation techniques are related and developed. It will help to link with the other contributions in the thesis since these representations are discussed in detail while presenting the corresponding method.

The process of tree representation is not just concerned with how the actual subtree is modelled and represented in memory; it is also concerned with how the complex computation and data manipulation tasks can be performed efficiently and effectively. The tree representation methods are developed, focusing on the static aspects of trees. The static aspect refers to the typical data representation, which demands improvisation according to the literature review. Whereas the dynamic aspects refer to data operations used in designing the algorithm mechanisms; this is covered to some extent in this thesis.

The chapter starts with the proposed tree traversal algorithm BOS that provides an optimal encoding of the tree [39, 155]. A description of the data structures, canonical forms and adjacency matrix, which are utilised to ensure efficient processing of the proposed tree mining algorithms [39, 57-59, 155, 156], is included in subsequent sections. Other data structures that amplify the performance of the proposed algorithms - such as dictionary and occurrence list - are presented next.

This chapter includes only the essential introductory material on the proposed representation forms, and puts them all together in a single chapter to give an overview. As discussed in Table 1.2, the full detail of BOS will appear in Chapter 4, adjacency matrices in Chapter 4 and canonical forms in Chapter 5 in the form of published papers.

3.1 THE BALANCE OPTIMAL SEARCH (BOS) ALGORITHM

The existing schemes for traversing trees provide different encodings for the variations of the same rooted unordered tree, which cause problems in tree mining algorithms (as discussed in Sub-section 2.3.1). A new tree traversal algorithm, named as *Balance Optimal Search* (BOS), is proposed based on the concept of optimisation [39] (detailed description can be found in Chapter 4 as outlined in Table 1.2). Due to having the order-independent scheme, the new traversal algorithm encodes all variations of the same rooted unordered tree identically.

To propose the BOS traversal algorithm, the tree traversal problem is reduced to the *Simple Assembly Line Balancing* (SALB) problem, which is a well-studied optimisation problem in the Operations Research (OR) paradigm [65, 157]. SALB is a combinatorial optimisation problem that chooses an optimal path for a network by avoiding the exhaustive search. In the literature, SALB has been used to solve networks in manufacturing problems that are represented by a predecessor digraph, i.e., a graph holding all properties of an unordered tree [65, 158]. This thesis conjectures that SALB can propose an optimal path for visiting an unordered tree like a network if the tree traversal problem is reduced to a SALB problem.

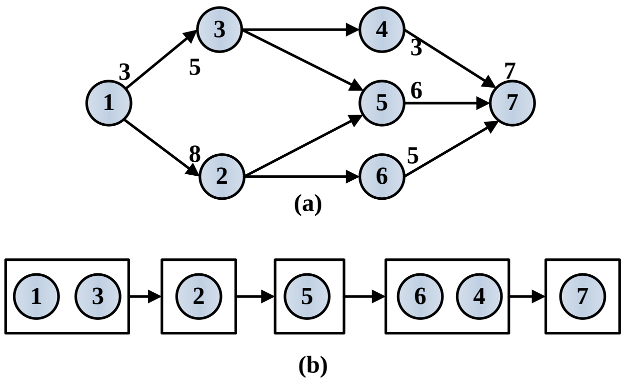


Figure 3.1: The simple assembly line balancing problem, (a) replicates an assembly line, (b) represents an optimal sequence of tasks on various machines

3.1.1 Simple Assembly Line Balancing (SALB) Problem

In manufacturing, the SALB problem is used to minimise the cost of production by balancing an assembly line [65, 157]. An assembly line is a sequence of linearly ordered stations where each station performs several machine tasks repeatedly during each cycle of the assembly line. The cycle of an assembly line is fixed; therefore each station must complete all the tasks in a way that the whole product can be delivered within the cycle time to avoid any delay. It becomes essential to identify the best possible sequence of tasks that will balance an assembly line. The solution of the SALB problem should conform to achieve an optimal sequence of tasks in the assembly line by ensuring minimum delay.

In Figure 3.1(a) an assembly line is shown using a predecessor digraph where the nodes are representing various tasks performed by different machines and the numerical values outside the nodes stand for the task time required for each machine. The tasks cannot be assigned to the station arbitrarily because of the sequencing requirement. This sequence constraint can be considered similar to the concept of ancestral constraint, which poses a partial order among the set of tasks. Hence a task can only be completed after completion of all of its predecessor tasks. In Figure 3.1(b) the optimal sequence of the completion of tasks is shown in accordance to the assembly line in Figure 3.1(a).

3.1.2 The BOS Traversal

In the proposed method an assembly line is a metaphor for an unordered tree which maps the parameters of assembly line to the parameters of a tree (e.g., tasks as tree nodes). Therefore the tree traversal problem can be reduced to the SALB problem and the mathematical model of the optimisation problem can be developed accordingly. This model is formulated with an objective function of minimising the computational cost of the overall traversal process. The other constraints are set by following the basic properties of a tree structure and restrictions as per tree traversal. By solving this model, an optimal sequence of tree nodes can be found, where if a tree is traversed, the minimum computational cost can be ensured. It is in the same line as the SALB problem that obtains the optimal sequence of performing tasks with an objective function of minimising delays.

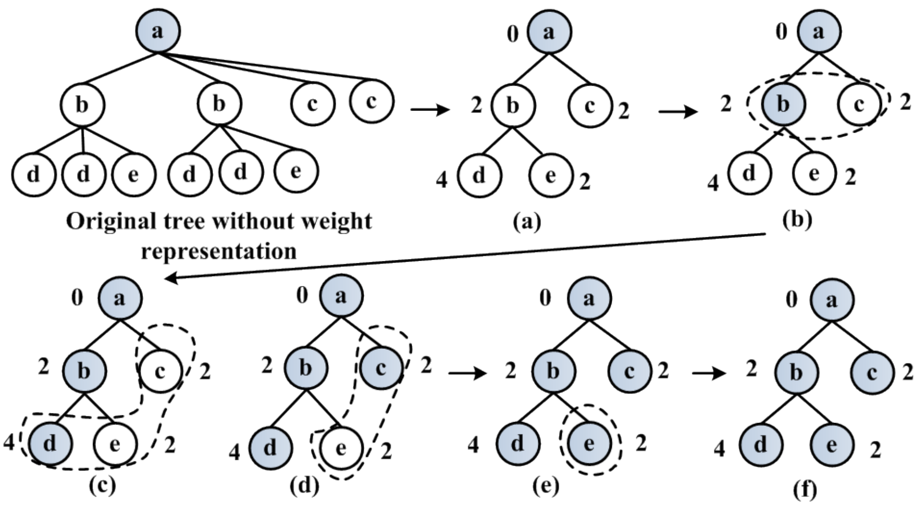


Figure 3.2: The BOS traversing order of the given tree is v_a-v_b-v_d-v_c-v_e. The arrow is directing the sequence of steps that traversing process is carried out and the highlighted nodes are showing the list of nodes that have traversed

The technical details of obtaining the optimal traversal sequence are provided in Chapter 4. A simple example is given in Figure 3.2 to show how the BOS traversal will encode a rooted tree. A rooted unordered tree is provided in Figure 3.2(a), where each node is associated with a numerical value. These numerical values are referred to as weights in this thesis. A weight is calculated by counting the number of appearances of a node under its parent node; the detail of this definition is provided in Chapter 4 and 5. Following BOS traversal, first the root node v_a will be traversed, therefore, its immediate followers or child nodes v_b and v_c will become eligible to traverse next. In the case of having multiple eligible nodes, the node that has highest weight will be chosen for traversing next. For this example, both the eligible nodes have same weight, but v_b is chosen as it has the maximum fan-out. After traversing

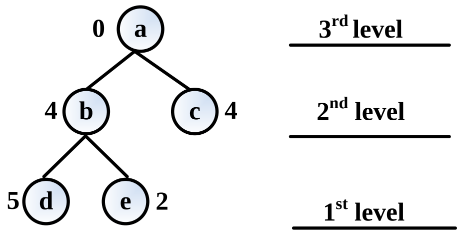
v_b, its child nodes v_d and v_e become eligible along with v_c. v_d is chosen next because of having maximum weight. Following this the final traversing order will be v_a-v_b-v_d-v_c-v_e.

The BOS traversal can encode an unordered tree effectively since its working approach is structure independent and it does not consider the sibling constraint. It is based on optimality and a variation of the unordered tree, due to swapping the order between siblings, will be treated equally during the optimisation modelling. In the SALB problem, the tasks that are initiated from the same immediate predecessor do not have any specific order in execution and hence changing the order of these tasks does not change the optimal point [65]. BOS ensures a unique traversing order as well as a unique encoding for a rooted unordered tree (and all its variations) which the other traversal approaches fail to provide. Using this uniqueness of BOS order, effective adjacency matrix and canonical form can be derived which may take the performance of tree mining algorithms to the higher level of efficiency.

The BOS traversal can also be used to traverse a free tree. A free tree is also unordered therefore the order independent traversing strategy of BOS is suitable for its encoding too. BOS is designed to work for a rooted tree; hence after identifying the root node of a free tree, the BOS traversal can be applied to it. Paper 6 shows the proof and lemma that BOS can be used to define the canonical form of free trees [59].

3.2 ADJACENCY MATRIX

Representing unordered trees is challenging than the ordered tree, due to the less constrained structure. Among various methods, a commonly used tree representation is a matrix that allows for simplifying computation of tree mining algorithms [62]. Adjacency matrix is a popular matrix representation of trees [159] that depends on the encoding scheme. For the same unordered tree T, there can be $|T|!$ different adjacency matrices using different permutations of the set of nodes [160]. It is not possible to get a unique adjacency matrix representation for the variations of the same unordered tree using any of the DFS and BFS traversal based encoding, as these encodings rely on sibling order. Moreover the traditional adjacency matrix only shows the adjacency information among the nodes, whereas trees have other important information that can be portrayed in their representation.



(a)

	a	b	d	c	e
a	1	2/3+4	1/3	2/3+4	1/3
b	0	1	1/2+5	0	1/2+2
d	0	0	1	0	0
c	0	0	0	1	0
e	0	0	0	0	1

(b)

Figure 3.3: Augmented adjacency matrix

In this thesis, a new Augmented Adjacency Matrix (AAM) using the BOS encoding is proposed, which has the ability to encode an ordered variation of the same unordered tree identically. AAM includes additional level information and weight information of nodes, which ensure rich portrayal of a tree structure.

3.2.1 Augmented Adjacency Matrix

This is a square matrix representation of a rooted unordered tree that utilises the BOS encoding, node level and node weight information of a tree to represent the cell values [39, 155].

Encoding information: The BOS order encoding is derived using the balanced optimal search traversing algorithm, which is unique for an unordered tree and its variations. The root node becomes the first row and column to be represented in the matrix and the other nodes are arranged in accordance to BOS order.

Level information: The level information in a tree represents the ancestor-descendant relationships of the nodes. This structural information is important for finding similarity between trees. The level information is generated from the node level based on their hierarchical relationships, which is explained in Chapter 4.

Weight information: The nodes in a tree carry a weight displaying how frequently the node occurs under its parent node. Besides including the node weight, an additional weight value of 1 is added to each diagonal cell of the adjacency matrix to represent the existence of a corresponding node on that tree.

In Figure 3.3, an example of AAM representation is shown. The level of the tree nodes are shown according to their position. The BOS order of the given tree

Figure 3.3(a) is v_a-v_b-v_d-v_c-v_e and the nodes are arranged accordingly. The diagonal cells are populated with a weight value 1 to confirm the node existence. The other or off-diagonal non-zero values of the cells are a summation of level information and weight information. The weight information is coming straight from the number of occurrences of a node under its parent node. If for a cell the respective nodes have a parent-child relation, then the weight value is added (the node relation should be read from row to column) into it and if the nodes have an ancestor-descendant relation then the level information will be added with the weight. The AAM resolves the issue of having different matrix representation for the isomorphic unordered trees. The incorporation of additional implicit information in tree representation allows more accuracy in tree matching, which is reported later in Chapter 4.

3.2.2 Extended Augmented Adjacency Matrix

Extended Augmented Adjacency Matrix (EAAM) is an extension of AAM that includes the frequent subtree information for imaging a tree [156]. By incorporating sub-tree information, EAAM includes a much richer structural relationship importance, in addition to ancestor relationship, in tree representation. Due to its use of BOS encoding, it ensures unique identity of a rooted unordered tree.

Frequent mining algorithms provide information on frequent structural dependencies like parent-child and siblings in a particular database. They provide the list of frequent sub-trees that, in turn, detail the most occurred parent-child or ancestor-descendant and sibling relations. A data structure such as an unordered tree has a vast flexibility; characterising the structural relationships based on frequent occurrence will aid in the global similarity calculation. Adding the frequent substructure as a representational component can be advantageous for tree structure processing like similarity measures. This is the inspiration behind proposing this new adjacency matrix.

Structural relationship importance weight: Based on the result of the frequent subtree mining algorithm, the structural relationships are characterised and the weights are defined accordingly. If a subtree is frequent then the inherent parent-child relation is considered as mandatory. Once all the mandatory parent-child or ancestor-descendant relationships are identified, the remaining relationships are classified as optional. During the EAAM representation, a weighted value of 1 and 0 are used to represent the mandatory and optional relationship respectively.

In the previous AAM representation, the off-diagonal non-zero entry of a cell is either level information or the summation of level information and node weight, but in EAAM the structural relationship importance weight will be also added based on the frequent information of the corresponding nodes. This representation is incorporated in the proposed tree matching algorithm, which is found useful and accurate in finding similarities between trees, as reported in Chapter 4.

3.3 CANONICAL FORMS FOR LABELLED ROOTED UNORDERED TREES

A key problem of mining unordered trees is the representation issue. Several ordered variations of an unordered tree are possible and during representation these multiple ordered trees should be mapped to one canonical form of an unordered tree. These trees vary in the order of sibling nodes only; the information contained within the structures is essentially the same. An example is given in Figure 3.4, where the four ordered trees are same if the sibling constraint is relaxed. Since the unordered tree can have many isomorphic trees as well as it can have automorphism, the canonical form representation becomes challenging.

This thesis presents a new Balanced Optimal Canonical Form (BOCF), which is proposed following the balance optimal search (BOS) traversing order. The BOCF ensures representing all isomorphic ordered variations of an unordered tree with a single canonical form.

3.3.1 The Balanced Optimal Canonical Form (BOCF)

BOCF is defined using the order of optimal search traversing [57, 58]. It is a string representation of a tree that records the label of each node along with its weight following the BOS order. This string representation includes four unique symbols, +1, -1, +2 and -2, to represent the breadthwise movement from sibling to sibling and depth-wise movement from a child to its parent. The symbols +1 and -1 are used for depth-forward and depth-backward travel respectively. The symbols +2 and -2 are used for breadth-forward and breadth-backward travel respectively. It is assumed that the alphabet of node labels includes none of these symbols.

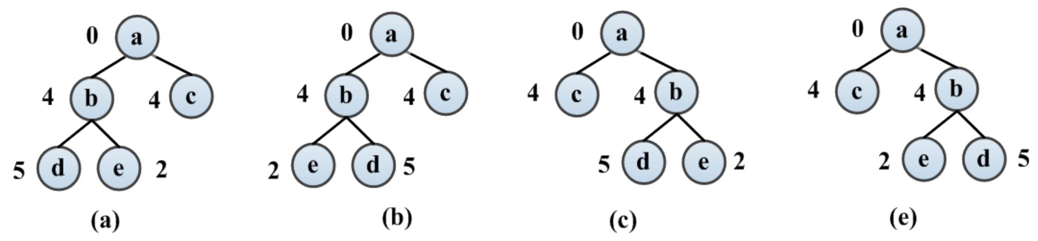


Figure 3.4: An example of four rooted ordered tree variations of the same rooted unordered tree

An Example: The balance optimal search (BOS) traversing order is v_a-v_b-v_d-v_c-v_e for all four trees in Figure 3.4. This order is unique for all the variations of a tree relaxing the sibling constraint. If each tree given in Figure 3.4 is treated as rooted ordered, the BOCF string encoding will be:

(a) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, +2, 2v_e";

(b) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, -2, 2v_e";

(c) "0v_a, +1, 4v_b, +1, 5v_d, -1, -2, 4v_c, +2, +1, +2, 2v_e";

(d) "0v_a, +1, 4v_b, +1, 5v_d, -1, -2, 4v_c, +2, +1, -2, 2v_e".

It can be noted that these BOCFs only vary in terms of breadth movement which shows that sibling order is preserved. If a tree is treated as unordered, the order of siblings is ignored and only the breadthwise movement from the existing rightmost sibling node is permitted. The BOCF string encodings for the trees, viewed as unordered, given in Figure 3.4 will be:

(a) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, +2, 2v_e";

(b) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, +2, 2v_e";

(c) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, +2, 2v_e";

(d) "0v_a, +1, 4v_b, +1, 5v_d, -1, +2, 4v_c, -2, +1, +2, 2v_e".

It can now be noted that all of these trees have the same BOCF string encoding which supports that they are variations of the same unordered tree. This encoding will provide great benefit to unordered tree mining methods where the counting or matching of the same trees is required. In the existing algorithms [96, 97], the expensive process of finding a representative canonical form for mapping the isomorphic unordered trees can be avoided if the BOCF string encoding is used. BOCF string encoding provides an improved unordered tree representation in comparison to their preorder traversal based canonical forms (e.g., BFCF and DFCF) because it is not only memory efficient but it also allows avoiding the expensive sorting process for choosing a representative canonical form.

3.4 CANONICAL FORMS FOR LABELLED FREE TREES

Generally, defining canonical form for free tree is more challenging than the rooted unordered trees. The main challenge is that there could be more possible ways to represent a free tree than that of a rooted tree because of having no defined root node and no direction among sibling nodes. Therefore the chance of having isomorphic trees in a database of free trees is very high. This necessitates of having a systemic approach for representing a free tree. A proper representation can ensure accurate indexing for further processing and knowledge discovery. In frequent pattern mining algorithms, defining a canonical form for free trees is required to identify the common patterns among free trees. This thesis proposes an efficient canonical form for free trees by extending the above mentioned BOCF for unordered trees to represent free trees.

3.4.1 Balanced Optimal Canonical Form of Free Trees

If the root node of a free tree can be uniquely defined, then the balanced optimal search order can be used to define its canonical form. In this thesis, the canonical form for free tree is defined by following a two-step process [59]. These steps are:

– Normalisation

– Canonical String Encoding

First, a free tree is normalised into the rooted unordered tree by fixing a root node and then the canonical form as well as the canonical string is defined. For normalising a free tree, first all of its leaf nodes along with their incident edges are removed at a time until a single node or two adjacent nodes are left. The free tree with a single remaining node is called a *central tree* and, with a pair of remaining nodes is called a *bicentral tree* [96]. In a *central tree*, the remaining single node becomes the root of the free tree. In a *bicentral tree*, the node with minimum lexicographically ordered label is chosen as the root node. After the normalisation step, a free tree is converted to a rooted unordered tree, and the BOCF for the rooted unordered tree can now be used to encode it.

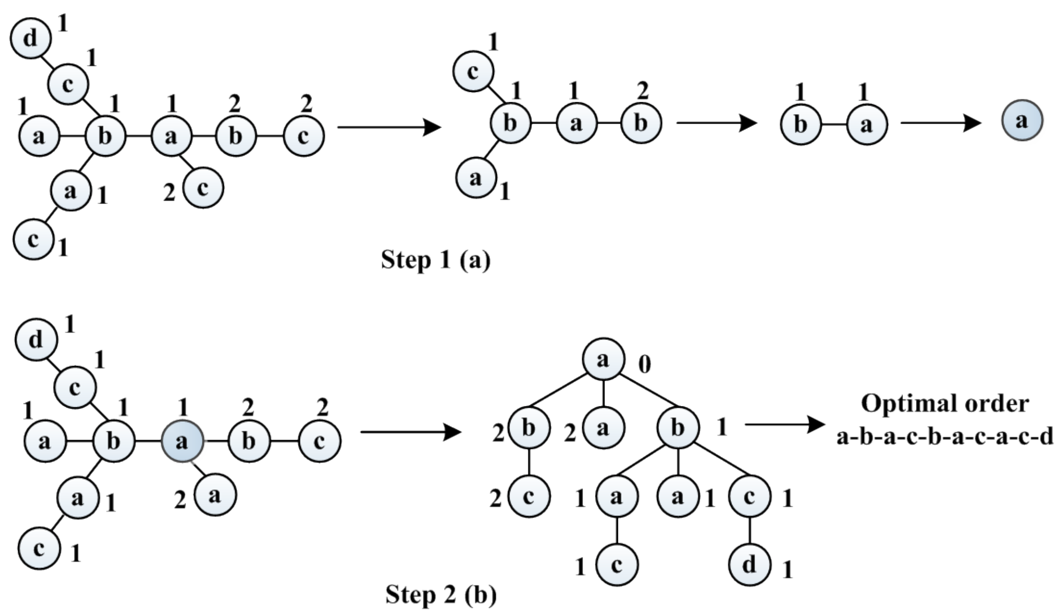


Figure 3.5: Process of finding canonical form for a free tree

An Example: Consider the free tree in Figure 3.5(a), during the step of normalisation, the tree is found bicentral for which node v_a is defined as the root node since this node has the minimum lexicographic label. After defining the root node, the BOCF of rooted unordered tree definition is followed to provide the canonical string encoding of this tree as follows:

"0v_a, +1, 2v_b, +2, 2v_a, -2, +1, 2v_c, -1, +2, +2, 1v_b, +1, 1v_a, +2, 1v_c, +2, 1v_a, -2, -2, +1, 1v_c, -1, +2, +1, 1v_d".

All of these proposed canonical forms have been implemented in the proposed corresponding frequent subtree mining algorithms. In Chapter 5, the algorithm details are provided with the results of empirical analysis, which proves the efficiency of these canonical forms by showing the superior performance over the state-of-the-art algorithms, even in the presence of isomorphism.

3.5 OTHER DATA STRUCTURES

During data operation, the choice of data structures becomes an important factor. For example, in the frequent subtree mining algorithm, both the candidate generation and frequency counting steps require a data operation that should be space efficient with fast access, since the efficiency of the frequent subtree mining algorithm is measured by how well the candidate generation and frequency counting steps are performed. Besides the above mentioned representations of trees, two more supporting data structures are introduced in this thesis that will help in fast execution of the proposed frequent subtree mining algorithms [57-59]. One of them is a dictionary structure that works as a look-up structure to reduce the local subtrees that are generated during the candidate generation process into the integer hyperlink form [148]. Another one is an occurrence list that allows efficient frequency counting by reducing search space. This discussion of effective data structure will help the reader to understand the mechanical details of the proposed frequent subtree mining algorithms in Chapter 5.

3.5.1 Dictionary

The dictionary is a horizontal representation of tree data that captures the inherent hierarchical relationships in it. This structure has been used in various frequent subtree mining algorithms [98, 148]. In a similar manner, a dictionary is presented as a global structure where an array object is used to store the information. Therefore, accessing any information out of this structure ensures less memory expense and can be performed in $O(1)$ time. In the dictionary, the index of each cell refers to the position of each node in the original tree following the BOS traversing order and each cell stores the information such as *label*, *level, fan-out, weight* and a *link* to the pre-order position of the parent node (for the root node, it is equal to -1). Thus, each cell in the dictionary will contain a tuple of {*label*, *level*, *fan-out*, *weight*, *link*} (as shown in Figure 3.6).

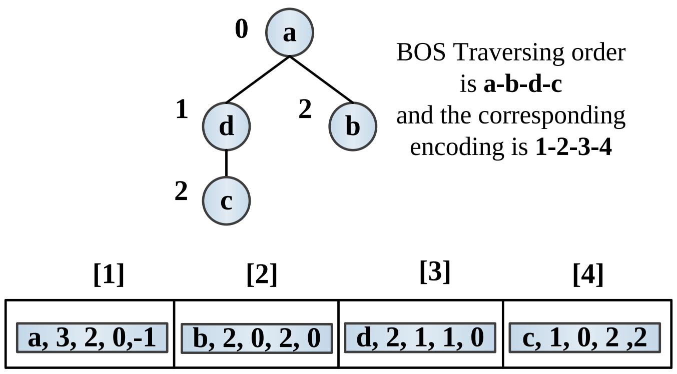

0
a
1
d
2
b
2
c
BOS Traversing order is **a-b-d-c** and the corresponding encoding is **1-2-3-4**
[1]
[2]
[3]
[4]
a, 3, 2, 0,-1
b, 2, 0, 2, 0
d, 2, 1, 1, 0
c, 1, 0, 2 ,2

Figure 3.6: An illustration of dictionary generation for a tree where each cell in the dictionary has a tuple as: {*label*, *level*, *fan-out*, *weight*, *link*}

It can be determined from observing the cells in this dictionary that a node is a leaf node if its fan-out is equal to zero and a node is a root node if its weight is 0. The level information can be utilised to determine whether a node is a descendant node or child node by checking level difference. The level information encodes the hierarchical notion of tree structures.

3.5.2 Occurrence List

To ease the frequency counting step of frequent subtree mining algorithms, a vertical data structure based on the concept called *occurrence list* [90, 96, 98] is utilised in this thesis.

The Occurrence List (OL) based vertical structure for a rooted unordered tree can be described as a list of each occurrences of that tree in the database. Later by simply calculating the size of OL vertically (column wise), the frequent subtree can be identified, since the frequency count of each subtree is equal to the OL size. The main advantage of using the OL is that the frequency count does not need to be updated separately in addition to inserting the occurrence in OL, which is needed anyway for the candidate generation process, and the size of OL can be determined at almost no cost.

The OL of tree t_v represented in its BOCF can be considered in a form as (ID; v_1; …; v_k) where ids of the transactions containing t_v in the database are indicating using ID and v_1; …; v_k indicate the mapping between the indices of nodes in t_v and those in the transaction. Whether t_v is frequent can be checked using its occurrence list, because the total number of elements in OL with distinct ID will be same as the support of t_v.

3.6 CHAPTER SUMMARY

The focus of this chapter is to concisely present the proposed representations and effective data structures which are part of the proposed tree mining algorithms discussed in later chapters. This chapter first introduced novel data representations based on optimal tree traversal that address the limitations of the state-of-the-art representations, which are all order oriented. A brief discussion is added on effective data structures that have been used in the proposed frequent subtree mining algorithms to ease the overall data processing cost and speed. An insight into these tree representations and data structure is essential to understand the interaction between these components and the proposed algorithms.

Since several tree representations are available in the literature, Chapter 2 presented the rationale behind proposing the new tree representations. The existing representations lack the capacity of dealing with the problem of isomorphism and automorphism associated with the rooted unordered and free trees. To address this problem, a novel tree traversal algorithm is proposed that provides a unique traversing order for the isomorphic unordered trees. Two adjacency matrices are introduced, which offer better portrayal of structural relations existing in rooted unordered trees than the traditional adjacency matrix. Two canonical representations are proposed that can effectively handle the isomorphism problem in unordered and free trees representations. All of these representations contribute greatly in the proposed tree matching and frequent subtree mining algorithms that are discussed in Chapter 4 and 5 respectively. These representations can be considered as backbone and a reason for improved performance of these algorithms.

In the last section of this chapter, the dynamic aspects of a tree representation are covered, which include the data structures responsible for effective processing of the designed algorithms. These data structures are mainly adopted to implement the frequent subtree mining algorithms, since these kinds of algorithms are very expensive to execute. The concepts of dictionary and occurrence list are introduced here, which alleviate the effort of the candidate generation and frequency counting steps in the proposed frequent subtree algorithms.

Chapter 4: Tree Matching

Chapter 4 focuses on tree matching – an important contribution of this thesis. A tree matching algorithm is proposed for measuring similarity between unordered tree pairs. This algorithm yields significantly less computational complexity than the traditional tree edit distance-based methods. Instead of using edit string operation, this algorithm adopts a matrix comparison approach using a novel equivalent matrix representation for trees. The first and second papers utilise the novel Augmented Adjacency Matrix (AAM) for tree matching, whereas the third paper utilises the novel Extended Augmented Adjacency Matrix (EAAM) representation.

This chapter is organised based on three papers that introduce the proposed tree matching algorithm. It follows the sequence of Papers 1 and 2 that describe the novel balance optimal search (BOS) traversal algorithm with technical detail and experiments. They also include the AAM-based tree similarity measure algorithm. Paper 1 is a published conference article whereas Paper 2 is a comprehensive under-review journal article. In the journal paper, the proposed tree matching algorithm is extended to do clustering. By using the similarity information, the trees can be clustered. Paper 3 introduces the tree matching algorithm with the EAAM representation that utilises the frequent subtree information for measuring similarity. It also shows an implementation of a clustering algorithm using the driven similarity information from the proposed tree matching algorithm.

Each paper is presented in its original form; a brief overview of each method is provided along with some of materials that were excluded from the papers due to space restrictions enforced by the publishers. Following on from this introduction a brief description about the clustering process is provided, which aims to give an insight into how the existing clustering algorithms can benefit from the knowledge discovered from other algorithms, such as frequent subtree mining and tree matching. In this thesis, clustering is only shown as a real life application of the proposed tree matching algorithms.

4.1 AN OVERVIEW OF THE CLUSTERING PROCESS

Tree data can be clustered using the pairwise similarity information derived by a tree matching algorithm. The tree matching algorithm finds the similarity information between trees that can be used in grouping them. The frequent substructure mining algorithm finds the commonality among the database of trees in the form of frequent subtrees that can be used in clustering trees [161, 162]. Figure 4.1 presents a generic framework for clustering, which helps to understand how tree representation, tree matching, frequent subtree mining and clustering can be integrated in the same framework. After representing trees in a suitable format, tree matching and frequent subtree mining algorithms can be implemented. Clustering is an unsupervised data mining task that does grouping of the data based on their similarity, which can be derived through a tree matching or frequent subtree mining algorithm. Tree matching can be carried out using the frequent pattern information and for finding a frequent subtree, tree matching can also be used. In the literature, it is shown that based on the frequently occurred subtrees, rules can be derived to calculate similarities between trees [50] and, tree matching algorithms like tree edit distance can also be used to restrict the candidate generation step in the task of mining frequent embedded subtrees [36]. Hence, these methods have some measures of interdependency.

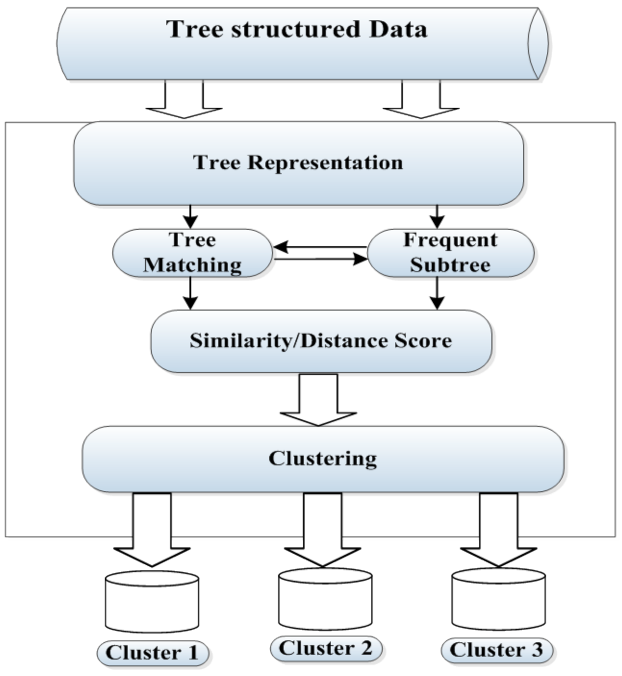


Figure 4.1: A generic tree data clustering framework

The success of a tree clustering algorithm largely depends on the representation and the similarity measure steps. Consequently, clustering can be seen as the end product or application of a similarity measure method. By evaluating clustering performance, the performances of tree matching and subtree mining algorithms can be evaluated.

Many established algorithms for tree clustering are available in the literature [12, 163]. Among them, one of the most widely used is partitional clustering. Any similarity matrix that is derived from a tree matching algorithm can be fed to a partitional clustering algorithm, i.e., k-way clustering [164] for getting the clustering results. The level-wise similarity-based clustering is also reported [72, 137]. A lot of clustering algorithms have used the matrix similarity [23, 77, 165]. Some clustering algorithms are proposed using frequent substructure extraction [162, 166]. So, a conclusion can be drawn by saying that the improvisation of tree matching and a frequent subtree mining algorithm can guarantee a better clustering output. The thesis objective is not to compare and critique the existing clustering methods. This discussion is only added as the clustering process is carried out in Papers 2 and 3 using the finding from the proposed tree matching algorithms, and thus the readers are given an overview of the available applications of the proposed methods.

4.2 A NOVEL METHOD FOR FINDING SIMILARITIES BETWEEN UNORDERED TREES USING MATRIX DATA MODEL

This paper contains the preliminary study results of the AAM representation-based pair wise tree matching algorithm for unordered trees. The initial concept of balanced optimal search traversal is introduced in this paper to explain the idea of augmented adjacency matrix construction. The promising results of this algorithm have been generated based on two real life data sets, Bill of Material (BOM) [23] and glycan data [80]. Both of these data can naturally be depicted as a rooted unordered tree.

4.3 MEASURING SIMILARITY BETWEEN UNORDERED TREES WITH THE BALANCED-OPTIMAL-SEARCH TRAVERSAL ALGORITHM

The promising results from Paper 1 encouraged the authors to conduct deep study on the BOS traversal algorithm and on the AAM representation. This paper details the mathematical modelling of BOS traversal as well as the performance evaluation in comparison to other traversal approaches. Important properties of BOS traversal and AAM representations are described in detail. Extensive experiments with more data sets have been reported to prove the efficiency of the proposed tree matching algorithm. Experimental study shows that the introduced tree matching algorithm ensures less computational expense in comparison to the recent tree edit distance based algorithms. The AAM representation ensures better accuracy performance in comparison to the traditional adjacency matrix. All of these results are obtained by using real life datasets. Further, this algorithm is extended to do clustering to show an application of the tree matching algorithm.

4.4 IDENTIFYING PRODUCT FAMILIES USING DATA MINING TECHNIQUES IN MANUFACTURING PARADIGM

This paper introduces the tree matching algorithm using the EAAM representation. The AAM representation and, hence the tree matching algorithm, only consider the tree specific information for quantizing similarity between trees. It would be interesting to check whether incorporating database specific information can provide an advantage in similarity measures. For obtaining initial insight of a database, frequent subtree information is found helpful [49]. Therefore, in EAAM representation of trees, a new weight based on the frequently occurred parent-child relations for the considered database is added and trees in EAAM forms are compared. The result of this similarity measure algorithm has been found useful in clustering trees.

Since the idea of this work is to use database-specific knowledge in finding similarities, therefore the whole contribution is presented, focusing on a particular domain data. The bill of material, which can be depicted as a rooted unordered tree, is used for conducting the experiments. BOM is an important domain data in the manufacturing paradigm and finding similarity between BOMs is essential in various applications. One of them is to accelerate the product design and planning for launching a new product in the market. However, the proposed method can be implemented in any domain as long as the domain data can be modelled as rooted unordered trees.

NB: The reader may be found the published paper a bit different than the version of the paper added in this thesis. This is done to correct some confusing wordings, which does not change any core concept of the work.

Paper 1: A Novel Method for Finding Similarities between Unordered Trees Using Matrix Data Model

Israt Jahan Chowdhury* and Richi Nayak*****

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

PUBLISHED IN: Lecture Note in Computer Science: Web Information Systems Engineering – WISE 2013, 8180, pp. 421-430

Abstract[2]**:** Trees are capable of portraying the semi-structured data which is common in web domain. Finding similarities between trees is mandatory for several applications that deal with semi-structured data. Existing similarity methods examine a pair of trees by comparing through nodes and paths of two trees, and find the similarity between them. However, these methods provide unfavourable results for unordered tree data and the tree matching problem has found NP-hard or MAX-SNP hard. In this paper, we present a novel method that encodes a tree with an optimal traversing approach first, and then, utilises it to model the tree with its equivalent matrix representation for finding similarity between unordered trees efficiently. Empirical analysis shows that the proposed method is able to achieve high accuracy even on the large data sets.

Keywords: Semi-structured Data, Unordered Tree, Similarity Measure, Matrix Representation.

1. INTRODUCTION

The Web domain consists of heterogeneous data in various forms such as HTML, XML, image, videos and text. Some of these data are naturally represented as tree data structures. Comparing the tree-structured data is important as it enable searching for interesting information among the abundant data efficiently. Many researchers confirm the significance of unordered tree data representation and their comparisons [46, 109]. An unordered tree does not have left-to-right fixed order among siblings node and only preserves the ancestor-descendant or parent-child relationship. Especially in the Web domain where the data source is heterogeneous, the unordered tree representation gives more freedom for flexible matching and concise representation.

A large number of tree mining methods have been developed for finding similarities [42]. Majority of them are for ordered trees and very few are available for unordered trees due to the complexities involved with the unordered tree processing. Existing similarity methods examine a pair of trees by comparing through nodes and paths of two trees, and aggregate the similarity between them [167]. Some similarity

[2] X. Lin et al. (Eds.): WISE 2013, Part 1, LNAI 8180, pp. 421–430, 2013.
© Springer-Verlag Berlin Heidelberg 2013

measure methods use tree level information by considering their common nodes in the corresponding levels and giving different weight in different levels, but it fails to reserve the child-parent relationship among tree nodes [51]. Higher order models such as Tensor Space Model (TSM) have also been used for representing tree data and finding similarities, though these techniques suffer from high dimensionality as well as complexity problems [134]. Tree edit distance methods are also commonly used in measuring similarity between the tree data. These methods measure the distance between two trees in terms of minimum cost to transform one tree into another tree by applying edit operations such as deletion, insertion and substitution [42]. The edit distance computing algorithms for ordered tree data are known to exhibit $O(n^3)$ complexity, where n is the maximum number of nodes in two input trees [114]. The tree-edit distance problem for unordered trees is NP-hard [45, 47]. A few methods have been developed by reducing the tree edit distance problem to the maximum clique problem [40, 124] or proposing variants of the tree edit distance problem [129]. However, they still suffer from high complexity for large unordered tree structure [40]. Other examples of unordered tree matching methods are tree pattern matching [102], maximum agreement subtree [168], largest common subtree [131], and smallest common supertree. These methods also suffer from the complexity problem. In summary, existing methods provide unfavorable results for unordered tree data and result in yielding high complexity.

We propose a novel idea of representing the trees with matrix data structure using tree encoding, and then comparing two matrix structures using efficient cosine similarity measure. An optimal traversing adapting a well-known optimisation problem called “Simple Assembly Line Balancing” is used to provide tree encoding for unordered tree data. A matrix based representation called “Augmented Adjacency Matrix” is proposed to represent the tree data based on the encoding information. The empirical analysis shows that the proposed method performs well with high accuracy and outperforms benchmarking methods for the large size data. The proposed method is able to achieve $O(n^2)$ complexity due to its incorporation of matrix data for comparison. This is remarkable as the existing similarity methods for unordered trees mostly give intractable solutions through exhibiting high computational complexity [45, 47].

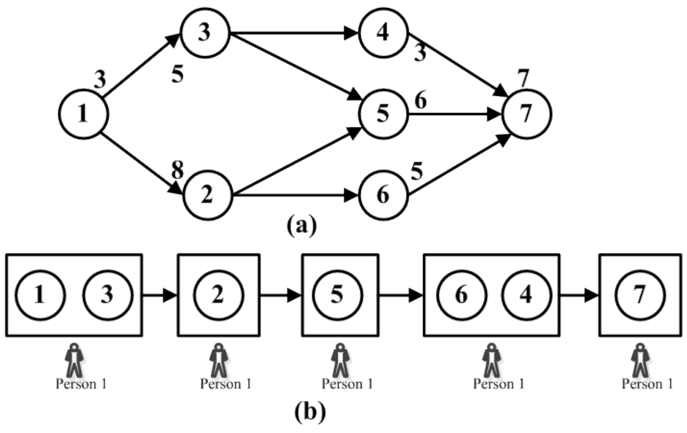


Figure 1: The simple assembly line balancing problem, first diagram replicates an assembly line (a), second one representing optimal sequence of operations on various machines (b)

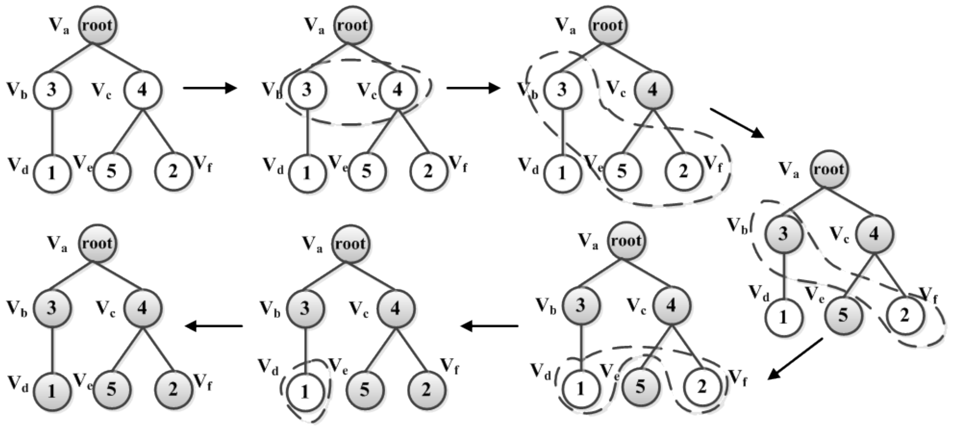


Figure 2: Optimal tree traversal

2. THE PROPOSED SIMILARITY MEASURE METHOD

The proposed unordered tree similarity method includes three steps. Firstly, the tree data is encoded with an optimal traversing approach. Secondly, an equivalent matrix representation is obtained for each tree structure utilizing the tree encoding with other tree information. Thirdly, cosine similarity measure is used to calculate the similarity between two matrices representing unordered trees.

2.1 Step 1: Tree encoding using an optimal traversal approach

Tree Traversal: A tree traversal is a systematic approach of visiting each node once in a tree by following certain strategy and returns a list containing the node sequence traversed along the way. The depth-first search (DFS) and breadth-first search (BFS) are two commonly used traversing algorithms that rely on the fixed ordering among sibling nodes. A DFS algorithm starts from root node and explores each branch as far as possible before backtracking. They can be classified as pre-order, in-order and post-order, based on the sequence of visiting nodes on right or left order. A BFS algorithm, also known as level order traversal algorithm, starts visiting a tree from its root node and then follows a strategy for traversing other nodes in the order of their level from left to right [83]. These strategies are able to represent ordered trees efficiently; however, they face challenges when applied for unordered tree traversal as there is no fixed order among sibling nodes. To our best knowledge, these are the only two strategies that have been used for representing and canonisation of unordered trees [90].

Optimal Tree Traversal: In this paper we introduce an optimal tree traversal method for representing unordered tree. This method is inspired by a well-known optimisation problem known as "Simple Assembly Line Balancing" from the "Operation Research" paradigm [65]. In manufacturing, the line balancing problem is used to minimise the cost of production by balancing the machine sequences of an assembly line based on their operating time and finds the optimal sequence that will support minimum operation or cycle time. Figure 1(a) illustrates a scenario where the nodes are representing various machines in an assembly line and the numerical values outside the nodes stand for the operation time requiring for each machine (Figure 1(b)) shows the optimal sequence of completion tasks according to the assembly line problem. In the proposed method we metaphor the assembly line as the unordered tree; a machine as a tree node; and the optimal sequence as the optimal tree traversal. The weight of a node is calculated by counting the number of occurrences of each node under its parent node. The traversal process begins at the root node. The children nodes are visited only after their parent nodes are visited. This is done to ensure that the ancestral ordering constraint is preserved. The objective of the traversal approach is to minimise the overall traversal time and return an optimal node sequence for the unordered tree.

Problem Definition: Let tree $T = (V, E)$ be an unordered labeled tree where $V = (v_0, v_1, \ldots, v_n)$ denotes the set of nodes that presumes a partial order ρ due to the ancestral relation (i.e., $i \, \rho \, j \rightarrow i > j$ where i and j are node indices and i is ancestor of j). If function tr: $T \rightarrow T^*$ that passes over the tree, listing all nodes that met along the way, then it is called tree traversal. T^* is n-dimensional vector, representing the list of nodes in the order of traversal according to the specified traversal strategy, $(v_0, v_\mathrm{i}, \ldots, v_n) = V \in T^*$, where, v_0 is the root node. By using the working principle of line balancing problem, we define the general traversal function to an optimisation problem for achieving the optimal node traversal sequence. Let the set of nodes $V = (v_0, v_1, \ldots, v_n)$, traversed in a sequence by using the line balancing principle, be called the optimal tree traversal if the traversal function tr does not violate the ancestry relationship given by the unordered tree and ensures minimum computational cost as well as traversal time.

Tree Encoding: After receiving the optimal sequence for traversing all tree nodes, each node will be encoded according to its order in this sequence. For instance, in Figure 2, the traversal will start from the root node V_a and the optimal sequence is v_a-v_c-v_e-v_b-v_f-v_d. The encoded values for the nodes in the tree will be 1-2-3-4-5-6 for v_a-v_c-v_e-v_b-v_f-v_d respectively.

2.2 Step 2: Tree Modeling with the Augmented Adjacency Matrix Representation

Adjacency matrix has been used for representing trees and graphs by modeling the adjacency information regarding parent-child relationship [88]. Let the adjacency matrix A model the tree T (V, E) as followings.

$$A_{ij} = \begin{cases} \text{true}, & v_i, v_j \in E(T) \\ \text{false}, & \text{otherwise} \end{cases} \tag{1}$$

A tree data is a hierarchical representation that includes the inherent implicit relationships and semantics of various nodes. The traditional adjacency matrix fails to represent the label information, level information, encoding information, and ancestry relationships. To overcome these limitations the following Augmented Adjacency matrix is proposed to model tree data more accurately.

Augmented Adjacency Matrix: This is a square matrix that utilises the level, encoding and weight information of a tree to represent the cell values.

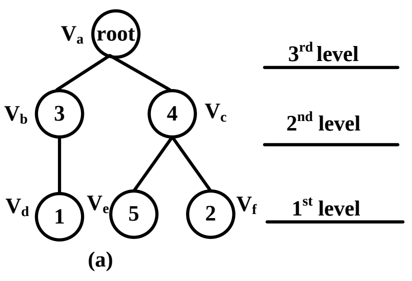
Encoding Information: By using the optimal traversing sequence, we obtain the encoding values of the tree nodes according to the order they are visited. The root node becomes the first row and column to be represented in the matrix and the other nodes are arranged in the optimal order achieved by the optimal traversal. This encoding value also integrates with the level value between two nodes.

Level Information: The level information in a tree represents the ancestry relationships of the nodes. This structural information is important for finding similarity between trees [51]. The nodes appearing high on tree carry more influence than nodes appearing near the leaf nodes. Consequently, the level assignment is bottom-up; the lowest leaf node is assigned the level 1 and the higher value is assigned to the root node level. The following rules are applied to assign a value to two nodes, v_i and v_j, incorporating the level information.

1. If an ancestor-descendant relationship exists between two nodes v_i and v_j, where v_i is the ancestor of v_j, or if the encoding value of v_i is less than the encoding value of v_j then the level value of cell C_{ij} is: $\frac{level(v_j)}{level(v_i)}$. The function *level* outputs the level value of a node.
2. If an ancestral relationship does not exists between two nodes v_i and v_j, or if the encoding value of v_i is greater than the encoding value of v_j then the level value for cell C_{ij} will be 0.

Weight Information: In this method, nodes carry a weight displaying how frequently the node occurs under its parent node. The node weight is added to the corresponding level value. Additionally, a value of 1 is added to each diagonal cell of the adjacency matrix to represent the existence of corresponding node on that tree.

We illustrate the process of modelling the tree with the augmented adjacency matrix and populating the matrix values. Figure 3 illustrates the traditional adjacency matrix and the augmented adjacency matrix for a given tree. The example tree has three levels, and the root node level is considered as the highest one. The encoding value of nodes is received from Figure 3 by using the optimal traversal. The traversal sequence is v_a-v_c-v_e-v_b-v_f-v_d and the encoding values for these nodes are 1-2-3-4-5-6 respectively. The level information of corresponding nodes is calculated, and the node weights are added to the level values. For instance, consider the calculation of the cell value, $C_{23,}$ showing the relation between v_a-v_c. The encoding value of $v_a = 1$ which is less than the encoding value of $v_c = 2$ that means v_a is ancestor of v_c. According to rule 1, the level value of C_{23} is $\frac{level(V_j)}{level(V_i)} = \frac{2}{3}$. The weight of v_c is 4. The final cell value will be 2/3+4. The rest of the cell values are being calculated in the same way.



	$V_a(1)$	$V_c(2)$	$V_e(3)$	$V_b(4)$	$V_f(5)$	$V_d(6)$
$V_a(1)$	0	1	0	1	0	0
$V_c(2)$	0	0	1	0	1	0
$V_e(3)$	0	0	0	0	0	0
$V_b(4)$	0	0	0	0	0	1
$V_f(5)$	0	0	0	0	0	0
$V_d(6)$	0	0	0	0	0	0

Traditional Adjacency Matrix (b)

	$V_a(1)$	$V_c(2)$	$V_e(3)$	$V_b(4)$	$V_f(5)$	$V_d(6)$
$V_a(1)$	1	2/3+4	1/3	2/3+3	1/3	1/3
$V_c(2)$	0	1	1/2+5	0	1/2+2	0
$V_e(3)$	0	0	1	0	0	0
$V_b(4)$	0	0	0	1	0	1/2+1
$V_f(5)$	0	0	0	0	1	0
$V_d(6)$	0	0	0	0	0	1

Augmented Adjacency Matrix (c)

Figure 3: Augmented adjacency matrix

2.3 Step 3: Measuring Similarity

Let A' and B' represent Augmented Adjacency Matrices of the corresponding trees. If the two trees differ in size, extra columns and rows with zero elements are added to the smaller matrix for making the size of both matrixes equal. A matrix can be considered as a $n \times n$ dimensional vector. The value of each cell of a matrix is a dimension of the vector, starting from the first row to the end row; the $n \times n$ dimensional vector is represented. Similarity between two matrices can be calculated by using cosine similarity. Table 1 illustrates the similarity process.

It is expected to achieve a polynomial time complexity with the proposed method detailed in Table 1. The method consists of three steps. The complexity of the first step is $O(n^2)$, same as the line balancing optimisation problem. The complexity of the second step is known to be $O(n^2)$ for modelling the adjacency matrix based on tree encoding information. The final step comprises cosine similarity calculation, too small to count in; consequently it can be ignored during complexity analysis. The overall complexity is $O(n^2)$ where n is the maximum number of nodes in the input trees pair.

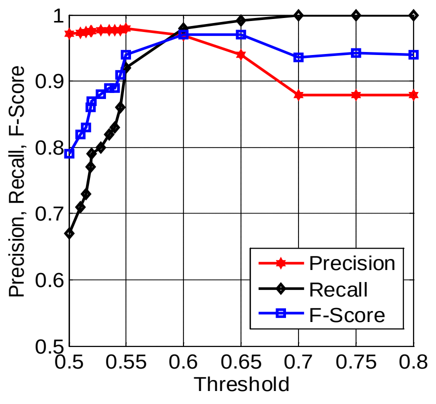
```
Algorithm : Measuring Similarity

Input: Unordered trees $T_a$ and $T_b$
Output: Measurement similarity between tree pair

1. Model the tree $T_a$ with the Augmented Adjacency Matrix $A'$;
2. Model the tree $T_b$ with the Augmented Adjacency Matrix $B'$;
3.    if $|B'|>|A'|$ then
          Add ($|B'| - |A'|$) rows and columns of zeros at the right end and bottom of the
          matrix $A'$;
      else
          Add ($|A'| - |B'|$) rows and columns of zeros at the right end and bottom of the
          matrix $B'$;
      end if
4. Calculate similarity between two trees using
   $Cos(A',B') = \frac{\sum_{x=1}^{n}\sum_{y=1}^{n} A'_{xy} B'_{xy}}{\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} {A'_{xy}}^2}\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} {B'_{xy}}^2}}$
```

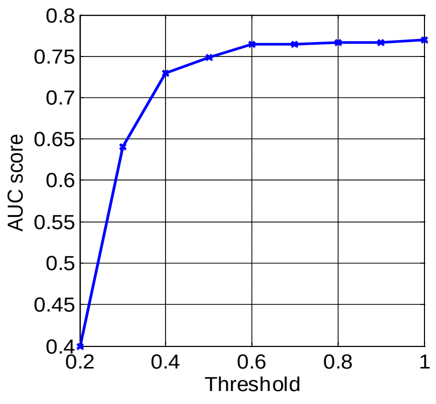
Table 1: The proposed similarity measure algorithm

3. EXPERIMENTAL RESULTS

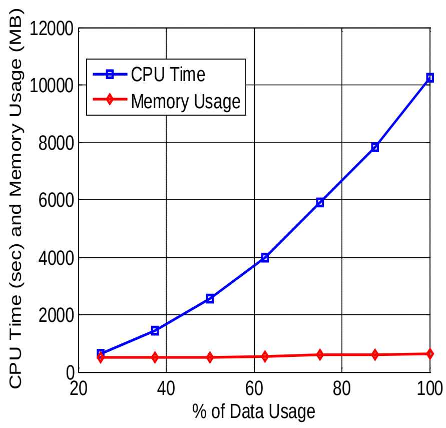
The proposed similarity measure method is evaluated on two datasets including the Bill of Material (BOM) data that has the similar structure as XML documents [23] and the Glycan structures obtained from the KEGG/Glycan database [80]. The proposed method is implemented on Matlab and experiments are performed on a PC with RAM size 8.00 GB and a processor Intel Core i7.

Performance on the BOM Data: The BOM data set consists of 404 sample BOMs with 50,000 nodes and 12,000 unique nodes. The dataset includes trees with maximum and minimum depth of 8 and 4 respectively, whereas the maximum and minimum breadth is 10 and 6 respectively. The well-known evaluation metrics such as precision, recall, F-score and AUC are calculated. To calculate these measures, positive and negative samples were needed. For this purpose, a tree pair in the data set is regarded as positive if the distance score is smaller than a given threshold. Otherwise it is regarded as negative. The threshold value is determined empirically. Figure 4(a) and (b) show the performance of the matrices with varied threshold values. As expected, data in Figure 4(a) shows that with the increase in threshold, matching accuracy is improved yielding the best matches showing increase in precision; however it reduces the number of matches resulting the fall in recall. Considering the trade-off between precision and recall, the proposed method produces the best result when the threshold is set in the range between 0.6~0.65 (Figure 4(a)). For thresholds below the value of 0.3, AUC score is less than 0.5, indicating the random classification (Figure 4(b)). The threshold value that is higher than 0.5 gives a good quality solution yielding higher AUC.





(a) (b)



(c)

Figure 4: Evaluation metrics with varied thresholds (a, b) and scalability test (c)

Total # nodes	Clique Edit	UwClique Edit	DpClique Edit-A	DpClique Edit-B	DpClique Edit-C	DpClique Edit-D	DpClique Edit-E	**Proposed Method**
55~59	1.987	0.433	8.968	0.108	**0.088**	0.086	0.096	0.374
60~64	2.746	4.949	1.78	0.167	0.163	**0.149**	0.177	0.47
65~69	64.29	9.303	39.46	0.381	0.364	**0.328**	0.357	1.513
70~74	58.69	**0.099**	1.337	0.545	0.436	0.463	0.501	1.517
75~79	2.441	0.918	4.051	0.953	**0.752**	0.754	0.781	1.547
80~84	7.150	6.570	44.63	2.516	2.268	1.620	1.653	**1.55**
85~89	237.7	28.03	21.11	3.205	3.205	2.413	2.490	**1.641**
90~94	303.2	1211	1710	38.81	26.30	8.165	9.475	**1.761**
Average	84.78	157.66	228.92	5.84	1.75	1.75	1.94	**1.29**

Table 2: Average CPU time (sec) per glycan pair is shown for each case. Bold text indicates the best results for each case and the highlighted cell indicates the worst results for each case

We performed a scalability test by varying the BOM data set of different size reporting the CPU time and memory usage. Figure 4(c) reveals that the method is able to provide the $O(n^2)$ complexity, confirming the theoretical complexity analysis. The memory usage does not change with the increased data size, as the proposed method just needs to keep a pair of trees in the memory at a time.

Performance on the Glycan Structures: We used the Glycan data for comparing scalability of the proposed method with the state-of-the-art similarity measure methods such as CliqueEdit, UwCliqueEdit, and DpCliqueEdit [124]. It is to be noted that none of these available methods empirically analysis their accuracy. They conduct the CPU time analysis to show the complexity. We compare our proposed method based on CPU time with these methods. For analysis, tree pairs are selected randomly from the data set with a specified range of the total number of nodes (i.e., sum of the numbers of nodes in two trees) and the average CPU time per pair is measured.

Results in Table 2 show that our proposed method performs well for almost all sizes of trees. Although the proposed method does not give best result for the smaller tree node sizes, between the ranges of 55~59 and 75~79, but several other methods perform worse than our method. After reaching the range 80~84, our method outperforms others due to the use of optimal traversal. Overall the average performance of all subsets of datasets (the last row) indicates that our method outperforms all methods, some with very large margin. The CliqueEdit, UwCliqueEdit, and DpCliqueEdit [124] methods implement several heuristics to cut the CPU expense, but provides no results about accuracy of the matching process. We provide the accuracy test for our proposed method on the BOM dataset. Results ascertain that the proposed method is able to achieve high accuracy and polynomial complexity.

4. CONCLUSION

The unordered tree data represents information inherent in many domains naturally. This presses the need of developing an efficient method of measuring similarity between trees especially when we are living in the big data era. This paper proposes an efficient method of measuring similarity between unordered trees. The proposed method introduces an augmented adjacency matrix structure for modeling the tree data. The matrix representation enables efficient computation of pair of trees for finding similarity. An optimal traversal of the tree is obtained using a line of balance optimisation problem. The encoding values of the nodes with this optimal traversal are utilised in representing the tree with the matrix structure.

Empirical analysis shows that the proposed method is able to achieve improved complexity in comparison to existing methods even for large datasets. Results also showed that an improved complexity is achieved with high accuracy. The proposed method is able to achieve polynomial complexity whereas the existing methods for calculating similarity amongst unordered trees suffer from the high computational complexity.

Our future plan is to work on the detail of the optimal traversal approach to improve the overall performance. We plan to apply heuristics to improve the scalability further. We also plan to do more experiments to analyze effectiveness and versatility of the proposed method.

Paper 2: Measuring Similarity between Unordered Trees with the Balanced-Optimal-Search Traversal Algorithm

Israt Jahan Chowdhury* and Richi Nayak*****

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

UNDER REVIEW IN: Journal of Knowledge and Information Systems

Abstract: Calculating similarity between trees is an elementary task in many applications. Tree edit distance is a commonly used method for performing this task. However, for unordered trees, this problem is known to be intractable, i.e., NP-hard and MAX SNP-hard. Apparently, the challenges in such manipulation come from the complex mapping inherent in unordered tree structures. This paper introduces an encoding scheme for unordered trees using a novel tree traversal algorithm that is proposed by reducing the traversal problem to a simple assembly line balancing problem - a well-known optimisation problem in the operations research paradigm. By minimising traversing cost, this algorithm achieves an optimal traversal path of an unordered tree and allows a new encoding embedded matrix representation of the unordered tree data. We propose a similarity measure based on this representation. Empirical analysis shows that the proposed method requires significantly less computational time than the baseline methods, without compromising the accuracy of output.

Keywords: Unordered tree, Optimisation, Tree traversal, Matrix representation, Similarity measure.

1. INTRODUCTION

Due to the unique capability of portraying topological and relational characteristics, the dominance of tree structured data presentation can be seen in a diverse range of real-life applications. Typical examples of tree structured data are XML data and weblogs in web intelligence; DNA and glycan data in bioinformatics; bill of material (BOM) documents in manufacturing; phylogenetic trees in evolutionary science and many others [33, 168, 169]. Tree matching is fundamental to the core operation of many data manipulation tasks such as clustering analysis, nearest-neighbour classification, data integration, data cleansing and data querying [60, 81].

Much research in this area concentrates on the ordered type of trees (i.e., trees in which the left-to-right order among siblings is fixed). However, important problems in the research fields of genetics, bioinformatics and web intelligence emphasise the need for developing efficient methods of manipulating unordered trees. For example, (i) in genealogical studies, various genetic diseases need to be diagnosed based upon the pattern of ancestry trees that are unordered; (ii) in bioinformatics representing glycan structures as unordered trees will ease the way of knowledge mining [21]; (iii) in the manufacturing industry, BOM documents can be depicted as rooted unordered trees [23]; (iv) in evolutionary science, the unordered tree data is used for finding the set of species that have a common ancestor for modelling their evolution [113]; and (v) in the web domain, the generated semi-structured data is mostly represented as unordered trees in order to capture the common patterns and irregularities [29].

The structure of a tree plays an important role in differentiating the data. The dependencies inherent in a structure need to be captured efficiently for data manipulation [46]. The expression of hierarchical dependencies in unordered trees is different (i.e., less constrained) from the ordered trees, which means there is a demand for an efficient data representation for capturing them. Apparently the lack of efficient equivalent representation raises the complexity in tree mapping as well as increasing the computational complexity of executing tree manipulation algorithms. A variety of methods based on nodes, paths, number of cliques, and subtree representations have been proposed to solve the unordered tree matching problem [42, 102, 167]. However, the majority of these methods have shown this problem to be NP-complete, even MAX SNP-hard, which means unless P = NP there is no polynomial time approximation scheme for this problem [45, 47]. Because of the high complexity yielded by existing methods, measuring similarities of unordered trees is still an open problem.

To address this challenge, we developed a new representation based similarity measure method for unordered trees. Using the optimisation theory, we developed a novel tree traversal algorithm called Balanced-Optimal-Search (BOS) that encodes unordered trees by ensuring an optimal traversing order. The idea is to reduce the traversing problem to the Simple Assembly Line Balancing (SALB) problem - a well-studied optimisation problem in operations research [65, 157]. An optimisation model is formulated for solving the traversing problem, which consists of feasibility constraints and an objective function for minimising the computation time of traversal. The solution of this formulation leads towards an optimal traversal order as well as an efficient encoding of the unordered tree. An approximate numerical matrix representation called Augmented Adjacency Matrix (AAM) is then presented by embedding this encoding along with other tree structural information for the tree data. Finally, we modified the vector cosine similarity metric to make it compatible with matrix computation for calculating similarity between a tree pair.

The proposed method is evaluated using several real life datasets and benchmarked against several recent methods [79, 124] for finding similarities between unordered trees. Empirical analysis shows that the proposed method significantly reduces the computation time, even for datasets that include large trees. The proposed method gives only $O(n^2)$ complexity, which is an achievement, as the existing methods show the problem of finding similarity between unordered trees as intractable. In this paper, an application of our proposed similarity measure to clustering is also presented with the accuracy analysis.

Summarising, the contributions of our paper are as follows:

1. Introduced a novel tree traversal algorithm BOS by reducing traversal problem to the SALB optimisation problem. By minimising computation cost of traversing, BOS gives an optimal traversing sequence without relying on a fixed left-to-right order among siblings, unlike existing traversal algorithms.
2. Developed a method with polynomial complexity that is comprised of a new matrix representation AAM and uses a modified cosine similarity metric for quick matrix pair comparison.

The rest of the paper is organised as follows. In Section 2 we provide the back-ground and an overview of the related work. Section 3 presents the proposed traversal algorithm BOS with its complexity analysis. Section 4 details the proposed method for finding similarity. We report experimental results and a clustering application of this similarity measure in Section 5. Finally, we conclude our work in Section 6.

2. BACKGROUND AND RELATED WORK

Unless otherwise stated, all trees we consider in the paper are rooted labelled and unordered.

2.1 Preliminary and Notations

A rooted labelled unordered tree has a unique root node and preserves the ancestor-descendant or parent-child relationships among nodes. All nodes are labelled in the tree. Unlike ordered trees, there is no fixed left-to-right order or any other order among siblings. Let $T = (V, E, L)$ be a *rooted labeled unordered tree*, where $V(T)$, $E(T)$, $L(T)$ denote the set of nodes, edges and node labels (In this thesis, we do not consider the edge label) that are constructed as

$V(T) = \{v_0, v_1, v_2, \ldots, v_{|T|}\}$, $v_0 = root$, $|T|$ = Tree size (Total of tree nodes),
$E(T) = \{(v_i, v_j) \mid v_i, v_j \in V\} = \{e_1, e_2, e_3, \ldots, e_{|T|}\}$,
$L(T) = \{lab_0, lab_1, lab_2, \ldots, lab_{|T|} \mid \Phi\colon V \rightarrow L\}$, Φ = mapping function.

Each node v_i has a unique path from its position to root v_0. The *parent* of v_i (and $v_i \neq v_0$), denoted as Pv_i, is the adjacent node of v_i in that unique path to v_0. The *ancestors* of v_i, denoted as Av_i, are all the other nodes in that unique path except v_i itself. The *children* of v_i are the immediate follower nodes of v_i, the number of the children is also known as *fan-out*, denoted by f_i. The *descendants* of v_i are the list of all follower nodes of v_i, denoted as Dv_i.

The *ancestral constraint* ($const_{anc}$) poses a partial order ρ among the nodes of an unordered tree. The '$\prec$' symbol represents 'precedes', e.g., if v_i is ancestor of v_j then this relation is denoted by $v_i \prec v_j$. It is defined as:

$const_{anc} = \{v_i \,\rho\, v_j \text{ iff } v_i \prec v_j, v_i \in Av_j, v_j \in Dv_i\}$

A distinctive fundamental property between ordered and unordered trees is the sibling constraint that can be presented as

$const_{sib}^{ordered} = \{v_j \,\tau\, v_k \ncong v_k \,\tau\, v_j \text{ iff } Pv_j = v_i = Pv_k, v_j \neq v_k\}$
$const_{sib}^{unordered} = \{v_j \,\tau\, v_k \cong v_k \,\tau\, v_j \text{ iff } Pv_j = v_i = Pv_k, v_j \neq v_k\}$

where τ denotes an order between two sibling nodes (i.e. left-to-right). We can assume that changing the position of sibling nodes v_j and v_k of an unordered tree from left to right will not change any fundamental structure of that tree. In this paper, n denotes the number of nodes in a larger input tree, $n = \max\{|T_1|, |T_2|\}$ where T_1 and T_2 are input trees.

2.2 Unordered Tree Matching

Although the focus of this paper is not arguing against the ordered tree matching, this section highlights the superiority of using unordered tree matching in data manipulation. A substantial difference between ordered and unordered tree matching is that the order of sibling nodes can be exchanged in unordered trees, and trees with those nodes can still be considered matched. This flexibility makes unordered tree matching advantageous from various aspects. In the era of big data, the existence of diverse data sources is increasing, and analysing inconsistent and overlapping data becomes challenging. Unordered tree matching can provide accurate insight of data even in the presence of inconsistency or irregularity. Let us consider the following examples.

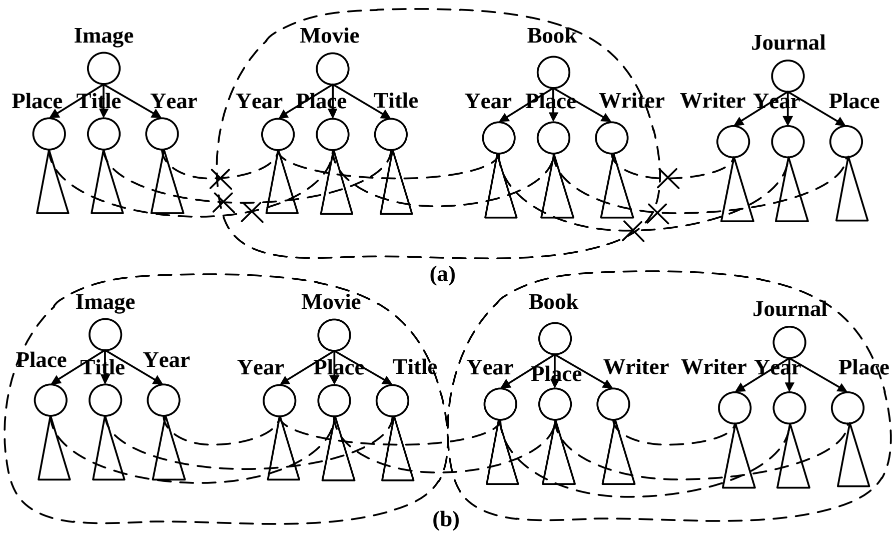


Figure 1: Possible mappings considering ordered trees (a) and considering unordered trees (b)

In a query system when searching for an element person with the sub elements first name and last name (possibly with specific values), ordered matching would give less relevance to the cases in which the order of these nodes, first name and last name, is reversed. However, in reality, changing the order of first and last names usually does not make any difference. The way to solve this problem is to consider the query subtree as unordered, in which only the ancestral constraint is preserved and the sibling order is ignored. The query can be posed and answered without being concerned about the sibling order.

In a heterogeneous domain comparison between documents that are part of different sources is challenging. The documents may portray the same information, but, in different structural order. Consider Figure 1 that shows a fragment of heterogeneous data that contain four tree structures. Considering these trees as ordered gives the possible mapping among nodes based on structural similarity as shown in Figure 1(a) that derives higher similarity between “Movie” and “Book” due to the sibling constraint. The matching between nodes is done by keeping the order of sibling nodes under the root node in consideration. In reality, “Journal” and “Book” should have higher similarity and so does “Image” and “Movie”. In this case, only unordered representation (as shown in Figure 1(b)) allows necessary mapping and results in accurate and flexible matching as desired.

2.3 Unordered Tree Representation

Similarity computation is known as the dual problem of distance computation, hence these two terms (i.e. similarity and distance) have often been used interchangeably [60]. Trees are complex in structure and any kind of manipulations using the tree structure format is a non-trivial task [11]. To enable efficient computation, trees are often represented as vector or matrix forms. In the vector or matrix representation of trees, the nodes are encoded with a traversal algorithm. Tree traversing is a systematic approach of visiting each node in the tree only once. This process returns a list containing the node sequence traversed along the way as the output. Traversal approaches adopting the breadth-first-search (BFS) and depth-first-search (DFS) have been used extensively for encoding both ordered and unordered trees [62].

Figure 2 shows an example of two trees (Figure 2(a) & Figure 2(c)) of identical properties except the varied order between sibling nodes (dotted rectangles) and their DFS and BFS encodings accordingly. It is clearly visible that both DFS and BFS traversals visit the sibling nodes by preserving an order from left-to-right, which supports the properties of being an ordered tree. However, for an unordered tree these two schemes encode two similar unordered trees (only varied by sibling order) differently, which may result in calculating a false distance measure. The example in Figure 2 shows the distinct encodings for tree 1 (Figure 2(a) & Figure 2(b)) and tree 2 (Figure 2(c) & Figure 2(d)) provided by both DFS and BFS traversals, which is desirable for ordered tree representation. For unordered tree representation the encoding should be the same, but the DFS and BFS traversals encode them differently, which creates the need for developing an alternative unordered tree encoding scheme without relying on left-to-right sibling order.

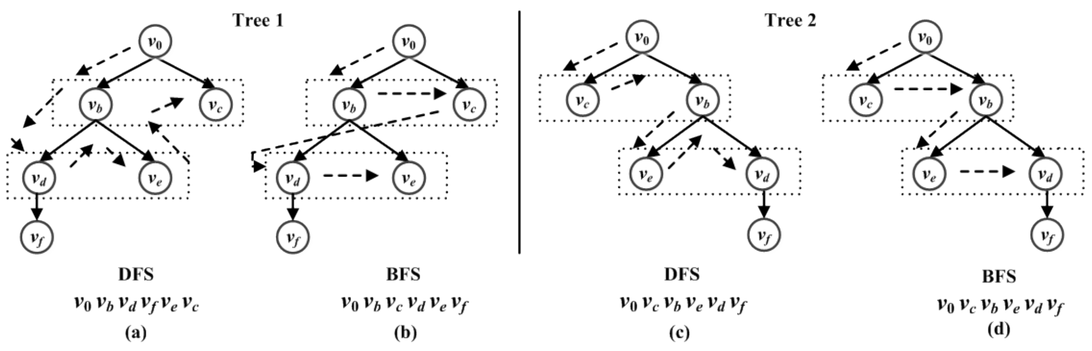

Tree 1
Tree 2
DFS
$v_0\, v_b\, v_d\, v_f\, v_e\, v_c$
(a)
BFS
$v_0\, v_b\, v_c\, v_d\, v_e\, v_f$
(b)
DFS
$v_0\, v_c\, v_b\, v_e\, v_d\, v_f$
(c)
BFS
$v_0\, v_c\, v_b\, v_e\, v_d\, v_f$
(d)

Figure 2: DFS & BFS traversal (dotted arrows indicate the traversing direction)

2.4 Related Works

A myriad of tree mining methods have been developed for finding similarities between tree pairs. The majority of them are applicable for ordered trees, and very few are available for unordered trees due to the complexities involved with unordered tree processing. Among various similarity methods, the most commonly used method is tree edit distance [42]. It measures the distance between two trees by the minimum cost required to transform one tree into another through several edit operations such as deletion, insertion and substitution. The complexity of edit distance problem for ordered tree data is $O(n^3)$, whereas the edit distance problem for unordered tree is NP-hard [48]. Furthermore, the problem was shown as MAX SNP-hard [45, 47].

To overcome the complexity problem, researchers have developed algorithms constrained to conditions such as tree size and other tree properties; however they result in compromising on accuracy. Akutsu et al [112] introduced an algorithm under fixed parameters, which exhibited improved complexity of $O(2.62^k.poly(n))$, however, it performs poorly when comparing non-similar trees. A few methods have been developed by reducing the tree edit distance problem to a clique problem [108]. For example, Fukagawa et al [40] proposed a method of computing maximum clique in which an instance of tree edit distance is directly transformed into an instance of the maximum vertex weighted clique problem, and then it is solved using a clique solver [170]. This method can work efficiently on moderate sized trees, but it will be slow for the large sized trees. This method is further improved with using dynamic programming that repeatedly solves instances of the maximum vertex weighted clique problem as subproblems [124]. However, this method still suffers from high complexity for large tree structures. Some similar reductions [128, 129] and methods of variants of the tree edit distance problem [107] have been proposed, however none of them exactly solves the formal tree edit distance problem for unordered trees.

Apart from tree edit distance, other examples of unordered tree matching methods are tree pattern matching [102], maximum agreement subtree [168], smallest common supertree and largest common subtree [131]. Most of these methods suffer from high complexity problem. An efficient method for computing tree similarity has been proposed using tree level information by counting the common nodes in the corresponding levels of two trees and giving different weights for different levels, but this fails to preserve the child-parent relationship among tree nodes [72]. Higher order models such as the Tensor Space Model (TSM) have also been used for representing tree data and finding similarities, though these techniques suffer from a high dimensionality problem [134].

In summary, existing methods provide unfavourable results for unordered tree data and result in yielding high computational complexity. Different from these methods, we propose a novel optimisation based traversal technique that allows an efficient and equivalent matrix representation of the tree. To our best knowledge this is the only method that uses optimisation for representing unordered tree data in order to calculate similarity. The optimisation technique allows us to achieve the polynomial time complexity and the representation from tree to matrix facilitates fast computation.

3. THE BALANCED-OPTIMAL-SEARCH (BOS) ALGORITHM

We reduce the tree traversal problem to the optimisation problem inherited from the "Operations Research" paradigm called simple assembly line balancing (SALB) [65, 157], and propose a new order independent traversal algorithm. SALB is a combinatorial optimisation problem that chooses an optimal path for a network by avoiding exhaustive searching. In literature, SALB has been used to solve networks in manufacturing problems that are represented by a predecessor digraph, i.e., a graph holding all properties of unordered trees. We conjecture that SALB can propose an optimal path for an unordered tree-like network. The tree traversal problem can be treated as an assignment problem as approached in SALB. This motivates us to reduce the unordered tree traversal problem to the SALB problem.

3.1 Simple Assembly Line Balancing (SALB) Problem

We first define the preliminaries of SALB. In the manufacturing domain, an assembly line is a sequence of p linearly ordered stations that are linked by a conveyor belt. A station performs the same set of tasks repeatedly during each cycle of the assembly line. The set of tasks J, processed by p stations within one cycle time c, is fixed. The time required to complete a task is termed as the process time t. The sum of the process time of all tasks assigned to a station is called the work content of that station. Since a cycle time is c, the set of tasks is available to a station for only c time units. Therefore, the work content of a station should not exceed c in order to let the line operate smoothly with no delays. The tasks cannot be assigned to the station arbitrarily because of the sequencing requirement. These factors, called as precedence relation, define a partial order on the set of tasks.

The objective of SALB is to find an optimal balance of the assembly line in such a way that the total slack (i.e. the sum of the idle times of all the stations along the line) is minimum. For a fixed cycle time, this can be attained by minimising the number of stations. If the tasks can be grouped such that all the work contents are exactly equal, the line will have a perfect balance. The aim of the SALB optimal model becomes finding the minimum number of stations that can complete a sequence of tasks with the minimum delay. The solution yields the optimal sequence of these stations.

Let a predecessor digraph $G = (J, A)$ with nodes J and edges A define a partially ordered set of tasks $J = \{ j_1, j_2, \ldots, j_z\}$. If the set of tasks J are assigned to station S_k, $k \in \{1, 2, \ldots, p\}$, where $p \leq z$ then the SALB problem can be defined as follows:

Definition 1 (*The SALB Problem*) Assignment b_r: the set of tasks $J = \{j_1, j_2, \ldots, j_u, j_w, \ldots, j_z\}$ $(1 \leq u < w \leq p)$ to p ordered set of stations $\{S_1, S_2, \ldots, S_p\}$ is balanced, if the following conditions are held.

1. Assignment b_r does not violate the partial order given by predecessor digraph $G = (J, A)$, i.e., inclusion $(a, b) \in A$ implies that task j_a is assigned to a station S_k and task j_b is assigned to S_l such that, $1 \leq k \leq l \leq p$.

2. Cycle time c is not violated for station S_k, i.e., sum of the processing time of all the tasks assigned to a station should be within cycle time c.
3. Assignment b_r assigns all tasks to a minimum possible number of stations p for the fixed cycle time c.

3.2 Reduction of the Tree Traversal Problem to SALB

Traversing a tree involves iterating over all nodes in the tree following a traversal strategy. We first give the basic definition of the tree traversal problem and then define the Balanced-Optimal-Search (BOS) traversal approach and its associated properties.

Definition 2 (*Tree Traversal*) Tree traversal is a function tr: $T \rightarrow T^*$ that iterates over the tree, listing all nodes that are met along the way. T^* is a n-dimensional vector, representing the list of nodes in the order of traversal according to the traversal strategy, $(v_0, v_1, \ldots, v_{|T|}) = V \in T^*$. Let $I = (i_0, i_1, \ldots, i_{|T|})$ be the set of iterations required to traverse a tree. Under each iteration a tree node will be traversed.

Definition 3 (*BOS Traversal*) BOS traversal is an order independent traversal that adopts optimisation as a strategy for traversing all nodes of a tree.

Definition 4 (*Equivalent Nodes*): Two nodes v_i and v_j are called equivalent nodes, denoted by $v_i \cong v_j$; if they have the same label ($lab_i = lab_j$ & $lab_i, lab_j \in L$), they are originated from the same labelled parent node ($Pv_i = Pv_j$), and they have the same labelled child nodes.

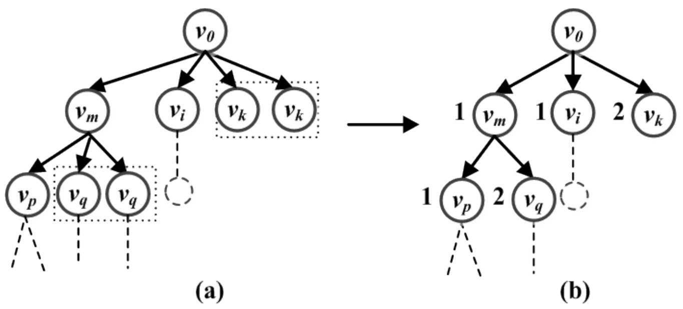


Figure 3: A fraction of an unordered tree (a) in which the dotted rectangles show the equivalent nodes (assuming that node v_q has the same labelled child node) (b) Using weighted nodes, a condensed tree representation

Definition 5 (*Weight of a Node*): Weight of a node v_i ($v_i \neq v_0$), is defined as the total number of its equivalent nodes.

A condensed representation of tree is possible by applying the Definition 4 and 5. For instance, Figure 3(b) shows the condensed tree with weights obtained by collapsing equivalent nodes.

According to the properties of unordered trees we have lemma 1.

Lemma 1: *Weight of the root node v_0 is always zero, $w_0 = 0$. For each node $v_i \in V$ ($v_i \neq v_0$), the weight w_i ($w_i \neq w_0$) should always have a minimum value of one.*

PROOF:

1. According to tree structure schema, no equivalent nodes of a root node are possible as the parent and ancestors are undefined for a root node. The weight of a root will always be zero (In the rest of the paper for displaying tree structure the zero root weight is omitted).
2. Each node v_i ($v_i \neq v_0$) of a tree T should have at least one equivalent node otherwise v_i does not exist in the tree. Hence the minimum weight of the node is one, $w_i = 1$. For node v_i, $w_i > 1$ if the node has more than one equivalent nodes.

Definition 6 (*Candidate Node*): Node v_i is called the candidate node if all of its ancestor nodes have been traversed and it is yet to be traversed. A candidate node is considered eligible for traversing in the next iteration. There can be multiple candidate nodes available for traversing in the next iteration. The set of candidate nodes is denoted by $V_{can} = \{v_i, \ldots, v_j\}$ where $\{Pv_i, \ldots, Pv_j\}$ are labelled as traversed. The weight and fan-out for a set of candidate nodes are denoted as w_{can} and f_{can}.

***Mapping*:** Using the metaphor of assembly line for an unordered tree, we explain the mapping process. A tree node can be considered as a task in the assembly line. The node weight is equivalent to the processing task time. The rationale is, as the weight can generate from the accumulation of several equivalent nodes, often a node (i.e., with weight greater than one) is not just a single node but rather, a multiple number of similar nodes. Therefore, we treat weight as an equivalent term of processing time, which allows us to assume that a higher weighted node will require more time to traverse than a lower weighted node. An iteration involved in traversing a tree node can be considered a station for a task completion. The objective of the SALB optimal model for tree traversal is visiting all nodes of the tree within the minimum possible traversing time. Each iteration involved in traversal yields a different execution time, due to the variations in sorting and storing time of nodes, i.e., there exists a different candidate node set at each iteration. Some nodes must be deferred after applying the heuristics and stored for later visits. The overall traversal time is equivalent to the total iteration time. Minimising the total iteration time will minimise the total traversal time.

Lemma 2: *A BOS traversal neither violates the ancestral constraint nor allows the sibling constraint during traversing an unordered tree. If $v_i \prec v_j$ then node v_i is traversed in iteration i_a and node v_j is traversed in iteration i_b, such that* $1 \leq a \leq b \leq |T|$.

PROOF: In the SALB problem, the precedence relationship is strictly followed for a task assignment (i.e. before assigning a task its immediate predecessor task must be processed); therefore a BOS traversal also follows the ancestral constraint by confirming that a node is visited only after traversing its parent node. On the other hand the sibling constraint can be proved by saying that BOS uses optimisation for completing traversal where no left-to-right order among sibling nodes is kept, which means this traversal process does not keep the sibling constraint for encoding an unordered tree.

Lemma 3: *For each iteration of BOS traversal, the upper bound of computation time is the maximum value of node weight for the tree.*

PROOF: It can be proved by using the second condition of the SALB definition. For reducing the traversal problem to SALB, weight of a node is regarded as the traversal time of the corresponding node. Therefore, the node weight under each iteration should be within the maximum weight of the considered tree.

Lemma 4: *BOS traversal ensures the complete enumeration i.e. all nodes will be visited for traversing a tree.*

PROOF: The SALB problem is aimed at processing all tasks along the assembly line. For reduction, we consider tree nodes as tasks. Hence the BOS traversal aims to visit all nodes of a tree and ensures the complete enumeration.

3.3 The Optimisation Model Formulation

This section details the optimisation modelling of the BOS traversal. For simplicity of modelling, only i is used for representing the i-th node denoted as v_i, likewise for iterations. x_{ij} is introduced as the decision variables for this mathematical model; x_{ij} is 1 if node i is traversed at iteration j, otherwise it is 0. Let c_j be the traversing time needed to complete iteration j. Based on these variables, the mathematical model is formulated as follows:

$$Minimize\ z = \sum_{i \in V_{can}} \sum_{j=1}^{|T|} c_j x_{ij} \qquad (1)$$

$$Subject\ to\ \sum_{j=1}^{|T|} x_{ij} = 1 \qquad \forall i \in V \qquad (2)$$

$$\sum_{i=1}^{|T|} w_i\, x_{ij} \le w_{max} \qquad \forall j \in I \qquad (3)$$

$$x_{ik} \le \sum_{j=1}^{k} x_{hi} \qquad k \in I\ and\ \forall i \in V\ and\ \ \forall h \in Av_i \qquad (4)$$

$$x_{ij}\ in\ \{0,1\} \qquad \forall i \in V\ and\ \forall j \in I \qquad (5)$$

Eq (1) represents the objective function and Eqs (2) to (5) represent constraints of the model. Eq (2), the occurrence constraint, guarantees that each node is assigned to an iteration, consequently, all the nodes are traversed in the whole process. Eq (3), the upper bound constraint, guarantees that the time required for an iteration is at most the pre-specified upper bound time. Eq (4), the ancestral constraint, preserves the topological sequencing, which is required for a tree to be an unordered tree. If x_{ik} = 1, i.e., if node i is traversed under iteration k, then the RHS of (4) must assume a positive value (for each ancestor h of i), i.e., each ancestor h is traversed before traversing of node i starts. If on the other hand, x_{ik} = 0, i.e., if node i is not traversed under iteration k, then the RHS of (4) may not be positive, i.e., ancestor h of i may or may not be traversed at the first k iteration. Eq (5), the non-divisibility constraint, guarantees that each variable assumes only a value of 0 or 1, thus a node cannot be split among two or more iterations.

Solving this model by standard optimisation techniques for finding an optimal solution is not a realistic choice as it will yield high complexity like NP hard [65, 171]. To cope with this complexity, we propose to apply two heuristics based on priority rules to restrict the enumeration process of the tree, which guarantees an approximate solution closer to true optimum. Consequently, it gives only polynomial time complexity. These heuristics are applied in the order given. Heuristic-1 assists in identifying a potential node during the traversal process. However, if many potential nodes are found, heuristic-2 is applied to breaking the tie among equivalent nodes.

Heuristic-1 Using this heuristic, the node for traversing next can be prioritised from the set of candidate nodes. We explain the steps and definitions necessary for the application of heuristic 1 as follows.

Candidate Node Generation: At the beginning of the traversing algorithm, the whole tree is scanned for finding the upper bound of an iteration time, denoted by *UB*. The maximum possible value of *UB* is the highest weight of the tree nodes. This is considered as the initial limit of *UB*. For each next iteration, the value of *UB* is updated by subtracting the weight of the recent traversed node from the initial value of *UB*. If the updated value of *UB* is found negative or less than a candidate node weight, the initial limit is set as *UB* of that iteration. The set of candidate nodes is formed by the following:

Node $v_i \rightarrow V_{can}$ *iff* $v_i \in V$ & $v_i \neq v_0$ & v_i *is not labelled as traversed* & Pv_i *is labelled as traversed. Now, any node* $v_i \in V_{can}$ *will be deferred for traversing in the next iteration iff* $w_i \geq UB$.

Prioritising a Candidate Node: After sorting the candidate nodes, the next node for traversing will be prioritised based on the following condition:

For[3] $V_{can} = \{v_i, v_j, \ldots\}$ with $\{w_i, w_j, \ldots\} \leq UB$, v_i *will be selected for traversing iff* $\int_{\max} \{w_i, w_j, \ldots\} \rightarrow w_i$

[3] *In this thesis,* $\int_{\max}$ and $\int_{\min}$ *respectively are referring the maximum and minimum value of a set.*

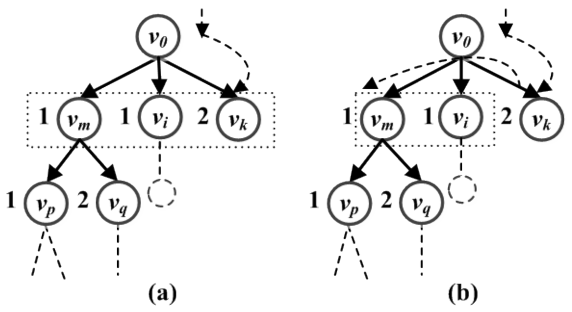


Figure 4: An example of implementing heuristics 1 and 2

In the case of multiple potential nodes, heuristic 2 is applied to rank the traversing order.

Heuristic-2 *For V_{can} = {v_i, v_j, …}* where *{w_i, w_j, …} ≤ UB, if the number of maximum weighted nodes > 1 then the traversing order will be ranked based on the nodes with the largest number of child nodes or highest fan-out. Two nodes {v_i, v_j}∈ V_{can} where $w_i = w_j$ then v_i will be chosen for traversed iff $f_i > f_j$.*

Heuristic-3 *For V_{can} = {v_i, v_j, …}* with *{w_i, w_j, …} ≤ UB, if multiple nodes exist with maximum weight and children count, the minimum lexicographically ordered label will be used to prioritise their traversing. Two nodes {v_i, v_j} ∈ V_{can} where $w_i = w_j$ and $f_i = f_j$, then v_i will be chosen for traversal due to having minimum lexicographical label.*

An Example: In Figure 4 we consider the same tree from Figure 3, where numerical values alongside the nodes present their corresponding weights. For the first iteration the upper bound, *UB* is 2. After labelling root node v_0 as traversed, nodes V_{can} = {v_m, v_i, v_k} because their parent node v_0, is labelled as traversed and none of them have been traversed yet (Figure 4(a); dotted rectangle). Moreover, all of their weights are ≤ *UB*. Following heuristic 1, v_k is chosen and traversed next as w_k > {w_m, w_i}. For the next iteration, the updated value of *UB* becomes 0, therefore, the current *UB* is set as the initial value, 2. For this iteration, V_{can} = {v_m, v_i} (Figure 4(b); dotted rectangle). Since $w_m = w_i$, heuristic-2 will be applied and node v_m will be chosen as $f_m > f_i$.

So, the BOS traversal is not keeping any left-to-right order among siblings while traversing a tree, which leads us to claim that the representation issue incurred by the previous DFS and BFS traversals has been resolved

Algorithm Time Complexity: Algorithm 1 (Figure 5) provides the pseudo code of the BOS algorithm. We discuss the complexity of this algorithm using the following lemma.

Algorithm 1: BOS Traversal

Input: Unordered tree, $T(V)$; $V = \{v_0, v_1, v_2, \ldots, v_{|T|}\}$; $w = \{w_1, w_2, \ldots, w_{|T|}\}$

Output: Optimal node traversal sequence return as vector $\mathbb{R}^{|T|} = (r_1, r_2, \ldots, r_{|T|})$

```
1.  $\mathbb{R}^{|T|} \leftarrow \{\ \}$;
2.  $r_1 \leftarrow v_0$;
3.  Label $v_0$ as traversed;
4.  for each $x = 2$ to $|T|$ do
5.      construct $V_{can}$ using definition of candidate node;
6.      Traverse ($V_{can}, w_{can}, f_{can}, UB$) = $r_x$;
7.      Update ($UB, w_{can}$);
8.  end for
9.  return $\mathbb{R}^{|T|}$;
```

Functions

Traverse ($V_{can}, w_{can}, f_{can}, UB$)

```
1.  for all y $\in V_{can}$
2.      if $w_y > UB$
3.          $V_{\text{can}} \leftarrow V_{\text{can}} \setminus v_{\text{y}}$;
4.      end
5.  end
6.  if count($\int_{\max} (\{w_i, w_j, \ldots\} \in w_{can})) = 1$ then
7.      $r \leftarrow v_i \leftarrow$ corresponding node of $\int_{\max} (\{w_i, w_j, \ldots\} \in w_{can}) \rightarrow w_i$;
8.  else
9.      sort (fan-out) $\leftarrow$ corresponding nodes of $\int_{\max} (\{w_i, w_j, \ldots\} \in w_{can})$;
10.     if count($\int_{\max} (\{f_i, f_j, \ldots\} \in f_{can})) = 1$
11.         $r \leftarrow v_i \leftarrow$ corresponding node of $\int_{\max} (\{f_i, f_j, \ldots\} \in f_{can}) \rightarrow f_i$;
12.     else
13.         $r \leftarrow v_i \leftarrow$ corresponding node that has lexicographically minimum label;
14.     end
15. end
16. Label $v_i$ as traversed;
```

```
Update ($UB$, $w_{can}$)
1. $UB \leftarrow UB$ – Weight of the recent traversed node;
2. if $UB = 0$ then
3.     $UB \leftarrow \int_{\max} (w_i, w_j, \ldots w_{|T|})$ ;
4. else $UB < \int_{\min} (\{w_i, w_j, \ldots\} \in w_{can})$
5.     $UB \leftarrow \int_{\max} (w_i, w_j, \ldots w_{|T|})$ ;
6. else
7.     continue
end
```

Figure 5: High level pseudocode of the BOS algorithm

Lemma 5: *The BOS traversing algorithm has time complexity $O(|T|\ log\ |T|)$, where $|T|$ is the number of nodes the tree has.*

PROOF: Implementing any of the heuristics of BOS traversal for sorting nodes will give a possible time complexity of $O(|T|\ log\ |T|)$. Assuming there are $|T_j|$ nodes in iteration j of the tree traversal, it will give $O(|T_j|\ log\ |T_j|)$ complexity to sort these nodes. The total complexity after considering all possible iterations for traversing the whole tree (i.e., all j iteration) is $\sum_j O(|T_j| \log |T_j|)$, which is $O(|T|\ log\ |T|)$.

4. PAIR-WISE SIMILARITY COMPUTATION WITH BOS TRAVERSAL

We propose a method for finding the similarity between unordered tree pairs by using the BOS traversal algorithm. The BOS traversal algorithm provides us the encoding of tree nodes. Using the tree node encoding and incorporating the tree structure information, we introduce an Augmented Adjacency Matrix (AAM) that provides accurate representation of tree structured data. A cosine similarity measure is then used to calculate the similarity between unordered tree pairs represented as AAMs. This method consists of the following steps:

1. Generate the BOS encoding sequence of each tree in the dataset.
2. Construct an equivalent augmented adjacency matrix for each tree.

3. Measure pair-wise similarity using cosine similarity.

4.1 Step1: Encoding the tree nodes using BOS traversal algorithm

Definition 7 (*BOS Encoding*): BOS encoding of a tree T labels the nodes with their position in the BOS traversal sequence order.

Let node v_i be traversed next after root node v_0 in tree T using the BOS traversal. If the position of v_i is changed with its sibling node, it would still be traversed at position 2. The BOS traversal does not necessarily give a sequence by following a left or right order, so the position would follow BOS order rather than its position in the tree. The BOS encoding ensures a distinct identity to a tree node regardless of its position in the tree.

Example Continues: Consider the tree from Figure 4, using BOS traversal the traversing sequence will be v_0-v_k-v_m-v_q-v_p-v_i, thus following definition 7 the encoded values of the nodes, v_0, v_k, v_m, v_q, v_p and v_i become 1, 2, 3, 4, 5 and 6 respectively.

4.2 Step 2: Constructing augmented adjacency matrix

Trees or graph structures have been widely represented as matrices for simplifying computation of tree or graph mining algorithms [62]. Using other representation such as the list of edges or the adjacency list can be cumbersome if there are many edges in a tree. the adjacency matrix is the most commonly used matrix representation of a tree [159].

Definition 8 (*Adjacency Matrix*): For tree $T = (V, E)$, the adjacency matrix $A \in R^{|T| \times |T|} = [a_{ij}]$ is a binary matrix, where $0 < j \leq i \leq |T|$.

$$a_{ij} = \begin{cases} 1 & \text{if an edge exist between } v_i \,\&\, v_j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The adjacency matrix representation of a tree directly depends on the encoding scheme. There can be $|T|!$ different adjacency matrices of a tree, T using different permutations of the set of nodes [160]. Therefore, it is not possible to get a unique adjacency matrix representation for the same unordered tree using any of the DFS and BFS traversal based encoding, as their encodings rely on siblings order. We overcome this shortcoming of adjacency matrix representation by using the BOS encoding. A BOS encoding-driven adjacency matrix will ensure unique identity of a same unordered tree by giving a total order among all adjacency matrices.

A major problem with an adjacency matrix representation is that it only encodes the adjacent links in a tree, whereas, a tree structure contains other information such as implicit relationships (i.e. ancestor-descendent and parent-child), level information, weights and so on. To overcome these limitations, we propose a new matrix representation AAM.

Definition 9 (*Augmented Adjacency Matrix*): Consider tree $T = (V, E)$, where nodes are encoded using the order driven by BOS traversal. The augmented adjacency matrix A' of T, with respect to this ordering of the nodes, is $|T| \times |T|$ matrix where each diagonal entry of 1 is referring the entry of a node and each off-diagonal non-zero entry is referred to as entry of adjacent node or descendent node of the entered node in the corresponding diagonal. The off-diagonal non-zero entry is either level information or the summation of level information and node weight. Since node weight is carrying the quantity information of a node under its parent therefore only those off-diagonal entries include the weight value for which the corresponding nodes are adjacent.

Populating values in AAM: For entry of each node, value 1 is inserted into the diagonally positioned element of AAM, which represents the existence of the corresponding node on that tree. To capture the structural information more accurately, the off-diagonal non-zero values are added in AAM. These values give the information regarding ancestor-descendant and parent-child relationship of the corresponding diagonal node entry. They have two components. The first component, level information, is incorporated to show the ancestor-descendant or parent-child relationships. The second component, weight value, is added to include the number of descendent or child nodes under the ancestor or parent node.

Level information is calculated using the level of each node in a tree. We define two rules for incorporating level information in populating a'_{ij}:

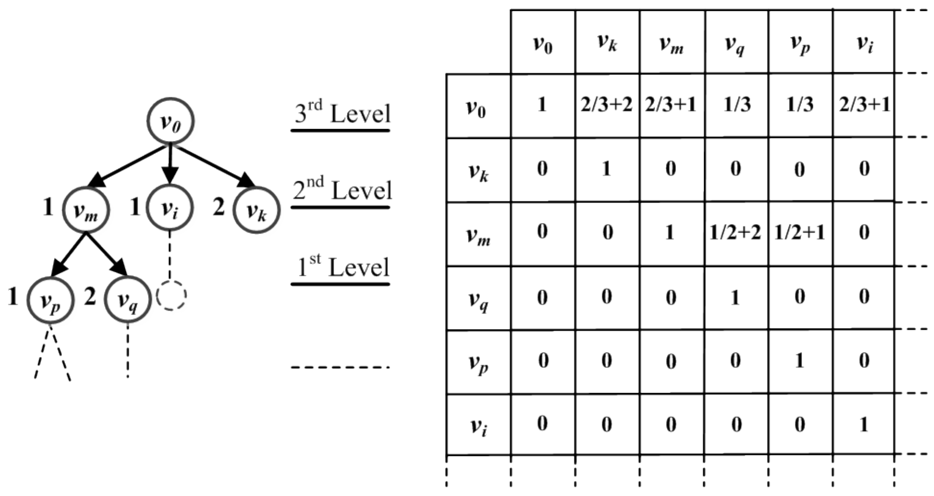
1. If an ancestor-descendant or parent-child relationship exists between two v_i and v_j, the level information of element a'_{ij} of that matrix is calculated as $\frac{Lv(T,\ v_j)}{Lv(T,\ v_i)}$.

2. If no such relationship exists between two nodes v_i and v_j, then the level information of element a'_{ij} is 0.

The $(i, j)^{th}$ entry, a'_{ij} of the augmented adjacency matrix $A' \in R^{|T|\times|T|}$, where $0 < j \le i \le |T|$, can be formulated as:

$$a'_{ij} = \begin{cases} 1 & \textbf{if } v_i \textbf{ is a node of } T \\ \dfrac{Lv(T,v_j)}{Lv(T,v_i)} + w_j & \textbf{if } v_i = Pv_j \\ \dfrac{Lv(T,v_j)}{Lv(T,v_i)} & \textbf{if } v_i \in Av_j \\ 0 & \textbf{otherwise} \end{cases}$$

where $Lv(T, v_i)$ is denoted the level of node v_i in tree T.



	v_0	v_k	v_m	v_q	v_p	v_i
v_0	1	2/3+2	2/3+1	1/3	1/3	2/3+1
v_k	0	1	0	0	0	0
v_m	0	0	1	1/2+2	1/2+1	0
v_q	0	0	0	1	0	0
v_p	0	0	0	0	1	0
v_i	0	0	0	0	0	1

Figure 6: Augmented adjacency matrix (AAM)

Example Continues: Figure 6 presents the equivalent AAM representation of the tree. It shows the level of each node as the way it is considered in constructing AAM. The root node of a tree is positioned at the highest level and rest of the node levels are specified accordingly. Let's calculate the value of $cell_{12}$. An ancestral relation exists between v_0 and v_k. Hence, following the first rule, the level information of $cell_{12}$ is 2/3, where 2 and 3 are the level value of v_0 and v_k respectively. After adding the weight of v_k, the final value of $cell_{12}$ becomes 2/3+ 2.

The AAM presentation provides a unique identity to its equivalent unordered tree, which can be efficiently used in pair-wise similarity computation. The definition and description of AAM give us the following lemma:

Lemma 6: *The AAM is a canonical matrix representation of an unordered tree.*

PROOF: Following the BOS traversing order, tree nodes are encoded and the row and columns of an AAM are arranged. This order is unique for a distinct unordered tree. Whatever permutation is carried out within the nodes of a tree as long as the structure and content of trees are same, the AAM will remain the same. Consequently, this new matrix form can be considered as canonical matrix representation of its corresponding unordered tree.

Algorithm 2: Constructing Augmented Adjacency Matrix

Input: Unordered trees T; $\{v_0, v_1, v_2, \ldots, v_{|T|}\}$, $\{w_0, w_1, w_2, \ldots, w_{|T|}\}$ and $Lv(T, V)$
Output: Augmented Adjacency Matrices A´ of T.

```
1.  Construct initial adjacency matrix $A' \in \mathbb{R}^{|T| \times |T|}$ using the BOS traversing order;
2.  for each $p \in |T|$
3.      for each $q \in |T|$
4.          if $v_p = Pv_q$ then
5.              $a_{pq} \leftarrow \frac{Lv(T, v_q)}{Lv(T, v_p)}$;
6.              $a_{pq} \leftarrow a_{pq} + w_q$;
7.          else if $v_p \in Av_q$ then
8.              $a_{pq} \leftarrow \frac{Lv(T, v_q)}{Lv(T, v_p)}$;
9.          else if $v_p = v_q$ then
10.             $a_{pq} = 1$;
11.         else
12.             $a_{pq} = 0$;
13.         end if
14.     end for
15. end for
```

Figure 7: High level pseudocode of the AAM constructing algorithm

4.3 Step 3: Measuring Similarity

Cosine similarity is computed with two AAMs to find similarity between a tree pair. Cosine similarity is designed to be applied on vectors, whereas AAM is a matrix format, a modification is needed.

Definition 10 (*Cosine Similarity Measure for Matrices*): Let A' and B' be two augmented adjacency matrices of trees T_1 and T_2 respectively. If the sizes of the two trees are not same, additional columns and rows with zero elements are added to the smaller matrix for making the size of both matrices equal. These two square matrices can be considered as two $|T|\times|T|$ (where $|T| = \int_{\max} (T_1, T_2)$) dimensional vectors. The value of each element of a matrix can be seen as a dimension of the vector. Starting from the first row to the end row, a $|T|\times|T|$ dimensional vector is found and the cosine matrix similarity between trees T_1 and T_2, denoted by $SCos\,(T_1, T_2)$ is:

$$SCos\ (T_1, T_2) = \cos(A', B') = \frac{\sum_{x=1}^{n}\sum_{y=1}^{n} A'_{xy}B'_{xy}}{\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} A'^{2}_{xy}}\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} B'^{2}_{xy}}} \tag{6}$$

Algorithm 2 (Figure 7) and Algorithm 3 (Figure 8) show the process of constructing an augmented adjacency matrix and the final similarity score computation using pseudo codes. The complexity of the overall pair-wise similarity measure method can be calculated as follows:

Algorithm 3: Similarity Computation

Input: T_1 and T_2 in form of A' and B'; their AAM representations respectively
Output: Similarity value $S_{Cos}\,(T_1, T_2)$

```
1. if |T1| > | T2| then
2.     Add rows and columns of zero to B' to equalise the size of two matrices;
3. else
4.     Add rows and columns of zero to A' to equalise the size of two matrices;
5. end if
6. Calculate SCos (T1, T2) using equation 6;
```

Figure 8: High level pseudocode of the AAM constructing algorithm

Complexity Analysis: The overall similarity measure calculation results in a polynomial-bounded algorithm. The proposed method consists of three steps: (1) BOS encoding of each tree of a pair; (2) AAM Construction of each tree of a pair; and (3) Similarity calculation. The complexity of generating BOS encoding is exactly the same as BOS traversal, which is $O(|T| \log |T|)$ as detailed in subsection 3.3. For a tree pair (T_1, T_2) the maximum possible complexity for this step will be $O(n \log n)$, where $n = \max \{|T_1|, |T_2|\}$. The complexity of AAM construction is known to be $O(|T|^2)$ based on the adjacency matrix construction complexity, therefore, the overall complexity of constructing AAM for a tree pair in Step 2 will be $O(n^2)$. Again, the complexity of the final step (i.e., similarity computation) is

$O(n^2)$, because of the computing dot product for every pair of row vectors. So, the final complexity of the proposed pair-wise similarity measure method is $O(n^2)$.

5. EXPERIMENTAL RESULTS

In this section we describe the experimental results of our proposed method. Several real life datasets are used for comparing the performance of the proposed method against the relevant baseline methods. The detailed description of the datasets, evaluation criteria, benchmarking methods and the experimental set-up are included below.

5.1 Datasets

We have used two real-life datasets of diverse characteristics (as shown in Table 1) in our experiments. The first data set consists of Bill of Material or BOM documents collected from the manufacturing domain [23]. BOM is a hierarchical portrayal of an end product comprising useful information regarding parts or components, raw materials, quantity and manufacturing process. A BOM document can naturally be depicted as an unordered tree [23]. The second dataset is CSLOGS that consists of Log Markup Language (LOGML), a compact way of structurally expressing the contents of the web log file information using XML [38]. Each user session extracted from the log file is expressed as a tree containing both structure and content information. Both of these datasets are labelled and have been used by researchers [23, 124] in similar experiments.

DB	No of Trees	Total Nodes	Unique Nodes	Max D	Min D	Max B	Min B	Avg D	Avg B
BOM	404	50,000	12,000	8	4	10	6	7	4
CSLOGS	59,691	716,263	13,209	25	3	28	2	10	6

Table 1: Summary of used datasets

5.2 Evaluation Criteria

Precision (P), Recall (R) and FScore (F) are calculated to measure the accuracy of the proposed tree matching algorithm in the following manner:

$$Precision(P) = \frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FP_j} \quad (7)$$

$$Recall(R) = \frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FN_j} \tag{8}$$

$$FScore(F) = \frac{2 \times \frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FP_j} \times \frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FN_j}}{\frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FP_j} + \frac{\sum_{j=1}^{j} TP_j}{\sum_{j=1}^{j} TP_j + FN_j}} \tag{9}$$

Where, *TP*, *FP*, and *FN* denote true positive, false positive and false negative respectively. A tree pair matching in the dataset is regarded as positive if the similarity score is greater than a given similarity threshold, γ; otherwise it is regarded as negative. The value of γ is tuned before accuracy analysis.

A second evaluation metric, NMI is used [172] to evaluate our clustering experimental results alongside FScore.

The computational complexity is checked by computing CPU expense or run time and the sensitivity analysis is conducted by tuning various tree parameters like breadth, depth and size to check the effect in performance.

5.3 Experiment Design and Benchmarks

We have used three sets of experiments. Firstly, we evaluate the proposed method with its variants created by changing one of the three steps included in the method: (1) changing BOS encoding to other preorder traversal (e.g. BFS or DFS) driven encoding; (2) using a different matrix representation (e.g. Adjacency Matrix); and (3) using a different similarity metric (e.g. Dise coefficient, Euclidean, Jaccard coefficient). When changing one component, all other components were kept exactly the same.

Secondly, we compare the proposed method against relevant baseline methods: CliqueEdit [112], UwCliqueEdit [79], and DpCliqueEdit [124]. These methods employ the commonly used similarity measure, tree edit distance [42]. To the best of our knowledge, the considered baselines are recent methods for precisely computing the unordered tree edit distance. CliqueEdit reduces a tree edit distance problem to a maximum vertex weighted clique problem, and an off-the-shelf maximum clique solver was used for getting the solution afterward. Further, in UwCliqueEdit, the maximum vertex weighted clique problem was reduced to maximum clique problem to improve the performance [79]. Later in the DpCliqueEdit method, a dynamic programming approach was combined with the clique-based method [112] and some other heuristics were used to reduce the computation time and labelled as DpCliqueEdit-A, DpCliqueEdit-B, DpCliqueEdit-C and DpCliqueEdit-D [124].

Lastly, we evaluate the performance of the similarity measure through its application in clustering. The similarity matrix illustrating pair-wise comparison between all pairs of trees is determined by the proposed method, and clustering is performed using this matrix. All experiments have been conducted on a 2.8GHz Intel Core i7 PC with 8GB main memory running the windows operating system. All algorithms are implemented in MATLAB R2013b.

5.4 Results: Comparison with Variants

5.4.1 Effect of Encoding Schemes

To have a meaningful comparison between various encoding schemes, we have tuned the similarity threshold, γ for all methods. The results of the first row in Figure 9 validate that over BOM data, all methods achieve stable performance when $\gamma \in [0.7, 1]$ and the last row shows that the stability is achieved when $\gamma \in [0.5, 1]$ over CSLOGS. These different value ranges of γ admit the presence of a high percentage of homogeneous trees in BOM data and trees with large structural difference in CSLOGS data (Table 1).

For comparing various schemes we set the similarity threshold as $\gamma = 0.7$ and $\gamma = 0.6$ for BOM and CSLOGS data respectively. If we check the results from Figure 10, encoding using the BOS traversal ensures better results in every aspect when compared to other traversal based encodings. The BOS traversal achieves higher recall and a fair value of precision. Precision is often referred to as the predictive power of an algorithm, whereas recall assesses the effectiveness of an algorithm on a single class. The results demonstrate that BOS traversal has good predictive power with high efficiency. The other schemes often treat similar unordered trees differently due to considering left-to-right sibling order, therefore the efficiency of these schemes are not as high as BOS.

Both BFS and DFS traversals have $O(n)$ complexity, however the presence of adjacency matrix construction in all of these methods exhibits the same computational complexity, $O(n^2)$. Therefore, the run time comparison is skipped.

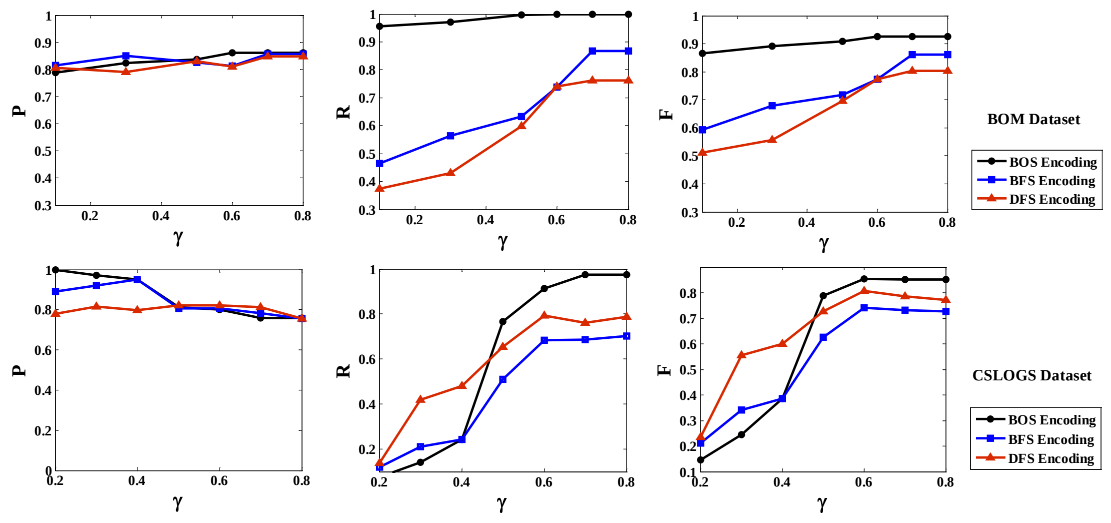


Figure 9: Precision (P), Recall (R) and FScore (F) curves with respect to γ over BOM and CSLOGS data

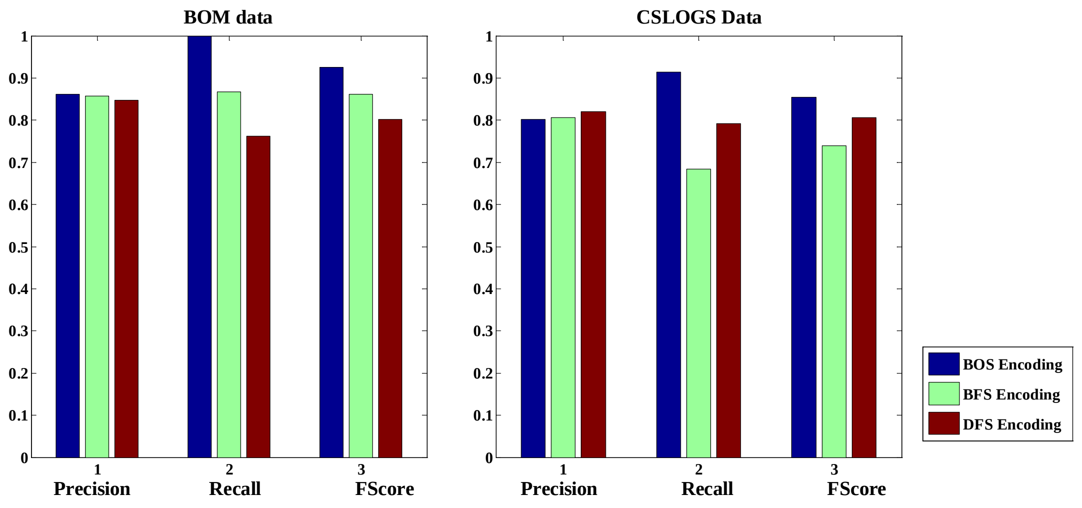


Figure 10: Performance of various traversal encodings over BOM and CSLOGS data

5.4.2 Matrix Representation

To check the effect of AAM in our overall similarity measure, we have considered a variant comprised of AM with BOS-driven encoding. The threshold values are tuned in a same way as before and set as γ = 0.7 and γ = 0.6 for BOM and CSLOGS data respectively (Figure 11).

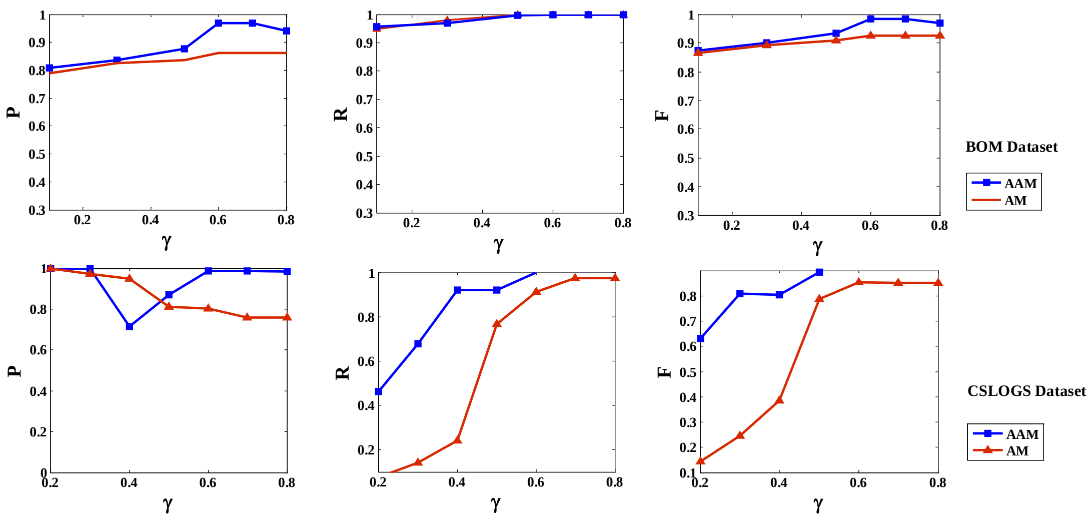


Figure 11: Precision (*P*), Recall (*R*) and FScore (*F*) curves with respect to γ over BOM and CSLOGS data

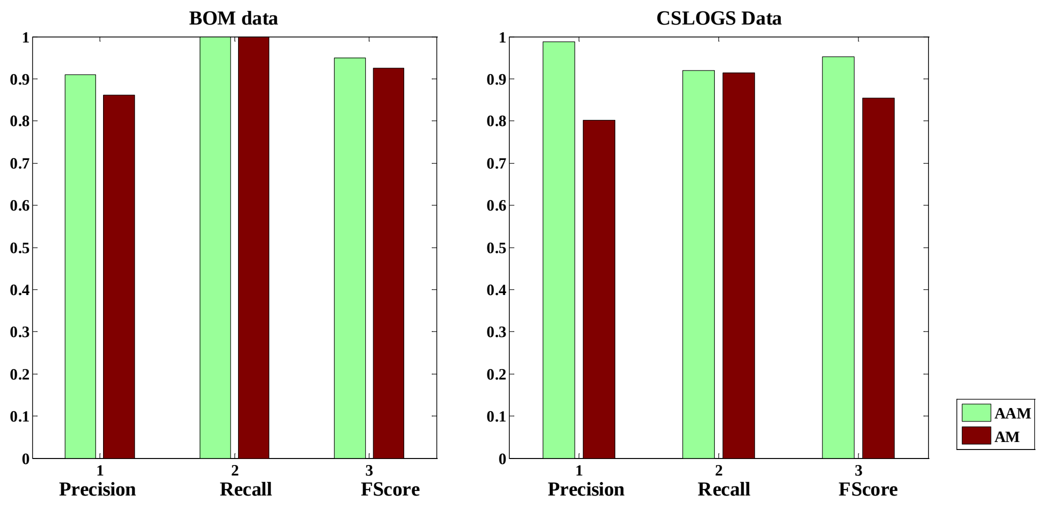


Figure 12: Performance of AAM and AM representations over BOM and CSLOGS data

Figure 12 shows the precision, recall and FScores for the BOM and CSLOGS data. It is evident that the AAM representation yields a better performance than the AM representation. The recall value of AM is just as high as AAM, as both of them are using BOS-driven encoding, which ensures the unique identity of each distinctive unordered tree, therefore all relevant examples are being retrieved in both cases.

The accuracy of a similarity measure method largely depends on how the intermittent steps are capturing the information of input objects that are being compared. AAM captures extra important features for representing a tree, and therefore it is able to outperform the basic AM representation.

5.4.3 Performance with Other Similarity Metrics

Several similarity metrics are available for Vector Space Model (VSM) representation; most of them can easily be applied to matrix representation by undertaking a simple modification. Cosine measure, Jaccard coefficient, Dice coefficient and Euclidean distance can be defined for matrices as:

$$S_{Cos}(T_1, T_2) = \cos(A', B') = \frac{\sum_{x=1}^{n}\sum_{y=1}^{n} A'_{xy}B'_{xy}}{\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} A'^2_{xy}}\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} B'^2_{xy}}} \tag{6}$$

Where, S_{Cos} is used for Cosine measure.

$$S_{Jac} = \frac{\sum_{x=1}^{|T|}\sum_{y=1}^{|T|} A'_{xy}B'_{xy}}{\sum_{x=1}^{|T|}\sum_{y=1}^{|T|} A'^2_{xy} + \sum_{x=1}^{|T|}\sum_{y=1}^{|T|} B'^2_{xy} - \sum_{x=1}^{|T|}\sum_{y=1}^{|T|} A'_{xy}B'_{xy}} \tag{10}$$

Where, S_{Jac} is used for Jaccard coefficient.

$$S_{Dice} = \frac{2\sum_{x=1}^{|T|}\sum_{y=1}^{|T|} A'_{xy}B'_{xy}}{\sum_{x=1}^{|T|}\sum_{y=1}^{|T|} A'^2_{xy} + \sum_{x=1}^{|T|}\sum_{y=1}^{|T|} B'^2_{xy}} \tag{11}$$

Where, S_{Dice} is used for dice coefficient.

$$D_{Euclidean} = \sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n} \left|A'^2_{xy} - B'^2_{xy}\right|} \tag{12}$$

Where, $D_{Euclidean}$ is representing the distance between two matrices, now using this distance the similarity score, $S_{Euclidean}$ between a tree pair is calculated as follows:

$$S_{Euclidean} = 1 - \frac{D_{Euclidean}}{\max(D_{Euclidean})} \tag{13}$$

The accuracy comparison in Figure 13 shows that cosine gives better precision than other measures. The cosine measure usually performs well when documents of varied length exist. Since both datasets include trees of diverse sizes, the cosine measure outperforms others.

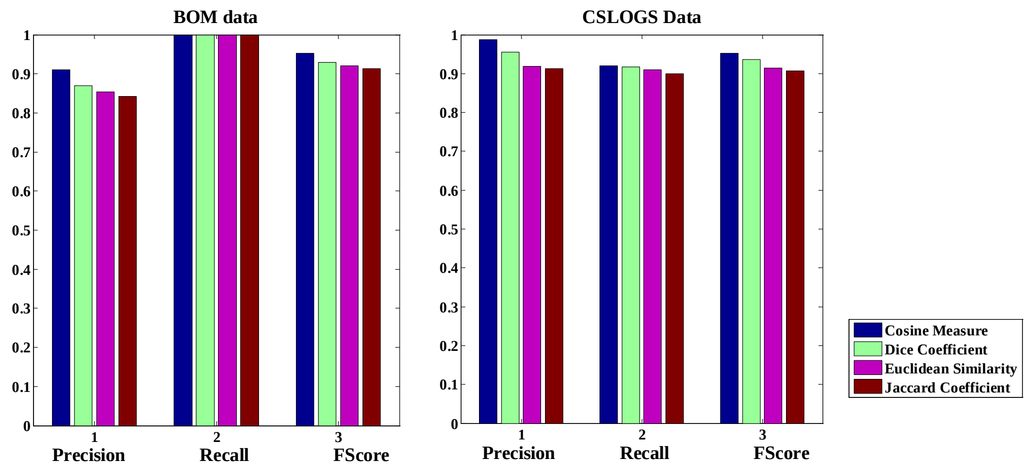


Figure 13: Performance of various measures over BOM and CSLOGS datasets

5.5 Results: Comparison with Tree Edit Distance Methods

5.5.1 Quality Comparison

Figure 14 shows that the proposed method (labelled as BOS+AAM) has achieved a minor improvement in accuracy over state-of-the-art tree edit distance-based methods. In this figure, CliqueEdit, UwCliqueEdit and DpCliqueEdit are abbreviated as CE, UCE and DCE respectively. All these methods perform similarly as they are developed over the similar concept. For both BOM and CSLOGS datasets, our method has a better FScore than the other methods. For, CSLOGS data the FScore difference with other methods is not very high as this dataset contains a tree with large structural variation; therefore any trivial method can distinguish between similar and non-similar trees, whereas BOM contains mostly homogeneous data and needs a sophisticated method to get accurate results. Our method considers intra structural relationships like hierarchical dependencies and optimal encoding, and hence achieves better results.

In reality, the tree edit distance methods are known to achieve high accuracy but they suffer from high computational complexity [42]. Achieving better accuracy than the tree-edit methods assures us that the proposed method does not compromise on accuracy when addressing the computational complexity problem. Let us see the runtime analysis next.

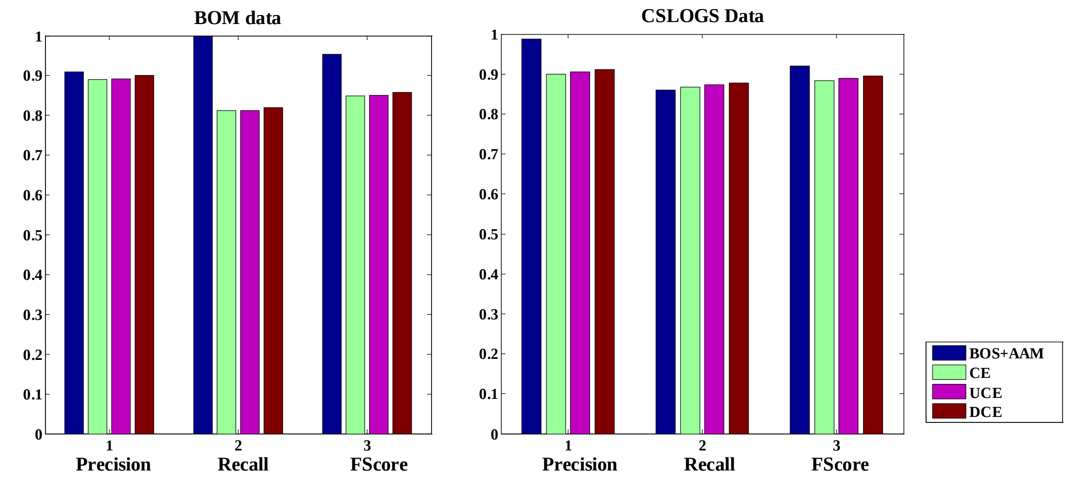


Figure 14: Accuracy performance of various tree edit distance based in comparison to the proposed method

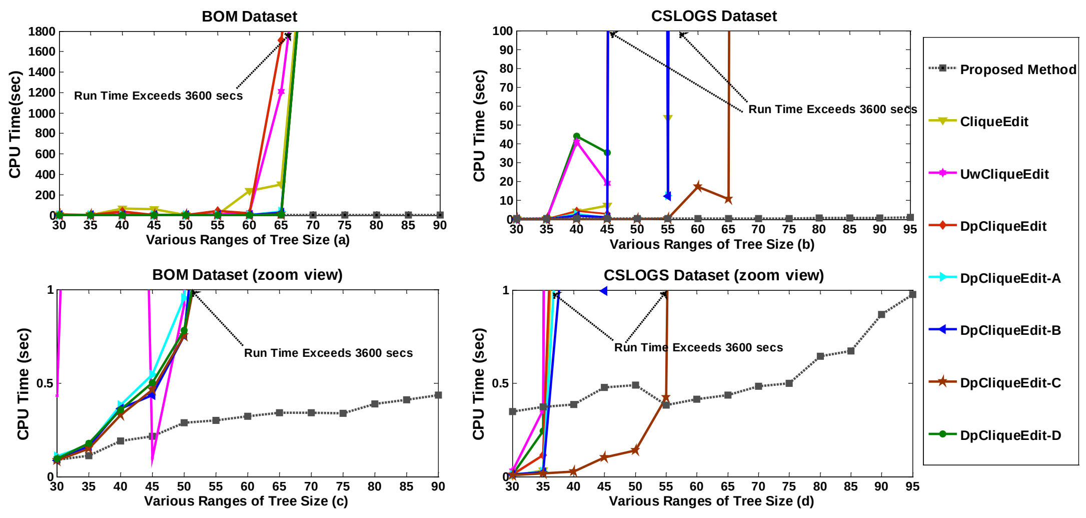


Figure 15: Run time comparison of all considered tree edit distance based methods vs proposed method over BOM and CSLOGS datasets

5.5.2 Running Time

The main contribution of the proposed method is that it can compute the similarity between unordered trees within polynomial time complexity, $O(n^2)$ whereas other methods have shown this problem to be intractable [45]. The benchmarking methods have also been designed to address this problem by ensuring fast computation. The runtime is reported as the average run time or CPU time per pair for all pairs within a specific tree size (maximum size among the tree pair) in the given datasets.

Results in Figure 15(a, b) reveal that the proposed method runs consistently faster than the existing methods. It displays polynomial complexity even for the trees of large size. In the BOM dataset (Figure 15(a)), all of the baseline algorithms show exponential complexity after reaching a tree size in the range of 60 ≅ 65 nodes. When the proposed method was showing very short runtime (less than a second), benchmarking methods were showing the runtime exceeding 3600 seconds for large sized trees. From the zoom view (Figure 15(c)), we can see the proposed method is able to achieve results within 0.5 seconds for all the considered ranges for BOM. In CSLOGS dataset (Figure 15(d)), the baseline methods exceed the 3600 seconds limit for trees of 30 ≅ 36 nodes in size, except DpCliqueEdit-C, which gives polynomial complexity up until the tree size reaches 55 nodes, whereas our method produced a solution within 1 second. Incorporating optimal navigation and matrix calculation into the proposed method allowed for the saving of a significant amount of computation time.

In summary, the proposed method achieved a small improvement in accuracy, however a very significant improvement in runtime over the existing tree edit methods.

5.5.3 Sensitivity Analysis

In the previous section, we observed that the runtime performance of the proposed method varies for different tree sizes as well as showing different runtimes for the same sized trees coming from different datasets. This indicates that the proposed method may be sensitive to some tree parameters. A series of sensitivity analyses is conducted with varied breadth, depth and size of the trees to find the reason of this uncertainty in output. Figure 16 and 17 display the performance of the proposed method by measuring runtime consumption (shown as the lines in graphs) with varied tree breadth, depth and size. Some subsets of the main data were created by varying a particular parameter while keeping other parameters constant. These figures also show the percentage of distribution of each case retrieved by varying a particular parameter in all over data (bar chart). So in a way, the number of trees of that particular parameter existing in the dataset can be known.

For testing the effect of tree breadth, the tree depth is fixed at 7 and tree size range is kept between 20 ≅ 29 nodes. For testing the effect of tree depth, the tree breadth was fixed at 4 and tree size was fixed at 20 ≅ 29 nodes. For testing the effect of tree size, the tree depth and breadth are fixed at 7 and 4 respectively. The reason behind choosing these parameters is because these are average parameters of the whole dataset. Besides, each of these cases reflects the majority distribution of the whole data.

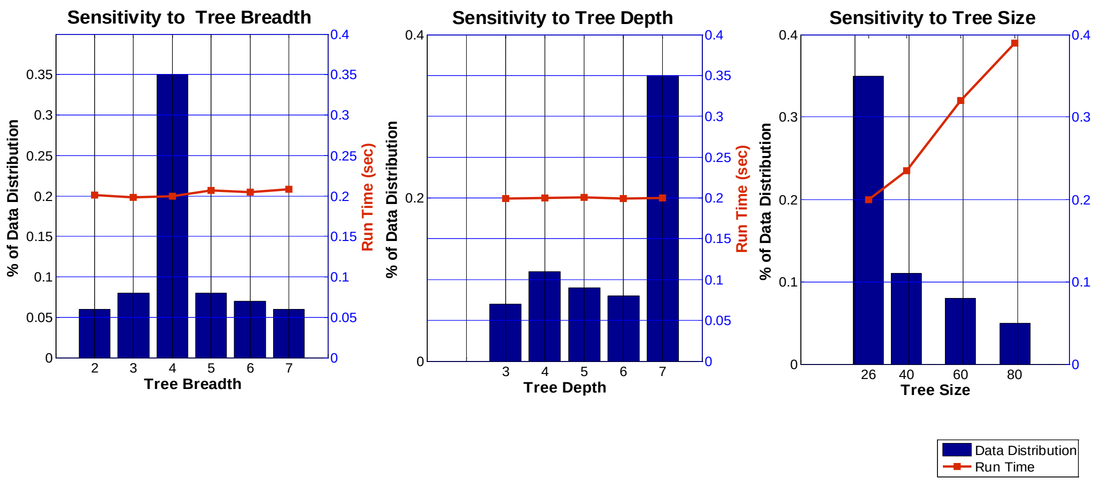


Figure 16: Sensitivity analysis over BOM Data

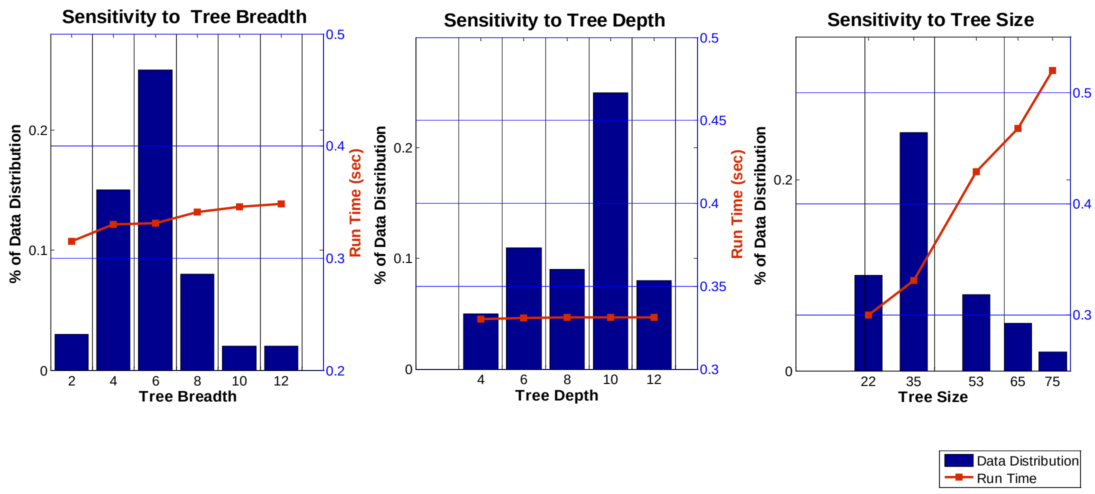


Figure 17: Sensitivity analysis over CSLOGS Data

For the CSLOGS dataset, a similar configuration is done using the following parameter values; depth = 10, breadth = 6 and tree size range = 30 ≅ 40 nodes. Figure 16 and 17, show that the proposed method is insensitive to tree depth, but slightly sensitive to tree breadth and when the values of the tree size increase, the time required for the computation of the similarity increases quadratically (line chart).

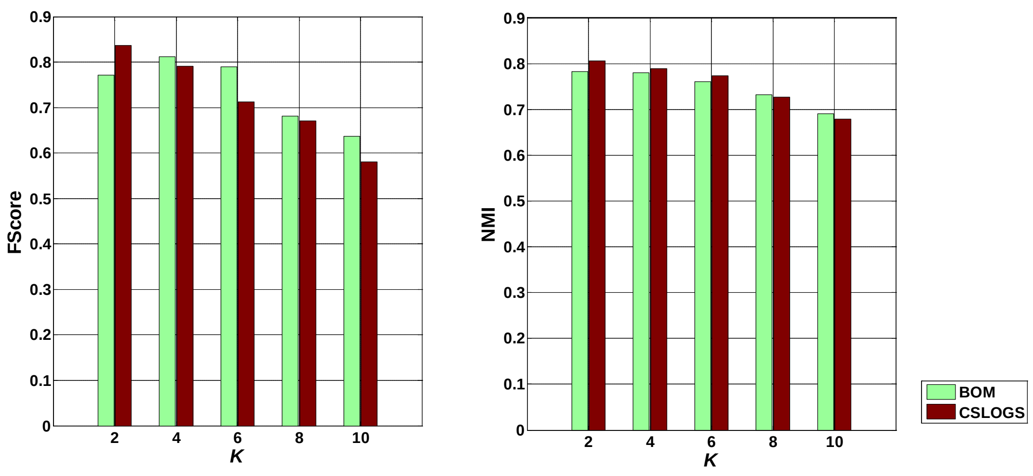


Figure 18: Clustering performance using the proposed similarity measure over BOM and CSLOGS datasets

5.6 Performance on a Similarity Measure Application

Clustering, classification, data integration and retrieval problems are some of the real-life applications of the proposed similarity measure method. To show one of these real-life applications, in this paper we have conducted clustering analysis on the pair-wise similarity matrix generated using the proposed method.

A clustering task on the tree data, like LOGML, BOM or XML, involves grouping them based on their similarity without any prior knowledge. Clustering has been frequently applied to group data based on the similarity of their content. However, tree data contains structural information with content that makes the clustering process more challenging [162]. The structure information is showed by the hierarchical relationship between the elements at various levels, which has been preserved while calculating pair-wise similarity in the proposed method. The majority of the existing algorithms utilise the tree-edit distance to compute the structural similarity between each pair of objects. This may lead to incorrect results, as the calculated tree-edit distance can be large for very similar trees.

The similarity matrix is fed to a partitional clustering algorithm such as *k* -way clustering [164]. The *k*-way clustering solution computes clustering by performing a sequence of *k* -1 repeated bisections. In this approach, the matrix is first clustered into two groups, and then one of these groups is chosen and bisected further. This process of bisection continues until the desired number of groups is reached. We chose partitional clustering because the incremental clustering technique for a given clustering threshold often generates a large number of clusters.

The k -way clustering algorithm option in CLUTO [164] is used to group both datasets to the required number of clusters. We varied the number of clusters k and recorded the value of evaluation metrics for both BOM and CSLOGS datasets. Figure 18 summarises the results, which ensure the reasonable performance of our similarity method based clustering. BOM data consists of four prominent classes, therefore better clustering performance was achieved when the value of k was set to 4, whereas for CSLOGS data the highest performance achieved was for when k = 2, as this dataset has two major classes. The results show that the similarity method proposed in this paper facilitates the final clustering solution of a data set.

6. CONCLUSION

Due to the strong representation capability of tree structured data, they have been commonly used in representing characteristics of real-life database applications. In this paper, based on optimisation, a novel tree traversal algorithm BOS has been proposed for unordered tree data. It is distinct from the existing approaches as it is order independent and ensures optimal traversing order for an unordered tree. This traversal order provides encoding of the nodes which enables us to represent the tree data with an efficient and equivalent matrix form, AAM. The BOS traversing and AAM representation facilitate the pair-wise similarity computation accurately and efficiently.

Empirical analysis showed that our method was able to achieve higher accuracy with less computation time in comparison to existing methods, even for large data sets. It requires only polynomial complexity, $O(n^2)$, whereas existing methods for calculating similarity between unordered trees suffer from the problem of high complexity and the problem has shown mostly as NP-hard or MAX-SNP hard. In the future, we will work on further improving the efficiency and scalability of our proposed method. We may consider data from other domains such as bioinformatics to check the versatility of the proposed method. Further, we are planning to expand the applicability of our proposed method into the area of information retrieval where the proposed method could be used in the filtering step or be used directly in a subtree query.

Paper 3: Identifying Product Families Using Data Mining Techniques in Manufacturing Paradigm

Israt Jahan Chowdhury* and Richi Nayak*

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

PUBLISHED IN: Australasian Data Mining Conference (AusDM 2014), Brisbane. Conferences in Research and Practice in Information Technology, Vol. 158. Richi Nayak, Xue Li, Lin Liu, Kok-Leong Ong, Yanchang Zhao, Paul Kennedy Eds

Abstract[4]**:** Identifying product families has been considered as an effective way to accommodate the increasing product varieties across the diverse market niches. In this paper, we propose a novel framework to identifying product families by using a similarity measure for a common product design data BOM (Bill of Materials) based on data mining techniques such as frequent mining and clustering. For calculating the similarity between BOMs, a novel Extended Augmented Adjacency Matrix (EAAM) representation is introduced that consists of information not only of the content and topology but also of the frequent structural dependency among the various parts of a product design. These EAAM representations of BOMs are compared to calculate the similarity between products and used as a clustering input to group the product families. When applied on a real-life manufacturing data, the proposed framework outperforms a current baseline that uses orthogonal Procrustes for grouping product families.

Keywords: Product families BOM, frequent mining, matrix representation, and clustering.

1. INTRODUCTION

Agile manufacturing has resulted in mass customisation and product proliferation, which consequently increases the number of products and part variations extensively. Simultaneously the current business climate demands for moving a product quickly from concept-to-market by reducing the product development lead time. A key element of shortening this lead time is the ability to use existing knowledge and designs to generate new variations of existing products, which ensure a reduction in time-to-market [173]. Therefore, the concept of grouping product families has been introduced. Besides leveraging product development cost, this grouping can offer multiple benefits including reduction in new product launching risks, improved ability to upgrade products, and enhanced flexibility and responsiveness of manufacturing processes [174]. For example, if two products have 45 out of 50 parts common, design of the similar parts can be reused and positioned

[4] Copyright (c) 2014, Australian Computer Society, Inc. This paper appeared at Australasian Data Mining Conference (AusDM 2014), Brisbane, 27-28 November 2014. Conferences in Research and Practice in Information Technology, Vol. 158. Richi Nayak, Xue Li, Lin Liu, Kok-Leong Ong, Yanchang Zhao, Paul Kennedy Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

for assembly early so that the remaining five parts can be added to the assembly when an order for a specific assembly has arrived. Exploring similarity among products may lead to the redesign of some parts.

Nowadays, with the advent of cheap storage and fast computer, a huge amount of data is generated during product design and development in a manufacturing system. The ability beyond human is required to process this huge amount of complex data into useful knowledge such as common product family information. The identification of product families is a non-trivial task due to the volume and complexity of the available data. A well-known historical approach of grouping product families is Group Technology (GT) [175, 176]. However, the practical acceptance of GT has been limited in modern manufacturing [23, 177], as it requires enormous effort to do groupings due to the involvement of manual intermittent steps for developing a "coding system" to summarise the key design and other attributes. Some efforts have been made towards automation [178], but acceptable performance is not reached yet, especially for situations where the sheer volume of data becomes overwhelming for both human and systems.

Data mining techniques have been specifically designed to deal with massive amount of data automatically (i.e. without human intervention) and to identify meaningful patterns and dependencies hidden behind the data. However, due to the complex nature of the data generated in product design domain, existing data mining algorithms require modifications. Although data mining algorithms have been specifically written to effectively analyse large datasets, the product design data often cannot be simply "plugged in" to these programs [179].

Bill of Materials (BOM) is a common product design data used in various domains like mechanical, electrical, electronic and civil/infrastructure. BOM is a hierarchical, structured representation of the products that details information such as parts descriptions, raw materials, quantities, manufacturing details, production times, etc. [23]. Researchers and practitioners have started using BOM specifications more commonly to represent their data [180]. It has become essential to propose similarity measures for BOM data to determine similarity between product designs, which will eventually lead to find effective groupings of product families.

For BOM data, the critical information lays in the recursive parent-child relationships between the end item, its components or subassemblies, and the raw (or purchased) materials they contain. This information can naturally be depicted in rooted labelled unordered tree format. In this paper we represent BOM data as unordered trees and introduce a novel matrix form called Extended Augmented Adjacency Matrix (EAAM) for equivalent tree representation. This representation facilitates search for similar designs and thus reduces the time consumption between concept and product launch. Our approach is to utilise the data mining techniques like frequent mining and clustering for ensuring efficient similarity calculation and reducing the search space for finding similar groups. Using frequent mining allows finding frequent structural dependencies like parent-child in a particular database, which gives the list of most occurred BOM parts or components relations. This information is then used with other content and topological information such as optimal part encoding, hierarchical position or level, and part quantity, in clustering. Using EAAM representations of BOM data, cosine similarity measure is used to generate a similarity matrix that becomes input to a clustering algorithm for identifying the product families.

When applied on a real-life manufacturing data, the proposed framework including the BOMs similarity measure method has proven to excel in solving the problem of grouping product families automatically. The results are also compared with a current baseline that uses Orthogonal Procrustes [77] for finding the product families and the proposed framework clearly outperforms.

Road map: In the following section, the related work is discussed. In Section 3, the background knowledge is presented. In Section 4, the proposed method for BOM similarity measure and the framework for identifying product families are given. The results are discussed in Section 5. In Section 6, the conclusion is drawn.

2. RELATED WORKS

Many efforts have been made for grouping the product families based on similarity schemes with emphasis on the different design areas and manufacturing. Most of them have focused on the historical approach of grouping individual parts into families, called as Group Technology (GT) [175, 176]. The practical acceptance of GT has remained limited due to the expensive coding system development for summarising the key product design and manufacturing attributes for doing the grouping. The main limitation of GT is the manual coding system. Though some efforts have been made towards automation, still more improvements are needed. Later, Authors in [181] used a back-propagation neural network based method for the product family grouping, but kept the existing GT classification and coding system. Another automated retrieval and ranking process for finding similar parts was proposed by authors in [178], but again based on GT coding. Authors in [182] employed genetic algorithm to form the families, however, this approach also required to use the existing classification and coding scheme.

Instead of using information derived from a fixed GT code; some methods proposed similarity based on product functional features. Authors in [183] used the Adaptive Resonance Theory neural network to develop a functional feature-based similarity method for grouping product families. Authors in [184] introduced another functional similarity-based combinatorial design method to produce a variety of products that satisfy various customer requirements in time. However, these functional feature-based schemes did not consider the hierarchical product design features. Authors in [77] attempted to calculate the similarity between BOMs considering the shape or geometrical structure, where a matrix representation and orthogonal Procrustes method were used to calculate the similarity score for grouping the product families. But BOMs are very flexible in shape, since there is no common rule or template to follow for generating them, therefore looking for geometrical or exact shape difference may give false similarity score. Emphasis should be put on the significant structural dependencies, hierarchical positions and other important contents during similarity calculation. The proposed framework in this paper focuses on the above for identifying the product families. To our best of knowledge, this is the first work on BOM data to determine product families using data mining.

3. BACKGROUND KNOWLEDGE

3.1 Bill of Materials (BOMs)

BOM represents hierarchical relations between various product parts with necessary details of manufacturing a particular product. It is a structural representation of a product including its required subassemblies, components and parts at various levels of production [185]. To understand the proposed framework, following definitions need to be considered.

Definition 1 (*End Items*): The entities that are sold directly to the customer without any further value added manufacturing step. End items usually contain several subassembly parts and raw materials and appear at the top of the BOM hierarchal position.

Definition 2 (*Subassemblies*): These are the entities that cannot be sold to the customer. Subassemblies may contain manufactured or purchased part or other subassemblies, and therefore, are appeared at a level of BOM hierarchy which is positioned neither at the top nor at the bottom.

Definition 3 (*Purchased Parts*): The raw materials which are the initial entities for finishing a final product. Purchased parts are positioned at the bottom level of the BOM structure.

Definition 4 (*Quantity Representation*): In BOM, repeated subassemblies or parts are represented by a quantity per value. This value is the number of the part required per one unit of the part's parent.

Definition 5 (*Part Number*): This is an alphanumeric string that uniquely identifies an end item, subassembly and a purchased part. Each number corresponds to a specific item with specific characteristics.

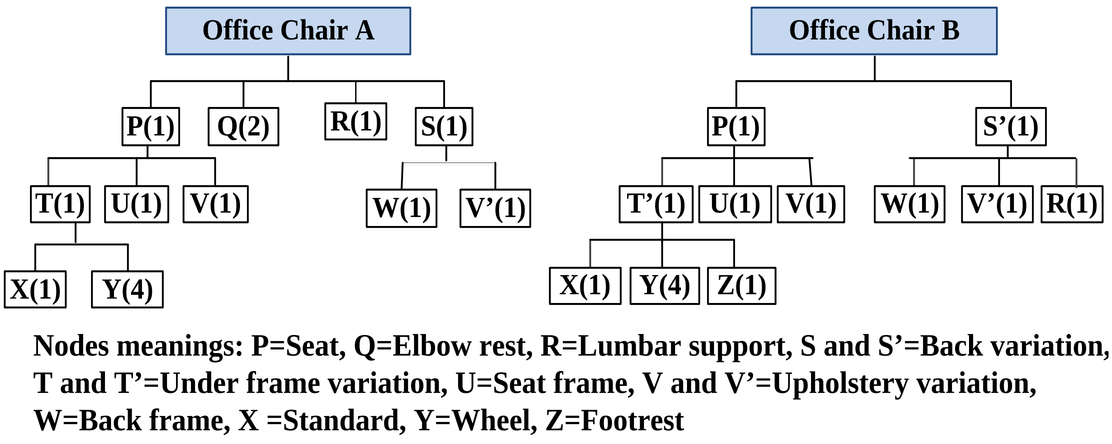

Office Chair A
P(1)
Q(2)
R(1)
S(1)
T(1)
U(1)
V(1)
W(1)
V'(1)
X(1)
Y(4)
Office Chair B
P(1)
S'(1)
T'(1)
U(1)
V(1)
W(1)
V'(1)
R(1)
X(1)
Y(4)
Z(1)
Nodes meanings: P=Seat, Q=Elbow rest, R=Lumbar support, S and S'=Back variation,
T and T'=Under frame variation, U=Seat frame, V and V'=Upholstery variation,
W=Back frame, X =Standard, Y=Wheel, Z=Footrest

Figure 1: Variants of office chair

Properties of BOM: BOM structures can be different for the identical end items, as each end item may be designed by a different company. Moreover, the product design is the result of human made input and developed completely based on individuals' understandings of how the product is manufactured or assembled. Similar BOMs may have different structures with same parts appearing at different level. However they will share similar components or parts and, most importantly, the structural dependencies among them will be usually kept same (Figure 1). BOMs substructures are unordered which means that the order of components is not significant. For instance, it does not matter if we say a chair has a seat, elbow rest and wheel, or a chair has a wheel, seat and elbow rest. In this paper we depicted BOM as rooted labelled unordered tree.

Definition 6 (*Unordered Tree*): A rooted labelled unordered tree has an identical root node and preserves only ancestor-descendant or parent-child relationships among nodes. There is no left-to-right order among the sibling nodes.

3.2 Data Mining Techniques Used

To satisfy the need of mass customisation and agile manufacturing, we need to apply techniques that will extract implicit, previously unknown, potentially useful and understandable pattern from a large database [1], thus the product design and manufacturing system will have substantial improvement. Using data mining techniques in advance manufacturing is becoming popular [7]. In the proposed framework, we have used frequent mining and clustering, two well-known data mining techniques for finding similarities between products and grouping them into families.

Frequent mining is used to extract interesting patterns from a database using a specified support [57, 58]. Support determines how often a pattern is applicable to or appears to a given data set. It represents the probability that a database instance contains that pattern.

BOM consists of structural dependencies like parent-child and ancestor-descendant relations between the end item, its components or subassemblies, and the raw (or purchased) materials they contain. The main challenge in BOM data analysis is dealing with the flexibility in its representation. It is very hard to put BOM data into a common format, thus the accurate analysis like similarity comparison can be carried out. Apparently, in BOM no other information keeps constant except the structural dependencies. So, instead of considering geometrical structure and shape, understanding structural dependencies is crucial for BOM similarity comparison. We utilise frequent mining to extract common structural dependencies in a database, which can be used as important representational component of the BOM data. These common structures can be input to clustering along with other information about the BOM data.

Clustering is an unsupervised data mining technique that can group objects based on their common characteristics, without the presence of any prior information about classification [162, 163]. Without using domain knowledge and GT coding based classification, the identification of product families can be possible using clustering. Clustering is now commonly used in manufacturing domain for doing unsupervised grouping [186]. To apply clustering, a similarity measure value needs to be calculated based on commonality of the features. In this work, we utilise cosine similarity [133] to determine a similarity matrix based on the equivalent Extended Augmented Adjacency Matrix (EAAM) of a BOM dataset.

4. PROPOSED BOM SIMILARITY FRAMEWORK FOR IDENTIFYING PRODUCT FAMILIES

In this section a method of similarity measure between two BOM data instances is presented. A framework is then proposed integrating the similarity measure for identifying product families.

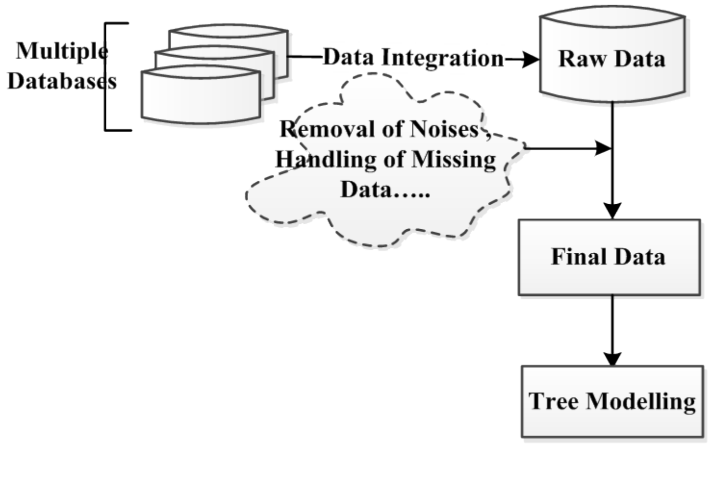


Figure 2: Data pre-processing steps

4.1 Data Pre-processing

To begin with our approach it is necessary to pre-process BOM data in order to make it useful for knowledge discovery. Figure 2 shows the tasks, which are used in this process.

4.1.1 Final Data

A company's database generally consists of a lot of data records. Only those records that correlate closely with the mining purpose are taken into account. Mostly BOM records are found in a tabular form, which typically contains the part name, part no, part revisions, part manufacturing description and the quantities required building a product assembly (as shown in Figure 3). Usually, the BOM input is given by human in spreadsheet, that can be formatted however one likes, but as anyone can format them, it often results in inconsistencies across a company's BOMs. Hence for mining BOM data, these inconsistencies need to be removed. Moreover not all of the information comprised by BOMs is necessarily mined for knowledge discovery. Therefore, once received the raw data through integration of multiple databases, the final data sets should be identified involving such data cleaning and filtering tasks as removal of noises, handling of missing data files, etc.

4.1.2 Unordered Tree Representation

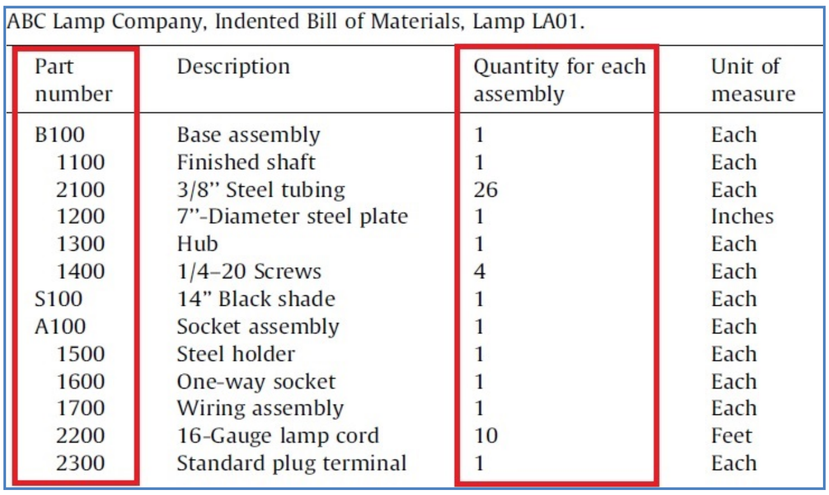
After identifying final BOM data, tree modelling is done to support the EAAM construction. This modelling is carried out by using unordered tree structure scheme as template, where only parent-child and ancestor-descendent relationships are important. The BOM data can naturally be represented as unordered tree. By considering the parent-child and ancestor-descendant relationships between end item, subassemblies and purchased parts, a mapping can be derived.

Table 1 shows a general mapping that can be used to represent the BOM data as unordered tree. The end item, or finished product, can be considered a root of the tree; manufactured or assembled components can become the nodes; purchased parts or raw materials can be the leaf nodes. For example, in Figure 3 the tabular or indented BOM of an ABC Lamps Product-LA01 [187] is given, where the lamp is the end product, and the parts given under first column are different subassemblies and purchased parts. For constructing a tree from this BOM only the relationships among various parts are important, such as B100, S100 and A100 are the children of the end item; 1100, 1200, 1300, 1400 are the children of B100, representing descendants of the end item.

Unordered Tree	BOM data
Root node	End item
Parent or ancestor node	End item and subassemblies
Child or descendant node	Subassemblies and purchased parts
Leaf node	Purchased parts
Parent-child or, ancestor-descendant relationships	End item-subassembly or, end item-purchased part or, subassembly-purchased part relationships
Node label	Part number

Table 1: Considered mapping for BOM to unordered tree representation

For node labelling, part numbers are used. If we compared two BOMs of product Lamp, using part numbers as labels, two BOMs would only match where the part numbers were exactly the same. For instance, suppose part S-14 is a shade with I.D. = 14” (inch). Part S-18 is a shade with I.D. = 18” (inch). These two shades would not be matched because of the unique part numbers. However, we are interested in finding BOMs of similar nature even if they do not share exact content and topology. For this reason, we replace the part numbers with general node labels derived from the part characteristics and types. In the case of these two parts, we would replace the unique part labels with a single label S for the class of shades.



ABC Lamp Company, Indented Bill of Materials, Lamp LA01.

Part number	Description	Quantity for each assembly	Unit of measure
B100	Base assembly	1	Each
1100	Finished shaft	1	Each
2100	3/8” Steel tubing	26	Each
1200	7”-Diameter steel plate	1	Inches
1300	Hub	1	Each
1400	1/4–20 Screws	4	Each
S100	14” Black shade	1	Each
A100	Socket assembly	1	Each
1500	Steel holder	1	Each
1600	One-way socket	1	Each
1700	Wiring assembly	1	Each
2200	16-Gauge lamp cord	10	Feet
2300	Standard plug terminal	1	Each

Figure 3: ABC lamps product-LA01 [187]

4.2 Finding Frequent Structural Relationship

The objective of the proposed framework is to form the product families based on the existing product models (BOMs). Due to the vast flexibility in BOM data, characterizing structural relationships based on frequent occurrence is essential to include in the global similarity calculation as in some cases, frequent-infrequent decision are used as a scale to measure the importance of the structural relations [49]. We consider these relationships as a representational component for the BOM dataset. We explain next how these relationships are derived.

4.2.1 Tree Traversal

Prior to implement frequent subtree mining algorithm, an optimal traversal [155] algorithm is used to ensure unique identity or canonical form [188] of each product model, which is in unordered tree form. Optimal traversal is included as it ensures optimality by providing unique encoding within minimum computation time [155]

4.2.2 Frequent Mining Algorithm

Once the canonical form is built, the frequent mining can now be applied that permits not only to explore the relationships and dependencies but also to handle a huge amount of data in an optimal way [57, 58]. However, such algorithms are sometimes limited to the memory because of its size and calculations that they perform. The candidate frequent subtrees generation can be exponential in large databases [49].

We propose to apply the BOSTER algorithm [57] which allows setting the subtree length equal to 1 and retrieves only single relationships exhibiting between parent-parts. This algorithm has proved to be memory efficient and exhibits limited computational complexity [57]. A support threshold is needed for frequent subtree mining process. A minimum support is set by trial and error, as it is a data specific parameter that prunes the infrequent subtree.

4.2.3 Characterizing Structural Relationships

Based on the result of the frequent subtree mining algorithm the structural relationships are characterized. If a subtree is frequent then the inherent parent child relation is considered as mandatory. Once all mandatory parent-child or ancestor-descendant relationships are identified, the remaining relationships are classified as optional. During the EAAM representation a weighted value of 1 and 0 are used to represent the mandatory and optional relationship respectively. The structural relationship importance weight is denoted as f_w, which contains binary variable.

4.3 Extended Augmented Adjacency Matrix (EAAM) Representation

In this paper a new matrix representation called EAAM is introduced. Although, EAAM is an extension of Augmented Adjacency Matrix (AAM) representation [155], but to our best knowledge, this is the first matrix, where the frequent structural relationship is included as one of the representational components. The rest of the components are:

- Optimal part sequence of BOM using optimal traversal.
- Part level information from BOM interface.
- Quantity representation (q) representing the number of the part required per unit of the part's parent.

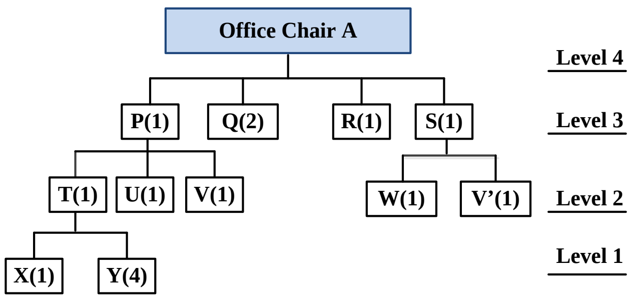
An adjacency matrix of a tree is based on the ordering chosen for the nodes [160]. For EAAM the ordering is achieved using optimal traversal [155] which ensures unique encoding of BOM represented in unordered tree form. For populating the cell of EAAM mainly structural relationship importance weight, level information and quantity representation are used.

Let a BOM, B is depicted as a rooted labelled unordered tree $B = (I, R)$, where $I = \{i_0, i_1, i_2, \ldots, i_n\}$ denotes the set of items with i_0 as end item, and other set elements as subassembly and purchased items, $R = \{(i_1, i_2)|i_1, i_2 \in I\} = \{r_1, r_2, \ldots, r_{n-1}\}$. The number of each item is given as $\{q_0, q_1, q_2, \ldots, q_n\}$. For B, the EAAM representation can be formulated in which a cell, a_{cd} is populated as follows:

$$a_{cd} = \begin{cases} 1 & \text{if } i_c \text{ is a node of } B \\ \dfrac{L(B, i_d)}{L(B, i_c)} + q_d + f_w & \text{if } i_c \text{ is a parent of } i_d \\ \dfrac{L(B, i_d)}{L(B, i_c)} + f_w & \text{if } i_c \text{ is an ancestor of } i_d \\ 0 & \text{otherwise} \end{cases}$$

These four components are explained as follows:

1. To represent the presence of each part in a BOM, each diagonal cell is populated with 1.
2. If the part is parent of the other respective part, then the cell is populated with level information (fraction of level of corresponding two nodes), weight information (quantity of the child node) and structural relationship importance weight value of 1 or 0 depending on the frequent or infrequent status of the parent-child relation in the respective database.
3. If the part is ancestor of the other respective part, then the cell is populated with level information (fraction of level of corresponding two nodes), and structural relationship importance weight value of 1 or 0 depending on the frequent or infrequent status of the ancestor-descendant relation.
4. If none of these are true, then the cell receives a value of 0.



	A	Q	P	S	T	Y
A	1	3/4+2+1	3/4+1+1	3/4+1+1	2/4+1	1/4+1
Q	0	1	0	0	0	0
P	0	0	1	0	2/3+1+1	1/3+4+1
S	0	0	0	1	0	0
T	0	0	0	0	1	0
Y	0	0	0	0	0	1

Figure 4: EAAM construction

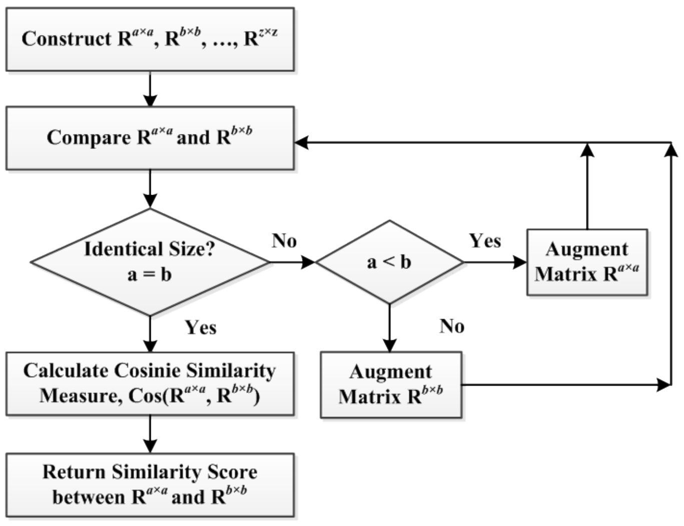
Example: From Figure 1, we consider the first example BOM of product model "office chair *A*" to explain the EAAM construction. Consider a BOM database that only consists of two BOM trees given in Figure 1, and the minimum support is two. It means that if a subtree appears twice or more in the database, it will be considered as a frequent sub-tree. Based on this, *A*-*Q*, *A*-*R*, *A*-*Z*, *T*-*Z* and *S*-*R* are found infrequent relationships and considered as optional. The order of the nodes for constructing EAAM is derived using optimal traversal. Consider the cell between nodes *A* and *Q*. For this BOM tree, *A* is the parent of part *Q*, therefore the level information is added as 3/4, where the level of *A* is 4 and the level of *Q* is 3. For the child part *Q*, the quantity representation value is 2, which is added after the fraction of level into that cell. Finally, the frequent parent-part relation adds a value 1 to indicate the mandatory relationship. The overall calculated value for this cell is 3/4+2+1 (Figure 4). The rest of the cell values are calculated following the same way.

4.4 BOM Similarity Measure

After constructing EAAMs, we use cosine similarity for matrix comparison for measuring the similarities between a BOM pair [155] as follows:

$$\cos(\mathrm{A},\mathrm{B}) = \frac{\sum_{x=1}^{n}\sum_{y=1}^{n}\mathrm{A}_{xy}\,\mathrm{B}_{xy}}{\sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n}\mathrm{A}_{xy}^{2}}\ \sqrt{\sum_{x=1}^{n}\sum_{y=1}^{n}\mathrm{B}_{xy}^{2}}}$$

Where, A and B are two (n×n) matrices.



Construct $R^{a\times a}$, $R^{b\times b}$, …, $R^{z\times z}$
Compare $R^{a\times a}$ and $R^{b\times b}$
Identical Size? a = b
No
a < b
Yes
Augment Matrix $R^{a\times a}$
Yes
No
Calculate Cosinie Similarity Measure, Cos($R^{a\times a}$, $R^{b\times b}$)
Augment Matrix $R^{b\times b}$
Return Similarity Score between $R^{a\times a}$ and $R^{b\times b}$

Figure 5: The flow chart of calculating similarity

If sizes of the two BOM trees are not same, then additional columns and rows with zero elements are padded to the smaller matrix for making the size of both matrices equal, this is called the augmentation of matrix. These two square matrices can be considered as two $|B|\times|B|$ (where $|B|$ = max $\{B1, B2\}$; B_1, B_1 are two BOM trees) dimensional vectors. The overall procedure for similarity measure is given in Figure 5 using a flow chart, where matrix is represented as $\mathbf{R}^{a\times a}$, where a is the size of that matrix representing the number of the components or parts in a BOM tree.

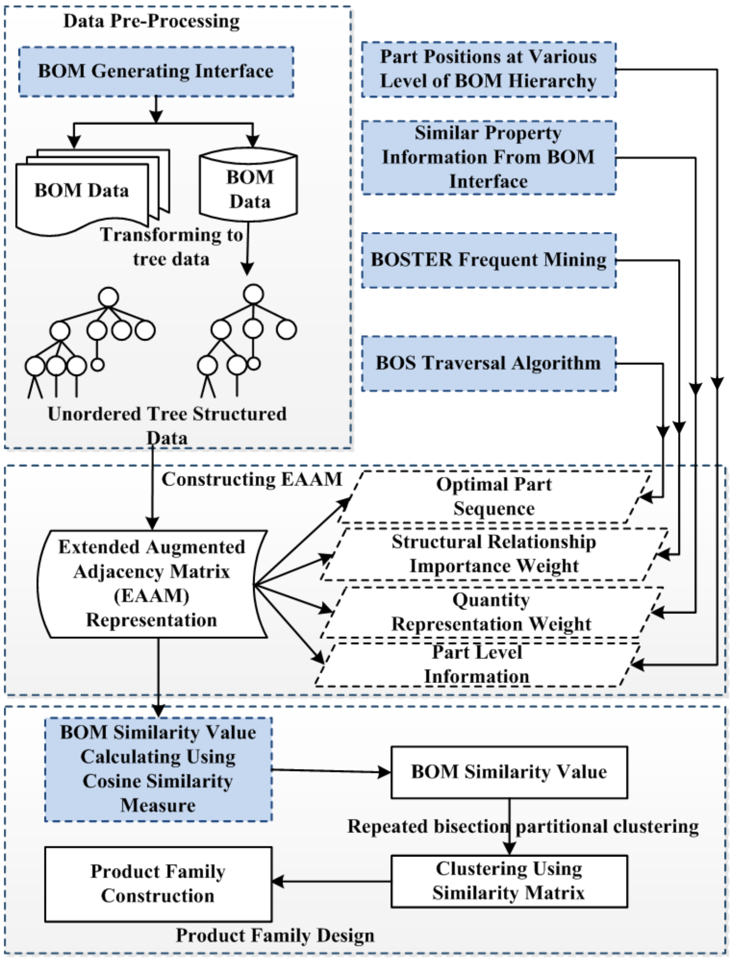

Data Pre-Processing
BOM Generating Interface
BOM Data
BOM Data
Transforming to tree data
Unordered Tree Structured Data
Part Positions at Various Level of BOM Hierarchy
Similar Property Information From BOM Interface
BOSTER Frequent Mining
BOS Traversal Algorithm
Constructing EAAM
Extended Augmented Adjacency Matrix (EAAM) Representation
Optimal Part Sequence
Structural Relationship Importance Weight
Quantity Representation Weight
Part Level Information
BOM Similarity Value Calculating Using Cosine Similarity Measure
BOM Similarity Value
Repeated bisection partitional clustering
Clustering Using Similarity Matrix
Product Family Construction
Product Family Design

Figure 6: Framework for product family design

4.4 The Proposed Framework

The proposed framework for grouping product families has three main phases as shown in Figure 6. In the first phase data pre-processing is done. BOM has different storage under different enterprises; some of them store BOM data in database, some in files like XLS file. Some enterprises use part table/relationship table to express BOM, and some enterprises use a single table. All these variations need to save in memory as a BOM generating interphase, from this node the pre-processing will carry out in next. Next phase covers the EAAM construction where all necessary steps (dotted blue boxes) are implemented for populating the feature weights. In the third and final phase, the pairwise similarity is calculated using the EAAM comparison and a similarity score is calculated between BOM pairs where a similarity score of 0 means completely dissimilar and a score of 1 means exactly similar. Using this similarity values a similarity matrix is constructed which is then employed as an input to a clustering algorithm. Table 2 shows an example of the similarity matrix. We used a well-known clustering algorithm, Repeated Bisection Partitioning [189], for grouping the BOMs into families. This algorithm divides trees into two groups and then selects one of the larger groups according to a clustering criterion function and bisects further. This process is repeated until the desired number of clusters is achieved. During each step of bisection, the cluster is bisected so that the resulting 2-way clustering solution locally optimises a particular criterion function. Other clustering algorithms can also be applied. Finally from the cluster result, the product families will be identified.

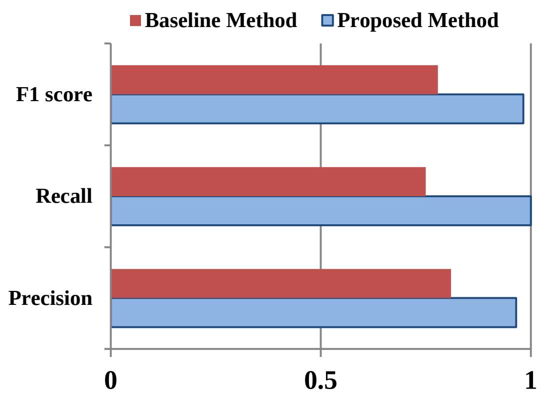
	B1	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**	**B11**	**B12**	**B13**	**B14**	**B15**
B1	1.00	0.40	0.43	0.57	1.00	0.50	0.61	1.00	1.00	0.64	0.41	0.30	0.41	0.58	0.44
B2	0.40	1.00	0.43	0.47	0.40	0.49	0.37	0.40	0.40	0.42	0.32	0.43	0.32	0.54	0.39
B3	0.43	0.43	1.00	0.65	0.43	0.53	0.43	0.43	0.43	0.45	0.39	0.44	0.39	0.52	0.33
B4	0.57	0.47	0.65	1.00	0.57	0.70	0.60	0.57	0.57	0.71	0.50	0.35	0.50	0.63	0.34
B5	1.00	0.40	0.43	0.57	1.00	0.50	0.61	1.00	1.00	0.64	0.41	0.30	0.41	0.58	0.44
B6	0.50	0.49	0.53	0.70	0.50	1.00	0.62	0.50	0.50	0.71	0.65	0.34	0.65	0.71	0.35
B7	0.61	0.37	0.43	0.60	0.61	0.62	1.00	0.61	0.61	0.58	0.56	0.33	0.56	0.58	0.41
B8	1.00	0.40	0.43	0.57	1.00	0.50	0.61	1.00	1.00	0.64	0.41	0.30	0.41	0.58	0.44
B9	1.00	0.40	0.43	0.57	1.00	0.50	0.61	1.00	1.00	0.64	0.41	0.30	0.41	0.58	0.44
B10	0.64	0.42	0.45	0.71	0.64	0.71	0.58	0.64	0.64	1.00	0.56	0.31	0.56	0.72	0.39
B11	0.41	0.32	0.39	0.50	0.41	0.65	0.56	0.41	0.41	0.56	1.00	0.25	1.00	0.47	0.31
B12	0.30	0.43	0.44	0.35	0.30	0.34	0.33	0.30	0.30	0.31	0.25	1.00	0.25	0.42	0.31
B13	0.41	0.32	0.39	0.50	0.41	0.65	0.56	0.41	0.41	0.56	1.00	0.25	1.00	0.47	0.31
B14	0.58	0.54	0.52	0.63	0.58	0.71	0.58	0.58	0.58	0.72	0.47	0.42	0.47	1.00	0.31
B15	0.44	0.39	0.33	0.34	0.44	0.35	0.41	0.44	0.44	0.39	0.31	0.31	0.31	0.31	1.00

Table 2: BOM similarity matrix

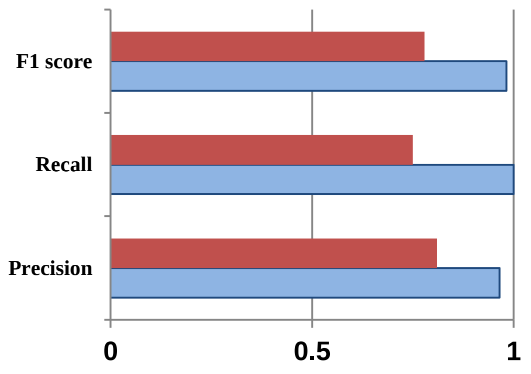
5. EVALUATION OF THE PROPOSED FRAMEWORK

We implemented the proposed framework on a real manufacturing data to evaluate the performance. This data is collected from a manufacturer of nurse calling devices [68]. It consists of 404 BOMs with four major product families. From this data set we randomly generated four samples, consisting 100 BOMs each and named them as Data 1, Data 2, Data 3 and Data 4. We used all these four datasets for empirical analysis.

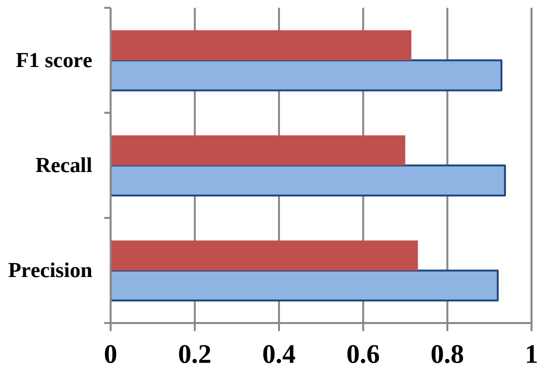
For benchmarking we consider a method that used the orthogonal Procrustes problem to find the orthogonal matrix for two given matrices that will closely map one matrix to another and used this as a geometrical similarities between BOMs and then clustered them into families [77]. For the benchmark method we used the same clustering algorithm, but we used the orthogonal Procrustes based similarity measure as input and performed the product grouping. Finally we checked the clustering results with the known product family information and compared the performances.



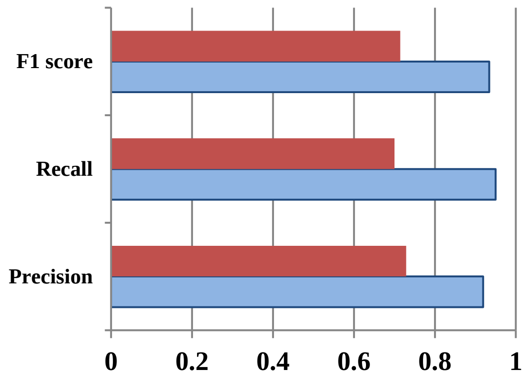
(a) Data 1



(b) Data 2



(c) Data 3



(d) Data 4

Figure 7: Accuracy performance over Data 1(a), Data 2(b), Data 3(c) and Data4 (c)

The main contribution of this paper is the similarity measure method of product BOMs. An efficient grouping of product families largely depends on an efficient similarity measure method. We evaluated our similarity measure approach using the well-known evaluation metrics including precision, recall and F1 score [190] and performed on all four data samples. For these metrics, the value close to 1 is considered as an indication of better performance. From Figure 7, we can see for all four data sets our proposed similarity measure method gives higher accuracy in comparison to the benchmark method. This good accuracy performance should also reflect during the clustering process, as we used this similarity method as an input for an off-the-self clustering algorithm for doing the product family grouping. Table 2 gives a partial view of the similarity matrix generated by our proposed BOM similarity measure method. For clustering we used this similarity matrix for identifying product families.

Table 3 reports the clustering performance results, where we mainly included the number of mis-clustered product BOM for each data by the proposed method and the benchmarked method. The proposed framework outperforms the baseline method.

Method	**Data 1**	**Data 2**	**Data 3**	**Data 4**
Proposed Framework	2	5	5	6
Baseline Method	19	21	25	35

Table 3: Number of Mis-Clustered BOMs for Different Data Sets

6. CONCLUSION

A product family is a group of related products based on a product platform, facilitating mass customisation by cost-effectively providing a variety of products for different market segments. In this paper we present a data mining approach based framework for grouping various products into families. We introduced a similarity measure method for a common product data type, BOM that can be used to cluster products into families. The benchmarking results confirm the efficiency of the proposed work.

In future work, we intend to expand the study on unifying the families into a single Generic Bill of Material (GBOM) [191] group.

Chapter 5: Frequent Subtree Mining

Frequent subtree mining is one of the major contributions of this thesis. Three efficient algorithms for mining frequent subtrees from databases of labelled unordered trees are proposed, which utilise the novel canonical representation BOCF. Each of these algorithms is published as a separate publication. The first paper presents the BOSTER algorithm, which is designed for mining frequent rooted unordered induced subtrees. The second paper presents the BEST algorithm, which is designed for mining frequent rooted unordered embedded subtrees. The third paper presents the algorithm FreeS for mining frequent free induced subtrees. All of these algorithms work toward achieving an optimal candidate generation process with a good growth strategy as well as avoiding the generation of false candidate trees, with a focus on a specific frequent subtree mining problem. Empirical analysis in these papers shows that these algorithms have proven their efficiency in dealing with the isomorphism and automorphism problem which is a pressing issue in the process of frequent rooted unordered and free subtree mining. Empirical analysis of each method also shows its superiority in efficiency of generating patterns in comparison to the corresponding state-of-the-art benchmarking methods.

Algorithm	**Input Tree type**	**Output Subtree type**
BOSTER	Rooted Unordered Tree	Induced Subtree
BEST	Rooted Unordered Tree	Embedded Subtree
FreeS	Free Tree	Induced Subtree

Table 5.1: A general overview of the proposed frequent subtree mining algorithms

Table 5.1 presents the general overview of the proposed frequent subtree mining algorithms in this chapter. This table mainly includes the information of input tree type and subtree type for which each of these algorithms is specially designed and the input database on which these algorithms can be applied.

This chapter is organised based on three papers that introduce the proposed algorithms and follow the sequence of Paper 4 on BOSTER, Paper 5 on BEST and

Paper 6 on FreeS. Before presenting each paper in its original form, a brief overview of each method is provided along with some of materials that were excluded from the papers due to space restrictions enforced by the publishers.

5.1 BOSTER: AN EFFICIENT ALGORITHM FOR MINING FREQUENT UNORDERED INDUCED SUBTREES

This paper focuses on designing, developing and testing the BOSTER algorithm for mining frequent induced subtrees from a database of labelled unordered trees. This paper first introduces the novel BOCF canonical form for representing the rooted unordered trees. To the best of our knowledge, this is the first tree mining algorithm that does not require additional isomorphism and automorphism checking during frequency counting. To ensure optimal enumeration, a tree structure, guided scheme-based enumeration tree is proposed for candidate generation. This enumeration approach uses tree weight, level and fan-out information to guide the candidate generation process. The enumeration tree is expanded with patterns using the extension and join operations defined to support BOCF and the structure guided enumeration. In order to limit the number of candidates, the growth rules are introduced that control the availability of right most nodes to be used in extension for candidate generation. Consequently, by using this approach of candidate generation, BOSTER is able to generate only valid subtrees, which results in saving time and memory by avoiding the generation of invalid subtrees and then pruning later on. Most of the existing algorithms spend a fair bit of time on checking whether the generated candidate subtrees are in considered canonical form or not, in order to remove the invalid subtrees. The modified occurrence list based frequency counting method is used to improve the efficiency.

BOSTER is evaluated with both the synthetic data and real life data. The real life data CSLOGS is used, which is a most commonly used data set for evaluating frequent subtree mining algorithms [34, 36, 49, 70, 192]. The two most relevant and state-of-the-art algorithms - UNI3 and HybridTreeMiner (HBT) - are used for benchmarking. BOSTER proved its scalability and efficiency in most cases. It particularly works efficiently (takes less computation time) when a dataset is more likely to have isomorphic trees.

Due to the space constraints, some of the results were not part of the published article and the results presented in the paper focused on showing the scalability of

BOSTER. Figure 5.1 shows the impact of using the BOCF canonical form when the datasets include isomorphic trees. In addition to benefitting BOSTER, this provides a greater benefit to the BEST algorithm for finding embedded subtrees from the unordered tree data, as well as to the FreeS algorithm for finding induced subtrees from the free tree data. As in these two algorithms, the problem of isomorphism and automorphism has greater impact, due to dealing with more flexibility in trees/subtrees.

5.1.1 BOSTER Handling Isomorphism

BOSTER uses the BOCF canonical form for tree representation, which makes it efficient to deal with the isomorphism problem. To empirically evaluate this statement, a synthetic dataset is generated based on the following parameters where Zaki’s tree generator [38] is used:

- The number of labels (N) = 50,
- The number of nodes (M) = 1,000,
- The maximum depth (D) = 5,
- The maximum fan-out of a node (F) = 5,
- The total number of trees in the dataset (T) = 10,000.

Due to the setting of node labels to very small with large number of nodes and trees, this data will have a high probability of getting a huge number of isomorphic trees. This would cause the presence of a large number of overlapping trees in the dataset, and a tree mining method would have to deal with this issue.

In this isomorphism test, the results are compared against two benchmarking algorithms UNI3 [98] and HybridTreeMiner (abbreviated as HBT) [96], which are also designed to mine frequent rooted unordered induced subtrees. As shown in Figure 5.1. BOSTER consumes the least runtime followed by UNI3. HBT has to be aborted due to high runtime for smaller support thresholds. HBT needs an exclusive isomorphism test for avoiding overlapping trees and, therefore, requires high runtime. UNI3 deals with the isomorphism problem by creating a separate embedding list, which allows saving runtime in comparison to HBT, but BOSTER is still the fastest one because of using BOCF canonical form. Since BOCF does not allow generation of isomorphic trees, no additional checking test is carried out. This result ascertains that BOSTER is robust to the problem of isomorphism.

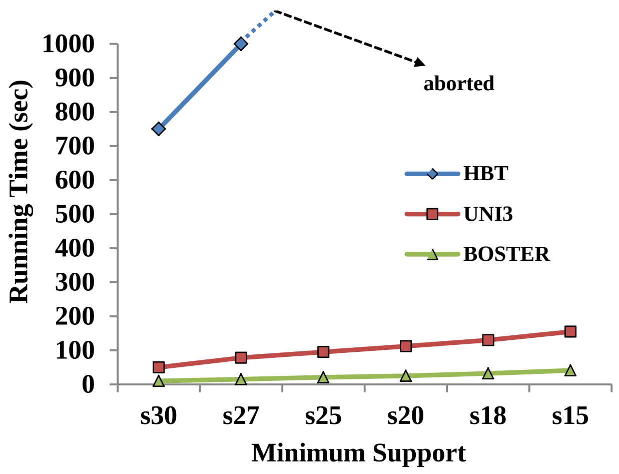


Figure 5.1: Runtime in presence of isomorphism

5.2 BEST: AN EFFICIENT ALGORITHM FOR MINING FREQUENT UNORDERED EMBEDDED SUBTREES

This paper contains the detail of the BEST algorithm, which mines frequent embedded subtrees from a database of labelled unordered trees. The BEST algorithm utilises the BOCF canonical form for representing the rooted unordered trees. In this paper, distinct properties including lemmas and proofs of balance optimal canonical form are presented.

Mining embedded subtrees can be considered as a generalised problem of mining induced subtrees, but the difficulty level of this problem is higher than the induced subtree mining problem [49, 97]. Embedded subtree mining requires examining several levels within a tree to identify an embedded subtree. BEST incorporates level conditions during candidate generation that are represented in BOCF forms and the enumeration operations are defined accordingly. The structure guided enumeration tree allows avoiding invalid candidate subtree generation, which makes BEST more time and memory efficient than the existing benchmarking algorithms like SLEUTH [70] and U3 [97]. BEST holds the downward closure lemma during its processing and avoids generation of pseudo frequent [154, 192] subtrees, which SLEUTH fails to do. Therefore during the test, SLEUTH extracted the higher number of subtrees as frequent in comparison to U3 and BEST. Both synthetic and real life datasets are used for evaluating this algorithm. BEST ensures the least runtime consumption on a real life dataset for a support (1.5%) without missing any frequent patterns. For the synthetic datasets, BEST also shows the competitive performance.

5.3 FREES: A FAST ALGORITHM TO DISCOVER FREQUENT FREE SUBTREES USING A NOVEL CANONICAL FORM

This paper presents the FreeS algorithm for mining frequent free induced subtrees from a database of labelled free trees. The BOCF tree representation is extended for distinctively representing free trees despite the presence of isomorphism. The BOCF of a free tree is generated by identifying a root node uniquely, which is called normalisation. FreeS uses a tree structure guided scheme-based enumeration for generating the candidate of free subtrees. This scheme includes a set of conditions that conforms generation of candidate free subtrees in their canonical forms. Required lemmas and respective proofs are provided in this paper. For growing the enumeration trees, both the extension and join operations are used which are defined to support the generation of candidate free subtrees. For counting frequency, a modified occurrence list based support is used. Finally, the performance of FreeS is checked against the state-of-the-art free tree mining algorithms FreeTreeMiner [63] and HBT [96]. Empirical analysis confirms that FreeS is faster and computationally efficient.

In this thesis only the frequent free induced subtree mining problem is addressed, and the frequent embedded subtree mining problem is not studied due to time constraints. The frequent embedded subtrees can be extracted by adding new conditions in the enumeration tree, which will allow generation of candidate embedded subtrees only. To the best of the authors' knowledge, there is no available work on mining frequent free embedded subtrees, so the evaluation process will be challenging in terms of benchmarking.

NB: The reader may be found the published paper a bit different than the version of the paper added in this thesis. This is done to correct some confusing wordings, which does not change any core concept of the work.

Paper 4: BOSTER: An Efficient Algorithm for Mining Frequent Unordered Induced Subtrees

Israt Jahan Chowdhury* and Richi Nayak*****

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

PUBLISHED IN: Lecture Note in Computer Science: Web Information Systems Engineering – WISE 2014, 8786, pp. 146-155

Abstract[5]**:** Extracting frequent subtree from the tree structured data has important applications in Web mining. In this paper, we introduce a novel canonical form for rooted labelled unordered trees called the **b**alanced-**o**ptimal-search **c**anonical **f**orm (BOCF) that can handle the isomorphism problem efficiently. Using BOCF, we define a tree structure guided scheme based enumeration approach that systematically enumerates only the valid subtrees. Finally, we present the **b**alanced **o**ptimal **s**earch **t**ree min**er** (BOSTER) algorithm based on BOCF and the proposed enumeration approach, for finding frequent induced subtrees from a database of labelled rooted unordered trees. Experiments on the real datasets compare the efficiency of BOSTER over the two state-of-the-art algorithms for mining induced unordered subtrees, HybridTreeMiner and UNI3. The results are encouraging.

Keywords: Web mining, frequent subtrees, labelled rooted unordered trees, induced subtrees, canonical form, enumeration approach.

1. INTRODUCTION

In order to improve the Web-based applications, finding frequent patterns is a common task in Web usage mining that discovers useful information from the Web data. The web usage data, the sequences of accesses pursued by users, can be easily represented as trees [193]. The frequent subtree mining task can be used in distinguishing various users according to their common browsing behaviour [50].

In this paper we study the problem of finding frequent subtrees from the database of unordered trees.

Unordered trees have shown the capability of identifying interesting relations due to not being constrained by sibling order (i.e. no fixed left-to-right order among sibling nodes) [29]. However, this distinct property makes the process of mining frequent unordered subtrees more challenging in comparison to ordered trees. Exponential candidate generation with redundancy is the main problem in mining frequent unordered subtrees. It is critical to determine a “good” growth strategy as there can be many possible ways to extend a candidate subtree due to not having sibling order constraint. Moreover, high computation and memory expense are

[5] B. Benatallah et al. (Eds.): WISE 2014, Part 1, LNAI 8786, pp. 146–155, 2014.
© Springer International Publishing Switzerland 2014

always an issue for mining tree data. Many algorithms have been proposed to overcome these challenges where they use a canonical form, and extend the candidates only that conform to the canonical form. Several canonical representations based on sorted pre-order string [93], depth-first traversal [90-92] and breadth-first traversal [96] have been proposed. These canonical forms need an additional isomorphism test for avoiding redundancy problem. Besides, the existing algorithms use extension and join operations for candidate enumeration [53, 96], which produce a large number of candidates including invalid subtrees. Authors in [98] have developed an enumeration approach using underlying tree structure information that generates only valid subtrees, but, the method suffers from extensive memory usage.

We have previously proposed an optimal tree traversal algorithm for traversing a rooted unordered tree [155] and finding similarity amongst tree data. In this paper, we extend this traversing algorithm by introducing a new heuristic that leads towards a new definition of canonical form for representing unordered trees, called the balanced-optimal canonical form (BOCF). The BOCF can alleviate redundancy problem as it is able to represent unordered trees uniquely even in the presence of isomorphism. Using BOCF, we specify an optimal enumeration approach to systematically enumerate all frequent subtrees based on underlying tree structure information. This enumeration approach is efficient as it restricts the search, by only generating the unambiguous and valid subtrees using the underlying tree structure information. Finally, the balanced optimal search tree miner (BOSTER) algorithm is proposed for mining frequent induced unordered subtrees from a database of labelled rooted unordered trees. Empirical analysis carried out using a real data has shown the effectiveness of BOSTER over the two state-of-the-art algorithms, HybridTreeMiner [96] and UNI3 [98].

2. PRELIMINARIES

Let $T = (V, E, L)$ be a *rooted labeled unordered* tree, where $V = \{v_0, v_1, v_2, \ldots, v_n\}$ denotes the set of nodes with v_0 as *root* node, $E = \{(v_i, v_j)| v_i, v_j \in V\} = \{e_1, e_2, \ldots, e_{n\text{-}1}\}$ denotes the set of edges and L denotes the set of labels. The label is given by a function Φ: $V \rightarrow L$ which maps nodes with unique labels. An unordered tree has no ordering relationship among the nodes except ancestor-descendent or parent-child.

The ancestor-descendent relationship between two nodes is denoted by $v_i \prec v_j$, i.e. v_i is ancestor of v_j, the '$\prec$' symbol represents 'precedes'. The level of a node v_i in a tree T is denoted as $Lv(T, v_i)$ and the height of a tree T is denoted as $H(T)$.

Definition 1 (*Induced Subtrees*): A tree $T'(V', L', E')$ is an unordered *induced subtree* of a tree T (V, L, E) iff: (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) $L' \subseteq L$ and the labelling of V' in T is preserved in T' (4) $\forall v_i' \in V'$, $\forall v_i \in V$ and v_i' is not the root node, then parent of v_i'= parent of v_i, and (5) no left-to-right ordering among the siblings in T is preserved among the corresponding nodes in T'.

Definition 2 (*Equivalent Node*): If two nodes v_i and v_j of a tree T, have the same label originated from the same labelled parent node (parent of v_i = parent of v_j) and have the same labelled child nodes then they are called *equivalent nodes,* denoted by $v_i \cong v_j$.

Definition 3 (*Weight of Node*): *Weight* of a node v_i ($v_i \neq v_0$) is defined as the total number of its equivalent node and denoted by w_i (Figure 1).

According to the properties of unordered trees we have Lemma 1.

Lemma 1 *Weight of the root node v_0 is always zero, $w_0 = 0$. For each node $v_i \in V$ ($v_i \neq v_0$), the weight w_i ($w_i \neq w_0$) should always have a minimum value of one.*

PROOF:

1. *According to the tree structure schema no equivalent node of a root node is possible as its ancestors are undefined. Hence, the weight of the root is always zero.*
2. *Each node v_i ($v_i \neq v_0$) of tree T should have at least one equivalent node, otherwise v_i doesn't belong to that tree. Hence, the minimum weight of the node is one, w_i = 1. For node v_i, $w_i > 1$ if the node has more than one equivalent node.*

Definition 4 (*Mining Unordered Induced Subtree*): Let T_{db} denotes a database where each transaction is a labelled rooted unordered tree. The task of frequent induced subtree mining from T_{db} is finding all induced subtrees that have minimum support s.

Definition 5 (*Support*): Support *s* of a tree *T*´ in database T_{db} is defined as the number of trees, *T* that has at least one occurrence of *T*´ as an induced subtree in its structure.

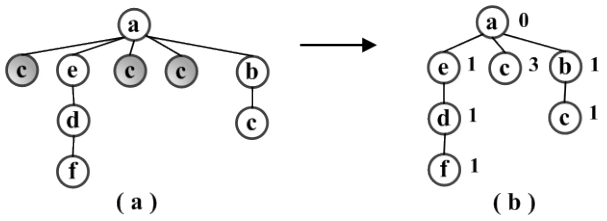


Figure 1: The highlighted nodes are the equivalent nodes (a) and the numerical values are the weights of the respective nodes (b), for simplicity only label is used to represent a node

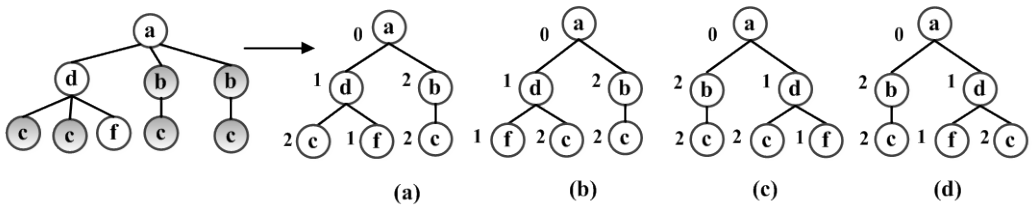


Figure 2: Four rooted ordered trees obtained from the same rooted unordered tree

3. OPTIMAL CANONICAL FORM

A canonical form (CF) of a tree is a representative form that can consistently represent many equivalent variations of that tree into one standard [90, 188]. The canonical forms for ordered and unordered subtrees are different. Due to having no sibling order, several ordered variations are possible from an unordered tree.

Definition 6 (*Equivalent Ordered Trees*): Two distinct ordered trees T_1 and T_2 are equivalent trees if they represent the same unordered tree *T*, denoted by $T_1 \cong T_2$.

An example of equivalent ordered trees is given in Figure 2, where four ordered trees can be derived from an unordered tree. We propose to represent these ordered variations by a single canonical form following an optimal traversal so that the same unordered tree is derived from each of them.

3.1 Balanced Optimal Canonical Form

We have earlier developed an optimal tree search traversal algorithm [155] by reducing the traversing problem to an optimisation problem called "simple assembly line balancing” [65]. Unlike existing traversal algorithms [188], our algorithm [155] works based on optimisation instead of fixing left-to-right order among siblings. We propose heuristics that are applied recursively for setting the rules of traversing the whole tree. Heuristic 1 identifies a potential node during the traversal process. Heuristics 2 and 3 select the best node if multiple nodes are identified as candidates for traversal. Induction of heuristics will result in the optimal traversal balanced.

Heuristic 1 *After traversing the root node, the enumeration of available nodes satisfying the ancestral relationship ($v_i \prec v_j$) will be prioritized based on their weights*.

Heuristic 2 *If there exist two or more nodes with maximum weight, the node with maximum number of children will get priority for traversing next.*

Heuristic 3 *In case of existence of multiple nodes with equal weight and children count, the minimum lexicographical order will be used to prioritize their traversing*.

Consider the example tree in Figure 1, following this traversal scheme, root node v_a will be traversed first. Next eligible nodes for traversing will be v_e, v_c, v_b as their parent node has been traversed. Node v_c will be chosen following heuristic 1. Heuristic 3 will need to be applied to choose between v_e and v_b, as the other two heuristics fail to prioritize the order. v_b will be traversed accordingly. Node v_e will be traversed next using heuristic 2. The final sequence for traversing the whole tree will be v_a, v_c, v_b, v_e, v_d, v_c, v_f, that is not restricted by depth-first or breadth-first order.

We propose a balanced-optimal canonical form for a tree represented in the optimal order obtained by this traversal. BOCF is a string representation of a tree along with four unique symbols, +1, -1, +2 and -2, that are used to represent the breadthwise movement from sibling to sibling and the depth-wise movement from a child to its parent. We use +1 and -1 for forward and backward travel towards depth, and +2 and -2 for forward and backward travel towards breadth respectively. We assume that none of these symbols are included in the alphabet of node labels.

Definition 7 (*BOCF String Representation of Unordered Tree*): The BOCF string representation of the rooted unordered tree is achieved by a guided record of sibling nodes. When a new node appears under its parent node, only the breadthwise movement from the existing rightmost sibling node is permitted.

Consider the trees in Figure 2. The optimal order of the equivalent trees in Figure 2 is: v_a, v_b, v_c, v_d, v_c, v_f. Using definition 7, the unique BOCF string representation of these four trees is 0v_a, +1, 2v_b, +1, 2v_c, -1, +2, 1v_d, +1, 2v_c, -2, 1v_f. It should be noted that all equivalent ordered trees is represented by a unique standard form. It indicates that they all are originated from the same unordered tree. This greatly benefits unordered tree mining. The optimal traversal poses a total order on all variants of an unordered tree which guarantees the uniqueness of BOCF for a labelled rooted *unordered* tree.

3.2 Dealing with the Isomorphism and Automorphism Problem

A main challenge in defining a canonical form for unordered trees is faced when two trees are found isomorphic. If a bijective mapping exists between the set of nodes of two trees T_1 and T_2, which preserves and reflects the tree structures, then these trees are called isomorphic to each other, denoted as $T_1 \cong T_2$. The term automorphism corresponds to isomorphism of a tree to itself. It is necessary to identify which of the ordered subtrees forms an automorphism group of an unordered subtree. During candidate generation, each subtree encoding should uniquely map to a single subtree only. Existing research addresses this problem by choosing one of the trees from the automorphism group as the representative of the group, and then all other isomorphic subtrees are ordered according to the representative of the automorphism group during candidate generation [90, 96]. This ensures that, for a particular unordered subtree, its occurrences are correctly counted so that the frequency can be easily determined. However, a checking is always required to find the presence of isomorphism in a tree. This causes an additional memory and time consumption for keeping the record of the representative tree and for doing isomorphism testing.

As shown earlier, the proposed BOCF encodes an unordered tree (including all of its ordered variants which are actually isomorphic to each other) uniquely. In other words, BOCF provides a unique representation to all isomorphic trees. This ensures that trees encoded with BOCF representation will be correctly grouped and counted. Unlike other canonical forms, BOCF does not require a record of representative trees or, an extra checking during candidate generation for dealing the isomorphism problem. Moreover, BOCF can naturally handle the automorphism problem. For applying the optimal traversal, the trees need to be pre-processed so that a concise tree representation can be derived by combining equivalent nodes. Consequently the weight of each node under its parent node is calculated. It is to be noted that the equivalent nodes (i.e. same labelled sibling nodes having the same child) should not be treated as distinct nodes. The order between them is not important, but, only the occurrences are important. This process allows us to avoid the isomorphism of a tree to itself, i.e. solving the automorphism problem. Consider the following example in Figure 3(a) where the dotted area is showing a case of automorphism problem for the considered tree. However, the BOCF representation is derived based on the weighted tree as shown in Figure 3(b) where automorphism can no longer exist.

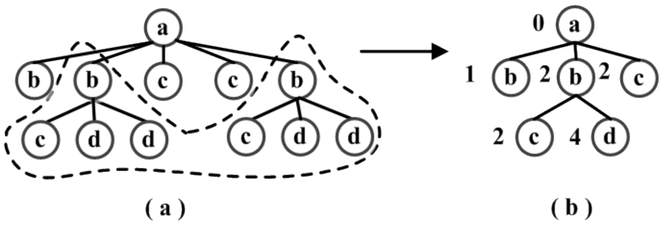


Figure 3: Automorphism problem

4. MINING FREQUENT LABELLED UNORDERED INDUCED SUBTREES

We define an enumeration tree that lists all induced unordered subtrees in T_{db} according to their BOCF strings. We used the right-path extension and join operations for growing the enumeration tree. Previous research has shown that the right-path extension produces a complete and non-redundant candidate generation [38, 90, 96]. The use of extension alone for growing enumeration tree can be inefficient because the number of potential growth may be very large, especially when the cardinality of the alphabet for node labels is large. This shortcoming necessitates of using a join operation [90, 96]. However, a join operation often generates invalid subtrees. We propose using a tree-structure guided schema for enumeration which allows the generation of valid subtrees only. In the proposed tree structure guided enumeration approach, the underlying level and fan-out information of nodes are utilised during candidate generation.

Operations on the Enumeration Tree: The basis of our enumeration tree is as follows. An unordered N-tree (i.e. a tree with N number of nodes) BOCF is formed from the unordered (N+1)-tree BOCF by removing the right-most path (i.e. the right-most node along with its edge) at the bottom level.

For growing the enumeration tree we define extension and join operations using the BOCF string and the tree-structure guided schema.

Definition 8 (*BOCF-extension*): For a node v_i (fan-out $\neq$ 0) of the BOCF T_1, extension is possible to apply using every frequent label v_j having level $Lv(T_1, v_i)$-1. This extension operation will result in a new BOCF T_2 in the enumeration tree where v_j will be the child of v_i. If T_1 is a N-tree BOCF, then the resultant new BOCF T_2 will be a (N+1)-tree with height $H(T_1)$ +1. Further extension is possible from this new right-most node v_j.

Before giving the definition of BOCF-join operation, we define equivalent groups.

Definition 9 (*Equivalent Group*): If two N-node trees T_1 and T_2 have height $H(T_1) = H(T_2)$ and share first N-1 node (along with labels and weights) in common, they are considered as equivalent group, denoted by $T_1 \cong T_2$.

Definition 10 (*BOCF-join*): Join operation is a guided extension between two BOCFs, T_1 and T_2, from an equivalent group, $T_1 \cong T_2$. Assume, v_i and v_j are the corresponding right-most node of T_1 and T_2, where $w_i > w_j$ or, $w_i = w_j$ with v_i lexicographically sorts lower than v_j. By joining v_j in T_1 at the position of $Lv(T_1, v_i)$-1 will result in a new (N+1) node BOCF, denoted by $T_1 \odot T_2$, of the same height as tree T_1.

Growth Rules: Candidate trees can have a large number of potential nodes to get a right-path extension. In order to restrict this growth, heuristics can be employed using BOCF definition. This will result in reduction of the number of candidates generated as well as in the reduction of the number of isomorphic subtrees. These rules support the basic formation principle of the enumeration tree, i.e. keeping the N-tree BOCF unchanged with the newly generated (N+1)-tree BOCF.

Rule1: *Among all the nodes at the bottom level, the node with the maximum weight will be chosen for BOCF-extension.*

Rule2: *If there are more than two maximum weighted nodes then the node with maximum children will be chosen for BOCF-extension.*

Rule3: *If more than two maximum weighted nodes with the same number of children exist then the node that sorts lexicographically lower will be chosen for BOCF-extension.*

Consider an example database in Figure 4(a). We compare our enumeration tree (Figure 4(b)) with the enumeration tree (Figure 4(c)) generated by following the HybridTreeMiner method [96] (abbreviated as HBT here). HBT also uses the right-path extension and join operations for growing the enumeration tree, but, these are defined using a different canonical form (BFCF) [90], whereas we use BOCF and the tree-structure guided schema for growing the enumeration tree. The dotted rectangles in (Figure 4(c)) are showing the generation of invalid subtrees in HBT. We did not show the full enumeration tree for HBT. If we continue it will grow in a much bigger size, resulting in much higher numbers of invalid subtrees. But, for our method, Figure 4(b) is the complete enumeration tree of the considered database.

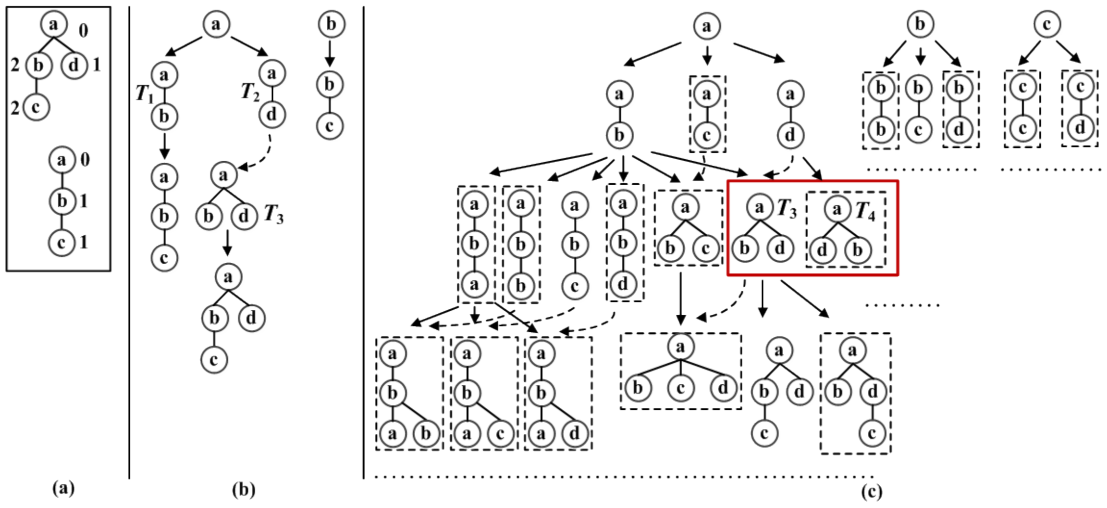


Figure 4: Comparison between the proposed and existing enumeration techniques considering minimum support 1 and the dotted rectangles indicate invalid subtrees

It can be clearly seen that our enumeration tree generates much less candidates in comparison to HBT enumeration tree because of producing only valid subtrees.

Generation of several invalid subtrees causes extra memory space and, then, pruning of these subtrees causes additional computational cost for HBT. Moreover, our enumeration approach is more robust to the isomorphism problem. In Figure 4(c) the enumeration tree produces two candidate trees T_3 and T_4, which are isomorphic. For counting the exact support these two should consider as same candidate. In that case an extra checking method is needed to count isomorphic trees; but our enumeration approach avoids growing any isomorphic tree. For example, in Figure 4(b); only tree T_3 exists, tree T_4 can't be generated. According to BOCF-join, join is supported only from T_1, "$0v_a$ +1 $2v_b$" to T_2, "$0v_a$ +1 1 v_d" as $w_b > w_d$.

BOSTER Algorithm

Input: a database T_{db} consisting of labelled rooted unordered trees represented as BOCF strings, a dictionary containing level and fan-out information of each node, a user defined minimum support (*min_sup*).

Output: All frequent induced subtrees.

```
1.  Result ←∅;
2.  F1 ← the set of all frequent nodes;
3.    for all $t_k$ ∈ F1 do
4.      if fan-out($t_k$) = 0
5.          continue
6.      end if
7.      Grow_Enum ($t_k$, level, weight, fan-out );
8.    end for
9.  return Result;
```

Grow_Enum (C_k, level, weight, fan-out)

```
1.  for all f ∈ $C_k$ do
2.      Select the right-most node of $C_k$ using Growth rules;
3.      Generate candidate $C_{k+1}$ by adding f;  //using BOCF-extension;
4.      if support ($C_{k+1}$) ≥ min_sup then
5.        Result ← Result ∪ $C_{k+1}$;
6.      end if
7.      Grow_Enum ($C_{k+1}$, level, weight, fan-out );
8.  end for
9.  for all $C_k'$ such that $C_k \cong C_k'$ do
10.     $C_{k+1} \leftarrow C_k \odot C_k'$; //using BOCF-join;
11.     if support ($C_{k+1}$) ≥ min_sup then
12.       Result ← Result ∪ $C_{k+1}$;
13.     end if
14.     Grow_Enum ($C_{k+1}$, level, weight, fan-out );
15. end for
```

Figure 5: High level pseudo code of BOSTER algorithm

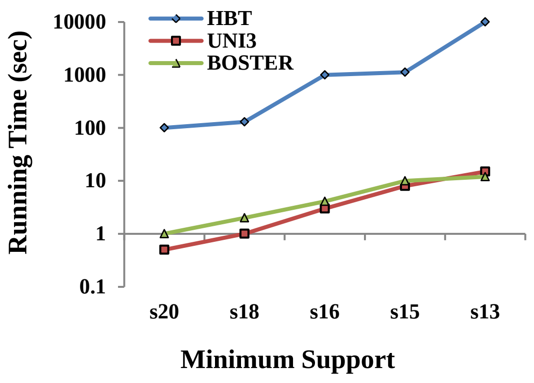
Figure 5 lists the pseudocode of the BOSTER algorithm. The process of frequent subtree mining is initiated by scanning the tree database, T_{db}, where trees are stored as BOCF strings along with weight, level and fan-out information of each node. The candidate generation method *Grow_Enum* is called recursively for growing the candidates. The frequency of every resultant candidate tree is computed according to the method used in [90, 96]. This is basically an apriori based frequency counting which gives us the exact frequent subtree list. In order to improve computational efficiency, we stop counting of a subtree as soon as the tree count reaches the minimum support value.

5. EXPERIMENTAL EVALUATION

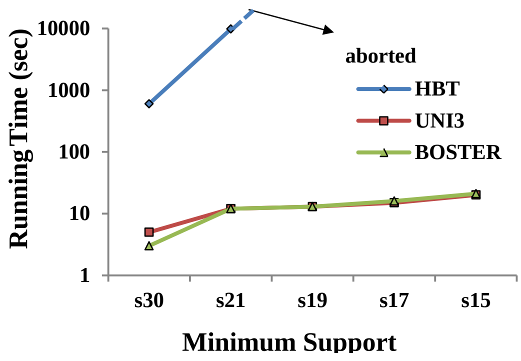
We have performed experiments to evaluate the efficiency of the proposed algorithm on real application data. All experiments have been conducted on a 2.8GHz Intel Core i7 PC with 8GB main memory and running the UNIX operating system. Two state-of-the-art unordered tree mining algorithms, HBT [96] and UNI3 [98] are used for benchmarking. We recorded the run time and memory usage of each algorithm and compared their performances.

In line with other research and to show scalability, three variations of the real weblog data, CSLOGS [38, 50], are used. (1) CSLOG1 - data generated from the first week web log usage consisting of 8,074 trees. (2) CSLOG12 - data generated from the first two weeks usage consisting of 13,934 trees. (3) CSLOGS - the entire data covering all weeks consisting of 59,691 trees, 716,263 nodes and 13,209 unique node labels.

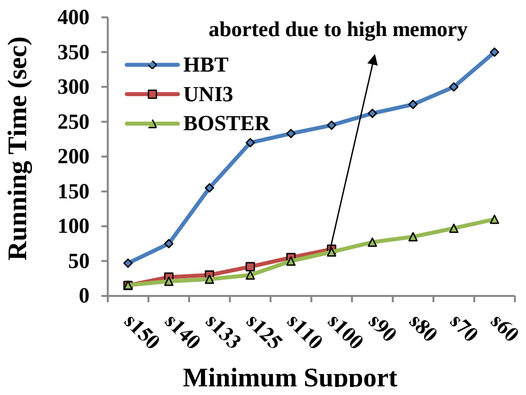
Figure 6(a, b, c) and Figure 7(a, b, c) compare the runtime and memory comparison of BOSTER against HBT and UNI3 respectively. For both runtime and memory comparison, BOSTER significantly outperforms HBT in all cases. However, UNI3 gave better memory consumption than BOSTER over CSLOG1 and CSLOG12. On the entire set of CSLOGS, BOSTER started to outperform UNI3 for support value less than 100. After this support value, UNI3 could not perform due to extensive memory usage (Figure 7(c)). We allocated about 15GB memory to run UNI3, but, it still failed to execute results. UNI3 includes a large number of extra data structure to hold intermittent information for the mining process. These additional structures cause the out of memory problem when mining the large data with small support values. Moreover, both HBT and UNI3 keep record of representative trees for performing an isomorphism test that causes additional time and memory expense, but BOSTER can avoid this extra cost using BOCF string representation.



(a)



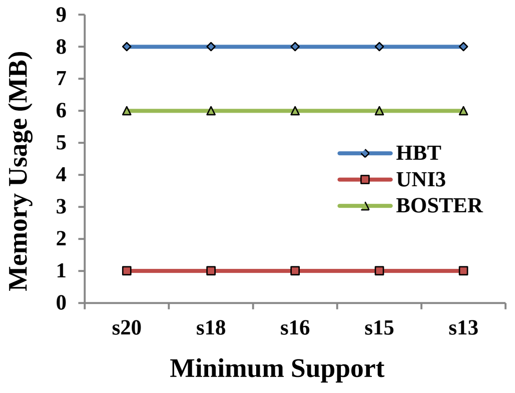
(b)



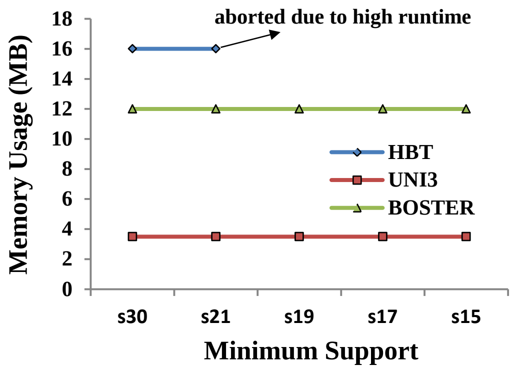
(c)

Figure 6: Runtime comparison over CSLOG1 (a), CSLOG12 (b), full CSLOGS (c)

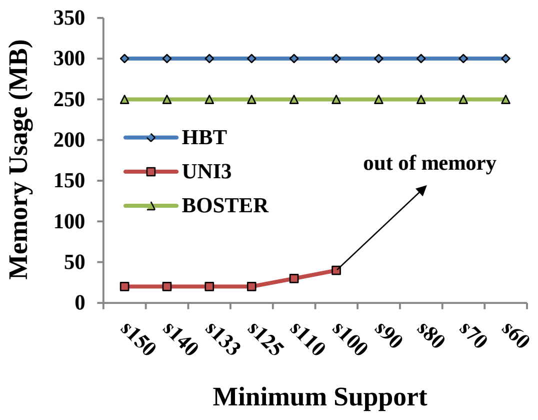
In real-life applications, memory usage can have a significant impact on the application's usability from the perspective of performance, interactivity, etc. BOSTER is able to consume less memory with yielding efficient time complexity, in comparison to the benchmarked algorithms, even in the presence of large data



(a)



(b)



(c)

Figure 7: Memory comparison over CSLOG1 (a), CSLOG12 (b), full CSLOGS (c)

6. DISCUSSION

In this paper, we presented a novel canonical form, and developed a new method of finding frequent induced subtrees from the dataset of labelled rooted unordered trees. We empirically evaluated the efficiency of the proposed algorithm, BOSTER, against the well-known algorithms in the literature, over real life datasets.

In future we will extend the proposed algorithm to find condensed representations like frequent closed patterns and we also will explore the scope for extending our canonical form to represent free trees in order to mine frequent patterns from them.

Paper 5: BEST: An Efficient Algorithm for Mining Frequent Unordered Embedded Subtrees

Israt Jahan Chowdhury* and Richi Nayak*****

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

PUBLISHED IN: Lecture Note in Computer Science: PRICAI 2014: Trends in Artificial Intelligence, 8862, pp. 459-471

Abstract[6]: This paper presents an algorithm for mining unordered embedded subtrees using the balanced-optimal-search canonical form (BOCF). A tree structure guided scheme based enumeration approach is defined using BOCF for systematically enumerating the valid subtrees only. Based on this canonical form and enumeration technique, the **b**alanced optimal search **e**mbedded **s**ub**t**ree mining algorithm (BEST) is introduced for mining embedded subtrees from a database of labelled rooted unordered trees. The extensive experiments on both synthetic and real datasets demonstrate the efficiency of BEST over the two state-of-the-art algorithms for mining embedded unordered subtrees, SLEUTH and U3.

Keywords: Frequent subtrees, labelled rooted unordered trees, embedded subtrees, canonical form, enumeration approach.

1. INTRODUCTION

The problem of finding frequent subtrees from the tree structured data has important applications in diverse areas including web mining, XML mining, computer vision, network routing and bioinformatics. From the tree structured data, frequent subtree mining discovers important patterns in the tree form showing the distinct features of the data. For example, in [50] frequent subtree mining is used in web log data to distinguish users according to their browsing behaviours on web. It also facilitates other data mining tasks such as association rule mining, classification and clustering.

The tree structured data is often represented in ordered form in which parent and siblings relationships (i.e., fixed left-to-right order) are preserved. However, in practice, the ordering among siblings is not always of great importance to users and is not always available [126]. Unordered trees have shown the capability of identifying interesting relations due to not being constrained by sibling conditions [29, 35]. This distinct property of unordered trees, however, makes the process of mining frequent subtrees more challenging in comparison to ordered trees. A huge number of candidate generation occurs where subtrees with similar structure are included. Besides, it is non-trivial to determine the “good” growth strategy and avoid

[6] D.-N. Pham and S.-B. Park (Eds.): PRICAI 2014, LNAI 8862, pp. 459–471, 2014.
© Springer International Publishing Switzerland 2014

redundancy, as there can be many possible ways to extend an existing pattern in a tree format, due to not having an order constraint in sibling nodes. Moreover, high computational and memory expense are an ongoing issue for mining tree structured data.

Two possible types of subtrees, Induced and Embedded, can be mined from the tree data, preserving parental and ancestral relationships respectively. Mining embedded subtree can be seen as a generalisation task of mining induced subtree that is essential to mine interesting relational information inherent within deeply embedded data objects in the tree database. It is a more difficult problem than induced subtree mining as it requires examining several levels within a tree to identify an embedded subtree [70].

In this paper we present an algorithm for mining unordered embedded subtrees. Distinct from existing tree traversal methods [188], we have previously proposed an optimal tree traversal algorithm for traversing a rooted unordered tree without enforcing an order among sibling nodes [155]. We extended this traversing algorithm by introducing a new heuristic that leads towards a new definition of canonical form for representing unordered trees, called the balanced-optimal canonical form (BOCF) [57]. The BOCF is able to represent unordered trees uniquely even in the presence of isomorphism.

In this paper we study some properties of the BOCF and design an optimal enumeration tree using BOCF that systematically enumerates all frequent embedded subtrees based on the tree structure guided scheme. This enumeration approach is efficient as it restricts the search by only generating the unambiguous and valid subtrees using the underlying tree structure information. For growing the enumeration tree as well as generating candidates, we define extension and join operations. Finally, the **b**alanced optimal search **e**mbedded **s**ub**t**ree miner algorithm (BEST) is proposed for mining embedded subtrees from a database of labelled rooted unordered trees. Empirical analysis carried out using both real and synthetic data has shown the effectiveness of BEST over the two state-of-the-art algorithms, SLEUTH [70] and U3 [97].

2. RELATED WORKS

For finding unordered frequent tree patterns, most of the proposed algorithms use a canonical form and extend only candidates that are in the canonical form. A sorted pre-order string canonical form that can be obtained in linear time was first defined by [94]. A few more similar canonical representations based on depth-first traversal and breadth-first traversal have been defined [90-92]. The proposed method BEST uses the optimal traversal based canonical form (BOCF) that is robust to isomorphism problem due to its order independence and use of optimisation. Using BOCF, we proposed a tree structure guided scheme based enumeration technique that uses both right-path extension and join to grow for mining unordered embedded subtrees. None of the above state-of-the-art methods used similar structure guided enumeration process. HybridTreeMiner uses extension and join operations for growing the enumeration tree like BEST using the BFCF canonical form, but for mining induced subtrees. Whereas, SLEUTH [70] is designed to mine embedded subtrees and also uses extension and join operations for growing the enumeration trees but the join is scope-list join via the descendant and cousin tests. More recent methods UNI3 [98] and U3 [97] also proposed a tree model guided enumeration where they used embedded level information, but we incorporated much more tree information including level, fan-out and a new tree parameter called weight for proposing the tree structure guided enumeration. Moreover they used only right path extension for growing the enumeration tree and used depth-first traversal based string representation which requires additional processing for tackling isomorphism. The unordered embedded subtrees [142, 194] mining algorithm, Treefinder, can miss some patterns especially for a lower support and others have been designed for mining maximal embedded subtrees [142, 194].

3. MINING EMBEDDED FREQUENT SUBTREES

We present the balanced-optimal canonical form, BOCF. We describe the tree structure guided scheme based enumeration approach and the proposed BEST algorithm.

3.1 Preliminaries

Unless otherwise stated, all trees considered in the paper are rooted, labelled, and unordered. Let $T = (V, E, L)$ be a *rooted labeled unordered* tree, where $V = \{v_0,$

$v_1, v_2, \ldots, v_n\}$ denotes the set of nodes with v_0 as *root* node, $E = \{(v_i, v_j)| v_i, v_j \in V\} = \{e_1, e_2, \ldots, e_{n\text{-}1}\}$ denotes the set of edges and L denotes the set of labels. The label is given by a function Φ: $V \rightarrow L$ which maps nodes with unique labels. The size of a tree is denoted as $|T|$ which is the number of nodes $|V|$. An unordered tree has no ordering relationship among the nodes except ancestor-descendent or parent-child. The ancestor-descendent relationship between two nodes is denoted by $v_i \prec v_j$, i.e., v_i is ancestor of $v_{j,}$ the '$\prec$' symbol represents 'precedes'. The level of a node v_i in a tree T is denoted as $Lv(T, v_i)$ and the height of a tree T is denoted as $H(T)$.

Definition 1 (*Embedded Subtrees*): A tree $T'(V', L', E')$ is an unordered *embedded subtree* of a tree T (V, L, E) iff: (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) $L' \subseteq L$ and the labelling of V' in T is preserved in T' (4) $\forall v_i' \in V'$, $\forall$ $v_i \in V$ and v_i' is not the root node, then ancestor of v_i'= ancestor of v_i, and (5) no left-to-right ordering among the siblings in T is preserved among the corresponding nodes in T'.

Definition 2 (*Equivalent Node*): In a rooted labelled unordered tree T, if two nodes v_i and v_j have the same label ($lab_i = lab_j$ & lab_i, $lab_j \in L$), originated from the same labelled parent node (parent of v_i = parent of v_j) and has the same labelled child nodes then they are called *equivalent nodes*, denoted by $v_i \cong v_j$.

Definition 3 (*Weight of Node*): *Weight* of a node v_i ($v_i \neq v_0$) is defined as the total number of its equivalent node. For tree T, weight of node v_i is w_i such that w_i = total number of equivalent nodes of v_i.

Definition 4 (*Mining Unordered Embedded Subtree*): Let T_{db} is a database, where each transaction is a labelled rooted unordered tree. The task of mining frequent unordered embedded subtree from T_{db} is finding all embedded subtrees that have minimum support s.

Definition 5 (*Support*): Support s of a tree T' in T_{db} is defined as the number of trees, T that has at least one occurrence of T' as an embedded subtree in its structure.

3.2 Balanced Optimal Canonical Form (BOCF)

We first describe the balanced optimal canonical form (BOCF) for a rooted ordered tree [57, 155]. A canonical form (CF) of a tree is a representative form that can consistently represent many equivalent variations of that tree into one standard

[90, 188]. The canonical forms for ordered and unordered subtrees are different. A main difference is the possibility of having several subtrees showing different orders between sibling nodes, even though, the information contained within the structure remains essentially the same. Several ordered variations can be formed from a unique unordered tree. This leads us to define Equivalent ordered trees [57].

Definition 6 (*Equivalent Ordered Trees*): Two distinct ordered trees T_1 and T_2 are equivalent to each other if they represent same unordered tree T, denoted by $T_1 \cong T_2$.

An example of equivalent ordered trees is given in Figure 1, where four rooted ordered trees can be derived from a rooted unordered tree. We propose to represent these ordered variations by a single canonical form following the optimal tree traversing so that the same unordered tree is derived from each of them.

The canonical form, BOCF is defined by using the order obtained by traversing the tree optimally [155]. BOCF is a string representation of a tree that records label of each node along with its weight following the optimal order [57, 155]. This string also includes four unique symbols, +1, -1, +2 and -2, to represent the breadthwise movement from sibling to sibling and depth-wise movement from a child to its parent. The symbols +1 and -1 are used for depth-forward and depth-backward travel respectively. The symbols +2 and -2 are used for breadth-forward and breadth-backward travel respectively. It is assumed that the alphabet of node labels includes none of these symbols.

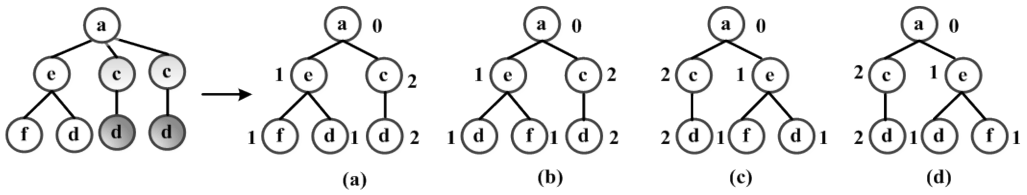


Figure 1: Four rooted ordered trees obtained from the same rooted unordered tree. Different equivalent nodes are shown as highlighted; weights of nodes are calculated accordingly

An Example: In Figure 1 the string encoding using BOCF of the four ordered trees are (a) "0v_a, +1, 2v_c, +1, 2v_d, -1, -2, 1v_e, +1, 1v_d, -2, 1v_f"; (b) "0v_a, +1, 2v_c, +1, 2v_d, -1, -2, 1v_e, +1, 1v_d, +2, 1v_f"; (c) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, -2, 1v_f"; (d) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, +2, 1v_f".

We prove that there exists a one-to-one correspondence between a labelled rooted *ordered* tree and its BOCF.

Lemma 1: *Each labelled rooted ordered tree corresponds to a unique balanced optimal canonical form. Each valid balanced optimal canonical form corresponds to a unique labelled rooted ordered tree.*

PROOF: Since the traversing path of a tree is determined using an optimisation model, each ordered tree from an equivalent group for that unordered tree actually represents the same network. Consequently, the optimal traversal gives the same traversing order to all equivalent ordered trees. BOCF is defined using this optimal order along with some unique symbols to capture the sibling constraints for the different ordered trees. As a result, each labelled rooted ordered tree will be represented by a unique BOCF.

The second statement of the aforementioned lemma is proved by the induction on the number of nodes N in a labelled rooted ordered tree. For the base case, when $N = 1$, the valid string representation of BOCF is of the form $0lab_i$, where lab_i ($lab_i \in L$) is the label of the single node v_i; weight 0 indicates a root node. In this case, the corresponding labelled rooted unordered tree is a single node, which is unique.

For simplicity of this proof we group all unique symbols of representing the sibling constraints; let C be the group containing all the unique symbols for representing constraints where $C \notin L$ and $\{-1, +1, -2, +2\} \in C$. So incorporating this notation the string representation, S of BOCF can be represented as "S = "w_0, lab_0, C, w_i, lab_i, …". For the induction step, we assume that, for each BOCF string representation S_n with $N = n$ nodes, there is a unique labelled rooted ordered tree in corresponding to it. A valid BOCF string representation S_{n+1} with $N = n + 1$ nodes is of the form "S_n . . . C, w_{n+1}, lab_{n+1}". S_n determines a unique labelled rooted ordered tree with n nodes. In addition, the last node (with label lab_{n+1}) becomes the rightmost child of node n. As a result, the labelled rooted ordered tree N_{n+1} corresponding to S_{n+1} is determined uniquely.

Consider the example in Figure 1, for a rooted unordered tree, different rooted ordered trees and the corresponding BOCFs are obtained by assigning different orders among the children of internal nodes. The BOCFs of equivalent ordered trees only vary in terms of breadth movement, which shows the order of siblings for different trees that can be ignored for portraying the unordered tree. The BOCF string representation of the rooted unordered tree is defined by a guided breadthwise movement while forming the string of ordered trees. The rest of the ordering that reflect ancestor descendent relationship is kept unchanged.

Definition 7 (*BOCF String Representation of Unordered Tree*): The BOCF string representation of the rooted unordered tree is achieved by a guided record of sibling node. When a new node is recorded under its parent node, only the breadthwise movement from the existing rightmost sibling node is permitted.

By doing so, all equivalent ordered trees will be represent by a unique standard form, which will be advantageous for unordered tree mining. Consider again the example of Figure 1, using definition 7 the string representation of all four equivalent ordered trees are: (a) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, +2, 1v_f"; (b) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, +2, 1v_f"; (c) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, +2, 1v_f"; (d) "0v_a, +1, 2v_c, +1, 2v_d, -1, +2, 1v_e, +1, 1v_d, +2, 1v_f", which are same and represent the fact that they are originated from the same unordered tree.

Lemma 2: *The BOCF construction procedure for unordered trees has time complexity O* (|*T*| *log* |*T*|).

PROOF: The optimal traversal algorithm gives $O(|T| \log |T|)$ time complexity where $|T|$ is the number of nodes in a tree. Implementing any of the three heuristics [57] of optimal traversal for sorting nodes will give a possible time complexity of $O(|T| \log |T|)$. Assuming there are $|T_j|$ nodes in recursion j of the tree traversal for $j = 1, 2, \ldots n$, it will take $O(|T_j| \log |T_j|)$ comparisons to sort nodes at recursion *j*. The total number of comparisons for normalising the whole tree is $\sum_j O(|T_j| \log |T_j|)$, which is $O(|T| \log |T|)$ (note that $\sum_j (|T_j| \log |T_j|) \le \sum_j (|T_j| \log |T|) = |T| \log |T|$). BOCF is driven using the exact ordering of optimal traversal, therefore its construction complexity is also $O(|T| \log |T|)$.

It can be noted that all equivalent ordered trees is represented by a unique standard form and indicate that they are originated from the same unordered tree. This greatly benefits unordered tree mining. The optimal traversal poses a total order on all variants of the same unordered tree which guarantees the uniqueness of BOCF for a labelled rooted *unordered* tree.

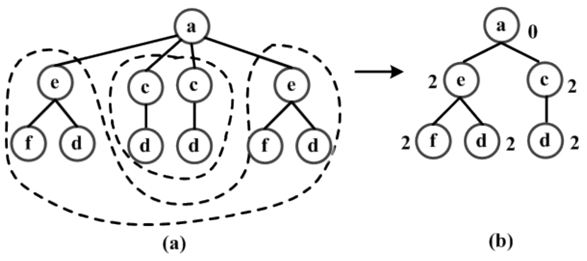


Figure 2: Automorphism problem

Handling the Isomorphism and Automorphism Problems: Two trees T_1 and T_2 are isomorphic to each other if a bijective mapping exists between their sets of nodes, which preserves and reflects their structures, denoted as $T_1 \cong T_2$. If isomorphism exists within a tree, then it is called automorphism. It is necessary to identify which of the ordered subtrees belongs to an automorphism group of an unordered subtree in order to ensure the exact count of its occurrences as well as the frequency. Therefore, canonical form should be defined in a way that will uniquely map each subtree to a single subtree during candidate generation. Existing research addresses this problem by choosing one of the trees from the automorphism group as the representative of the group, and then all other isomorphic subtrees are ordered according to the representative of the automorphism group during candidate generation [70]. However, a checking is always required to find the presence of isomorphism in a tree, which causes additional memory consumption for keeping the record of the representative tree during the candidate generation phase, thus, the exact ordering can be followed for generating other isomorphic subtrees.

Proposed BOCF addresses this problem [57] as follows. It gives a unique representation to all isomorphic trees without requiring any representative tree record or, any extra checking during candidate generation. Moreover, it naturally handles the automorphism problem by using the concept of weights (Definition 3) to represent equivalent nodes (Definition 2). The equivalent nodes for an unordered tree should not be treated distinctively since their occurrences are important for mining, not the inherent ordering between sibling nodes. Consider the following example where the dotted area shows a case of automorphism problem for the considered tree.

The proposed canonical form is derived based on the weighted tree as shown in Figure 2 where automorphism can no longer exist.

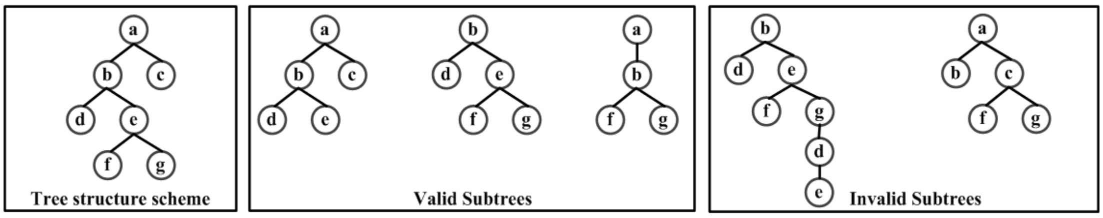


Figure 3: Valid and invalid subtrees following tree structure guided scheme

3.3 The Enumeration Tree

In this section we define an enumeration tree that enumerates all embedded unordered subtrees in T_{db} according to their BOCFs. We used both right-path extension and join operation for growing the enumeration tree. Previous research has shown that the right-path extension produces a complete and non-redundant candidate generation [195]. Due to the large number of potential growth, only using extension for growing an enumeration tree can be inefficient, especially when the cardinality of the alphabet for node labels is large [70, 90]. This emphasises the need of using a join operation; however, it often generates invalid subtrees. Since we use a tree structure guided scheme for enumeration, this generates valid subtrees only.

Tree Structure Guided Scheme Based-Enumeration: This enumeration is a bottom-up approach that generates non-redundant candidates [55]. A candidate generation technique can generate valid frequent and infrequent candidates as well as invalid frequent and infrequent candidates. It is desirable to enumerate valid frequent subtrees only to save memory and computational expense, instead of generating all possible candidates and prune invalid and infrequent subtrees later.

To illustrate this, we show a simple tree structure as an example database in Figure 3. We also show some possible valid and invalid subtrees that can be generated from this example tree. The subtree that does not follow the available tree structure information (i.e., the position of various nodes at various levels, ancestor-descendent or parent-child relationship, number of child nodes under parent node, etc.) is considered invalid. In our proposed tree structure guided scheme based enumeration, we utilise underlying level and fan-out information of nodes during candidate generation to make the approach structure guided. For efficiently growing the enumeration tree we define the extension and join operations using BOCF and the tree structure guided scheme.

Definition 8 (*Extension*): From a node v_i (fan-out $\neq 0$) of the BOCF tree T_1, extension is possible by adding a frequent label v_j having a level > $Lv(T_1, v_i)$. This will result in a new BOCF tree T_2 in the enumeration tree where v_j will be the child of v_i. If T_1 is a N-tree then the resultant new BOCF tree T_2 will be a (N+1)-tree with a height $H(T_1)+1$. Further extension will be possible from this newly added right-most node v_j.

Before giving the definition of join operation, we define equivalent groups.

Definition 9 (*Equivalent Group*): If two N-node trees T_1 and T_2 have height $H(T_1) = H(T_2)$ and have the first N-1 nodes (along with labels and weights) common, they are considered as equivalent group, denoted by $T_1 \cong T_2$.

Definition 10 (*Join*): Join operation is a guided extension between two BOCF trees T_1 and T_2 from an equivalent group, $T_1 \cong T_2$. Assume v_i and v_j are the corresponding right-most nodes of T_1 and T_2 respectively, where $w_i > w_j$ or $w_i = w_j$ with v_i lexicographically sorts lower than v_j. By joining v_j in T_1 at the position of $Lv(T_1, v_i)$-1 will result in a new (N+1) node BOCF tree, denoted $T_1 \odot T_2$, of the same height as BOCF tree T_1.

Growth Rules: Candidate trees can have a large number of potential nodes to get a right-path extension. In order to restrict this growth, heuristics can be employed. This will result in reduction of the number of candidates generated as well as in the reduction of the number of isomorphic subtrees. These rules support the basic formation principle of the enumeration tree, i.e., keeping the N-tree BOCF unchanged with the newly generated N+1- tree BOCF.

Rule1: *Among all the nodes at the bottom level, the node that has the maximum weight will be chosen for applying an extension.*

Rule2: *If there are more than two maximum weighted nodes then the node that has the maximum children will be chosen for applying an extension.*

Rule3: *If more than two maximum weighted nodes exist with the same number of children then the node that appears lexicographically lower will be chosen for applying an extension.*

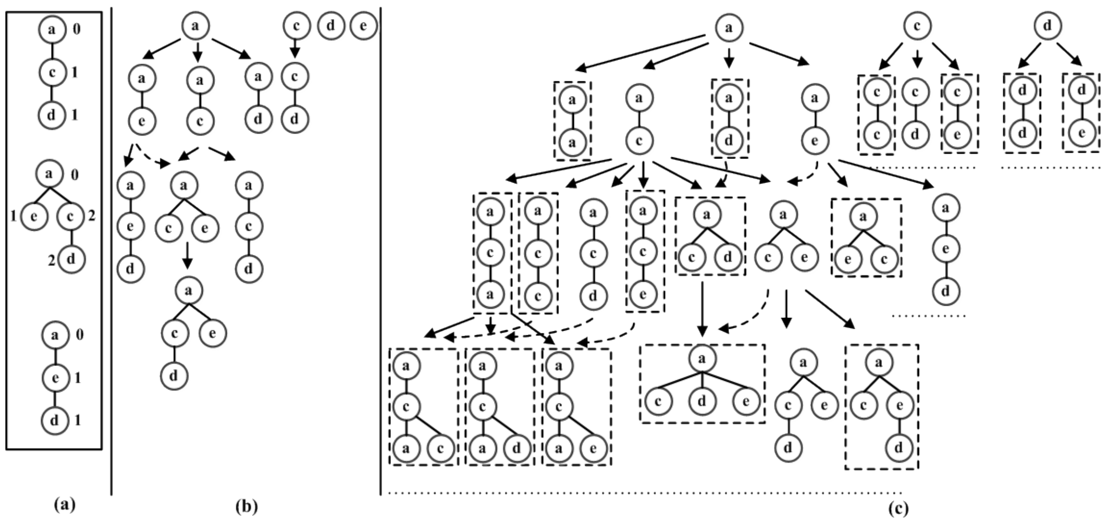


Figure 4: Comparison between the proposed and an existing enumeration technique considering minimum support 1 and the dotted rectangles indicate invalid subtrees

An Example: We compare the enumeration tree generated by BEST with another enumeration tree generated by SLEUTH [70] using an example database in Figure 4(a). Considering all labelled nodes as frequent, the SLEUTH enumeration tree grows as Figure 4(c), where the extension and join operations are defined using another canonical form (Figure 4(c)) and are not following tree structure guided scheme. In Figure 4(b), the proposed BOCF and the tree structure guided scheme based BEST enumeration tree is shown, which is the complete enumeration tree for the given database, whereas the state-of-the-art enumeration tree cannot be completed due to limited space. If we continue, it will grow more. The dotted rectangles in Figure 4(c) show an example of generated invalid subtrees in SLEUTH. Figure 4(c) only shows some, a lot more is generated during the process, whereas no invalid subtree is generated by BEST. It can be noted that the BEST enumeration tree generates much less candidate trees in comparison to SLEUTH because the former only produces valid subtrees. Consequently, a lot of memory space and additional computational time can be saved that will be required to prune these invalid subtrees afterwards. Empirical analysis ascertains these claims.

BEST Algorithm

Input: a database T_{db} consisting of labelled rooted unordered trees in their BOCFs, a dictionary containing level and fan-out information of each node, a user defined minimum support (*min_sup*).
Output: All frequent embedded subtrees.

```
1.  Result ← ∅;
2.  F1 ← the set of all frequent nodes;
3.  F2 ← ∅;
4.  while  F1 ≠ ∅ do
5.      for all $t_k$ ∈ F1 do
6.          if fan-out($t_k$) = 0
7.            continue
8.          end if
9.          Ext_can ← Enum ($t_k$, level, weight, fan-out );
10.         for all $t_{k+1}$ ∈ Ext_can do
11.             if support ($t_{k+1}$) ≥ min_sup then
12.                 F2 ← F2 ∪ $t_{k+1}$;
13.             end if
14.         end for
15.     end for
16.         F1← F2;
17.         Result ← Result ∪ F1;
18.         F2 ← ∅;
19. end while
20. return  Result
```

Enum

Input: candidate C_k, level, weight, fan-out
Output: all (k+1) extensions of C_k

```
1.  out ←∅;
2.  for all frequent label f do
3.      Select the right-most node of $C_k$ using Growth rules;
4.      Generate candidate $C_{k+1}$ by adding f;   //using definition 8;
5.      out ← out ∪ $C_{k+1}$;
6.  end for
7.  for all $C_k'$ such that $C_k \cong C_k'$ do
8.      $C_{k+1} \leftarrow C_k \odot C_k'$;    //using definition 10;
9.      out ← out ∪ $C_{k+1}$;
10. end for
11. return out;
```

Figure 5: High level pseudo code of BEST algorithm

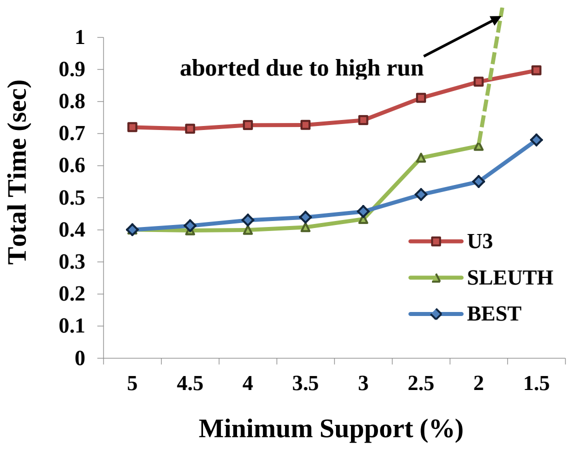
3.4 The BEST Algorithm

The overall BEST algorithm is presented in Figure 5. The process of frequent subtree mining is initiated by scanning a database, T_{db}, where trees are stored as BOCF strings along with weight, level and fan-out information of each node. The set of frequent labels (frequent subtrees of size 1) is generated and larger sized subtrees are generated by calling the *Enum* function recursively. In *Enum* (Figure 5), a subtree is extended if the right-most node of the tree supports any of the three rules of growing strategy. For implementing extension, the level difference of the right-most node of the considered tree is checked with the frequent label and the new candidate subtree is generated if the condition is met. Frequency of every resultant candidate tree is computed according to the method used in [90]. This is an apriori based frequency counting which gives us the exact frequent subtree list. In order to improve computational efficiency, we stop counting of a subtree as soon as the tree count reaches the minimum support value. Throughout the BEST algorithm the downward-closure lemma [140] is hold; each *N*-subtree of a frequent *N*+1-subtree has to be frequent. In the *Enum* function, we also used join for generating candidates from *equivalent groups* that support the join operation and the frequency of each subtree is calculated for further processing.

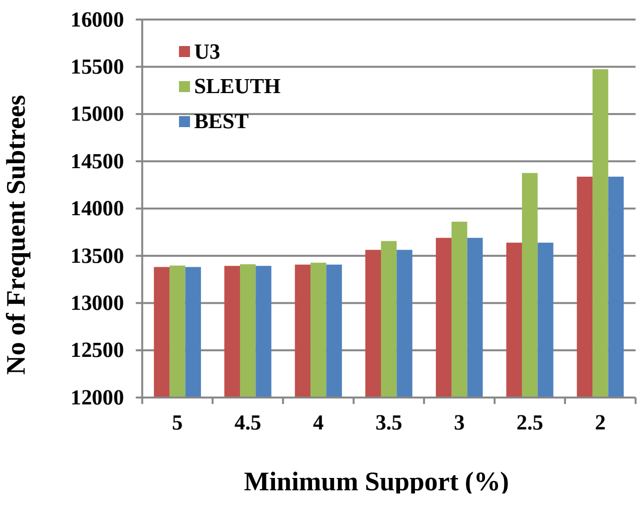
4. EXPERIMENTAL EVALUATION

We have performed extensive experiments to evaluate the efficiency of the proposed BEST algorithm on real application data as well as on synthetic data. All experiments have been conducted on a 2.8GHz Intel Core i7 PC with 8GB main memory and running the UNIX operating system. SLEUTH [70] and U3 [97], used for benchmarking, are designed for mining unordered embedded subtrees and are most relevant to our proposed method.

Performance on Real Application Data - CSLOGS: In our experiments**,** we used the CSLOGS dataset a real weblog data that consists of 59,691 trees, 716,263 nodes and 13,209 unique node labels [70, 195]. This data set has been largely used to evaluate various frequent subtree mining algorithms [70, 97].



(a)



(b)

Figure 6: Comparison over CSLOGS data based on runtime (a) and no of frequent subtrees (b)

For evaluating the performance we consider the runtime and candidate generation for all three algorithms. For CSLOGS dataset, BEST consistently outperformed SLEUTH and U3 (Figure 6(a)). Although SLEUTH performs almost same as BEST, but after a certain value of minimum support (1.5%) it took longer time than the other two algorithms. For SLEUTH the number of candidate subtrees is higher than the other two algorithms, i.e., it includes a lot of invalid subtrees during enumeration, therefore, spends more time on candidate generation and pruning afterwards. Besides, both SLEUTH and U3 require a canonical form test to avoid isomorphism and take longer processing time than BEST.

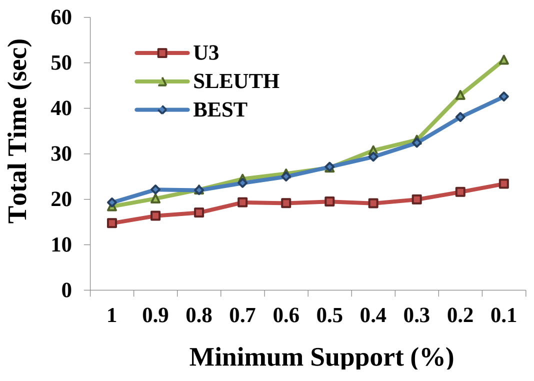
From Figure 6(b), it can be observed that SLEUTH generated more frequent subtrees in comparison to BEST, as it uses the opportunistic pruning technique which does not fulfil the downward closure lemma and may generate pseudo frequent subtrees [55].

Performance on Synthetic Data: Zaki's tree generator [38] is used for generating a synthetic data using following parameters: the number of labels N = 100, the number of vertices in the master tree M = 10,000, the maximum depth D = 10, the maximum fan-out F = 10 and the total number of subtrees T = 100,000. We used three synthetic datasets: D10 had all default values, F5 had all values set to default except for fan-out F = 5, and for T1M we set T = 1,000,000, with remaining default values. These are used for doing scalability and sensitivity analysis.

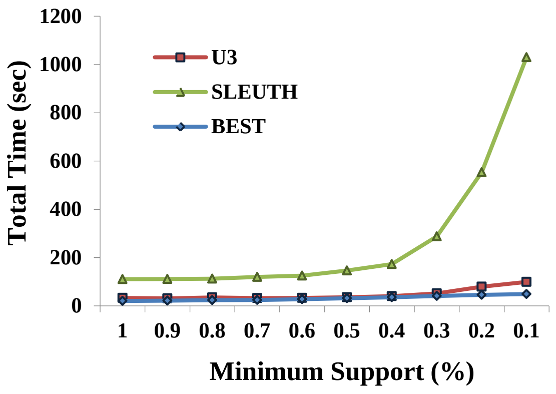
In Figure 7(a) for D10 dataset, U3 performed better than the other two, but the results for U3 are reported here for level difference one, otherwise the algorithm was aborted due to very high memory expense. As we restricted the level difference value to one, so the list of embedding subtrees is not completed and accordingly required less time, whereas both SLEUTH and BEST retrieved all of the embedding subtrees within reasonable time and memory expense.

For F5 dataset, we can see in Figure 7(b) BEST outperformed both SLEUTH and U3. Here U3 results are again reported based on restricted level difference, still BEST performed slightly better. Finally for T1M dataset we can see again BEST performed a little better than SLEUTH for lower and higher support values. Again, we only managed to run U3 for extracting embedded subtrees for level difference = 1, hence, it is not reporting the real time for extracting all embedded subtrees.

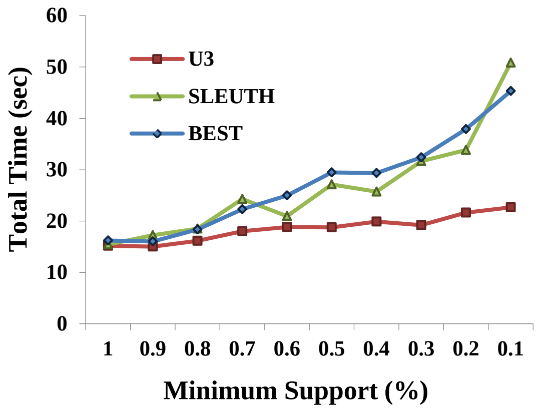
From these results we notice that both SLEUTH and U3 are sensitive to breadth, for small breadth value (small tree width), these baseline algorithms took high run time, as shown by F5 dataset (the fan-out number is less than D10 and T1M datasets). When SLEUTH and U3 performed over F5, the runtime increased about 8 and 2 times respectively in comparison to runtime over D10 and T1M. BEST seems not sensitive to this parameter and gives a consistent performance. It can be ascertain that BEST is a robust and efficient algorithm in comparison to existing state-of-the-art algorithms for mining embedded subtrees. It can tackle isomorphism using BOCF canonical form and generates only valid subtrees using the tree structure guided enumeration. These allow BEST to save reasonable amount of time and memory.



(a)



(b)



(c)

Figure 7: Comparison over $D10$ (a), $F5$ (b) and $T1M$ (c) synthetic datasets

5. CONCLUSION

In this paper, we presented a novel method for finding frequent embedded subtrees, using an optimal canonical form, from the dataset of labelled rooted unordered trees. We empirically evaluated the efficiency of the proposed method and benchmarked with the well-known algorithms in the literature, over both real and synthetic datasets.

Although finding the condensed representations of frequent patterns has found more interest in recent years, developing efficient algorithms for finding frequent patterns is still important. The efficiency of the algorithms for finding condensed representations depends on the efficiency of the base, i.e., frequent pattern mining algorithms. In future we will extend the proposed algorithm to find condensed representations.

Paper 6: FreeS: An Efficient Algorithm for Mining Frequent Unordered Embedded Subtrees

Israt Jahan Chowdhury* and Richi Nayak*

*School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO BOX 2434, Brisbane, Australia

PUBLISHED IN: Lecture Note in Computer Science: Web Information Systems Engineering – WISE 2015 (Accepted)

Abstract[7]: Web data can often be represented in free tree form; however, free tree mining methods seldom exist. In this paper, a computationally fast algorithm *FreeS* is presented to discover all frequently occurring free subtrees in a database of labelled free trees. *FreeS* is designed using an optimal canonical form, BOCF that can uniquely represent free trees even during the presence of isomorphism. To avoid enumeration of false positive candidates, it utilises the enumeration approach based on a tree-structure guided scheme. This paper presents lemmas that introduce conditions to conform the generation of free tree candidates during enumeration. Empirical study using both real and synthetic datasets shows that *FreeS* is scalable and significantly outperforms (i.e. few orders of magnitude faster than) the state-of-the-art frequent free tree mining algorithms, *HybridTreeMiner* and *FreeTreeMiner*.

Keywords: Web data, free tree, canonical form, enumeration approach

1. INTRODUCTION

In the Web domain, graphs and trees are commonly used data structures for modelling information with complex relations. Free trees - the connected, acyclic and undirected graphs - have become popular for presenting such data due to having unique properties [54, 63, 64, 96]. For obtaining useful structural information, free tree mining provides a good compromise between the more expressive but computationally harder general graph mining and the less expressive but faster sequence mining. As a middle ground between these two extremes, free trees have been widely used for representing and mining data in diverse areas including web, bioinformatics, computer vision and networks. For example, in analysis of molecular evolution, an evolutionary free tree, called phylogeny, can describe the evolution history of certain species [196]. In bioinformatics various useful patterns can be treated as free trees during pattern mining [54]. In computer networking, multicast free trees have been mined and used for packet routing [197]. Web access trees treated as free trees give interesting insight about the browsing behaviour since they do not take the point of entry into consideration [49].

[7] J. Wang et al. (Eds.): WISE 2015, LNAI 9418, pp. 123–137, 2015.
© Springer International Publishing Switzerland 2015

The process of finding frequent subtrees incurs high cost due to the inclusion of expensive but unavoidable steps like frequency counting and candidate subtrees generation. Frequency counting step often requires subtree isomorphism checking which is computationally hard, even known as NP-complete problem in graph mining algorithms [54]. Exponential and redundant candidate generation is another problem. During candidate generation, determining a “good” growth strategy is critical as there can be many possible ways to extend a candidate subtree. These problems become worse in free trees, due to being less-constrained structurally, in comparison to other tree forms such as ordered and unordered. With these complexities involved, only a few free tree mining algorithms are available in the literature. Chi et al. developed an apriori-like algorithm *FreeTreeMiner* [90] as well as an enumeration tree based algorithm *HybridTreeMiner* [96] to discover frequent free subtrees in a database of free trees. Rückert et al. [54] and Zhao et al. [64] have proposed algorithms for mining frequent free trees from a graph database. These algorithms generate large number of false positives (i.e., invalid candidate subtrees) during enumeration that need to be pruned in the frequency counting step. This causes high processing time. Moreover, the necessity of performing isomorphism checking to avoid redundant candidate tree generation and false frequency counting causes additional computational complexity.

In this paper, we propose an algorithm, *FreeS* which is a fast and accurate method for mining frequent free induced subtrees in a database of labelled free trees. First, we propose a unique representation of free trees by introducing a new order-independent *balanced optimal canonical form* (BOCF) that can effectively handle the subtree isomorphism problem. We introduce conditions to conform free tree candidate generation in their BOCFs for which the necessary proofs are also provided. Second, we propose a *tree-structure guided scheme based enumeration* approach that only generates valid candidate subtrees. To the best of our knowledge, *FreeS* is the first algorithm that uses the underlying tree-structure information to avoid invalid subtree generation while mining frequent free subtrees. Because of using the optimal canonical form and tree-structure guided scheme based enumeration, *FreeS* does fast processing. Our experiments with both synthetic and real-life datasets confirm that *FreeS* is faster by few orders of magnitude than two leading free tree mining algorithms, *HybridTreeMiner* and *FreeTreeMiner* (abbreviated as HBT and FTM respectively).

2. PRELIMINARIES

Let a graph constitute a set of nodes $V = \{v_1, v_2, \dots, v_n\}$ and a set of edges $E = \{(v_i, v_j) | v_i, v_j \in V\} = \{e_1, e_2, \dots, e_{n-1}\}$. A labelled graph has a set of labels Σ, where a function $L: V \cup E \rightarrow \Sigma$ maps nodes with unique labels. A graph is connected but acyclic when it has at least one node that is connected to the rest of the graph by only one edge, which is leaf. For our purposes, the class of connected acyclic labelled graphs is of special interest, which is also called free tree, an unrooted unordered tree-like structure. In this paper, we denote a free tree with n nodes as n-free tree.

Let two free trees be t and T. t is a subtree of T if t can be obtained from T by repeatedly removing one degree nodes from its structure. Free trees t and T are *isomorphic* to each other if a bijective mapping exists between their set of nodes that preserves node labels, edge labels and also reflects the tree structures.

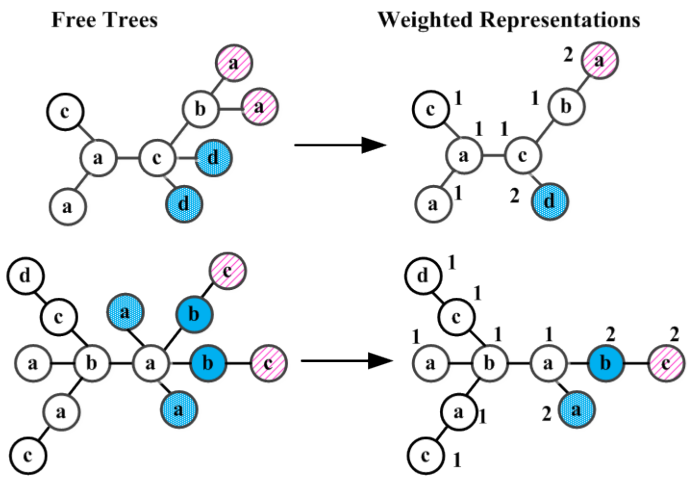


Figure 1: Equivalent nodes and the condensed weighted representations of free trees[8]

Let T_{db} be a database where each transaction is a labelled free tree. The problem of frequent free tree mining is to discover the complete set of frequent free subtrees. If tree $T \in T_{db}$ has a subtree isomorphic to subtree t, that indicates T has an

[8] Tree nodes are represented using labels and the edge labels are ignored in this paper.

occurrence of t in its structure. Formally we define the support of subtree t in T_{db} using the concept of occurrence as follows,

$$Occurrence\ (t, T) = \begin{cases} 1 & if\ t\ exists\ in\ T \\ 0 & otherwise \end{cases} \tag{1}$$

$$Support\ (t, T_{db}) = \sum_{T \in T_{db}} Occurence(t, T) \tag{2}$$

The subtree t is called frequent if *Support* $(t, T_{db}) \geq minsup$ where *minsup* is user-defined minimum support threshold.

In this paper, in a free tree, two adjacent nodes v_i and v_j with same label are defined as *equivalent nodes*, denoted by $v_i \cong v_j$. The *weight* of a node v_i is defined as the total number of its equivalent nodes and denoted by w_i (as shown in Figure 1). Using weights, we represent free trees of a database in a concise manner for further processing. Figure 1 shows an example of two free trees and their corresponding weighted representations by combining equivalent nodes (highlighted using different color patterns).

3. CANONICAL FORM FOR LABELLED FREE TREES

A *Canonical Form* (CF) of a tree is a representative form that can consistently represent many equivalent variations of that tree into one standard form [90, 188]. Several CFs have been proposed for rooted tree representations using traversing algorithms such as *depth-first-search* (DFS) or *breadth-first-search* (BFS) [90]. However, defining CF for free trees is non-trivial as it requires handling the vast variants that a free tree can have, i.e., the isomorphism problem. Due to the inherent structural flexibility (e.g., undefined root node and no direction among sibling nodes), there are more ways to represent a free tree than that of a rooted tree. A canonical form is critical for appropriate representation and efficient processing of free trees, because it ensures finding a common pattern amongst free trees. Before we define CF of free trees, we explain the process for unordered rooted trees and extend it to free trees.

3.1 Why Canonical Form Is Needed For Free Trees?

A *rooted tree* has a distinguished root node. A rooted tree that preserves order among the sibling nodes is called *rooted ordered*. This type of trees can easily be represented uniquely by using either the depth-first or the breadth-first string representations [90]. They do not face isomorphism. Two ordered trees will be similar iff all of its properties are identical; no variation is possible in similar rooted ordered trees [63]. Whereas, two similar unordered trees can have different orders among sibling nodes and these trees are called isomorphic trees. A free tree is also an unordered tree. The chance of having isomorphic trees in a database of free tree is very high due to the flexible property of being unrooted and unordered. Representing free trees using a systematic approach is non-trivial but critical to ensure its proper indexing for further processing and knowledge discovery.

Optimal Order: we will now briefly describe the concept of *optimal order* that is the basis of the proposed canonical form. An optimal order of a tree is an order obtained by the *balance optimal tree search (BOS) algorithm* [155] that traverses a rooted labelled tree uniquely, without the presence of sibling order information. Unlike existing traversal strategies [188], this algorithm works based on optimisation instead of enforcing a left-to-right order among siblings. Three heuristics are applied recursively in this traversing algorithm to find out the optimum traversing path of a tree. Heuristic 1 identifies a potential node during the traversal process. Heuristics 2 and 3 select the best node if multiple nodes are identified as candidates for traversal.

Heuristic 1 *After the root node traversal, the children of the root node, i.e.,* $\{v_i, v_j, \ldots, v_k\}$ *with weights* $\{w_i, w_j, \ldots, w_k\}$ *become eligible for traversing. The traversal order of these eligible nodes will be prioritized according to their ascending weights. The node with the highest weight is chosen first.*

Heuristic 2 *If two or more nodes* $\{v_i, v_j, \ldots, v_k\}$ *have the same maximum weight* (i.e. *maximum weight =* $MAX\{w_i, w_j, \ldots, w_k\}$), *the next node in the traversal order is selected based on the maximum number of their children* (*i.e., fan-out*).

Heuristic 3 *If two or more nodes hold the maximum weight with equal number of children, the traversal order will be prioritized using the minimum lexicographical order.*

The *optimal order* is unique even for trees that are isomorphic. This property is advantageous for mining frequent labelled free trees. For a free tree, several rooted ordered tree variations are possible only by changing the position of root node and the order among sibling nodes. An example can be seen in Figure 2, where a free tree is treated as rooted unordered tree with root node "v_a" (Figure 2a). Considering v_a as root node, several ordered variations of this free tree are shown in Figure 2(b, c, d, e).

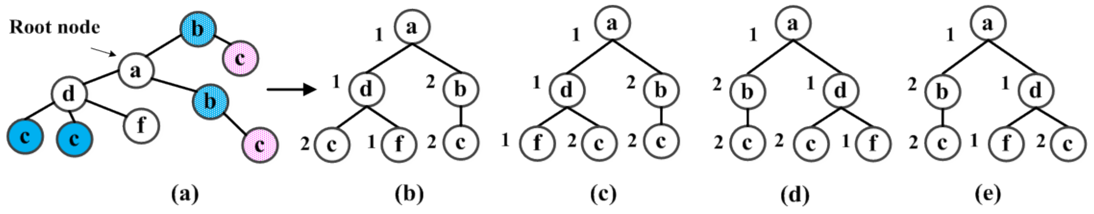


Figure 2: Four rooted ordered trees obtained from the same rooted unordered tree

According to the BOS algorithm [155] the unique optimal traversal order of all these equivalent ordered trees will be "v_a, v_b, v_c, v_d, v_c, v_f". In contrast, the BFS or DFS traversal [90] will provide different traversing order for each equivalent ordered tree because of its structure dependent strategy. It is desirable to obtain a unique canonical form of an ordered tree representation; however, it is absolutely critical to obtain a single canonical form for all equivalent variations of a free tree to allow efficient indexing for further processing. The proposed optimal traversal strategy is based on optimisation and is not sensitive to the structural changes. It gives the same optimal traversing order for all equivalent ordered trees that originate from a same free tree.

3.2 Balanced Optimal Canonical Form of Free Labelled Trees

If we can uniquely define root node of a free tree, then the optimal order can be used to define its canonical form. In this paper, we propose a two-step process for defining the canonical form of free trees. First, we normalise a free tree into the rooted unordered tree by fixing a root node and then we define the canonical form as well as canonical string.

Normalisation: This step includes a systematic approach to define a root node in a free tree. Following the commonly used technique [63, 64, 96], all the leaf nodes along with their incident edges in the free tree are removed at each step until a single node or two adjacent nodes are left. The tree with a single remained node is called a *central tree* and, the tree with a pair of remaining nodes is called a *bicentral tree* [96]. With the remaining single node, this node becomes the root of the free tree.

With the remaining two nodes, we apply *heuristic* 3 to obtain the root; therefore the node with minimum lexicographically ordered label becomes the root node.

The overall normalisation takes $O(|T|)$ time, where $|T|$ is the number of nodes in the free tree. Figure 3 shows the process of obtaining the root node from the free trees.

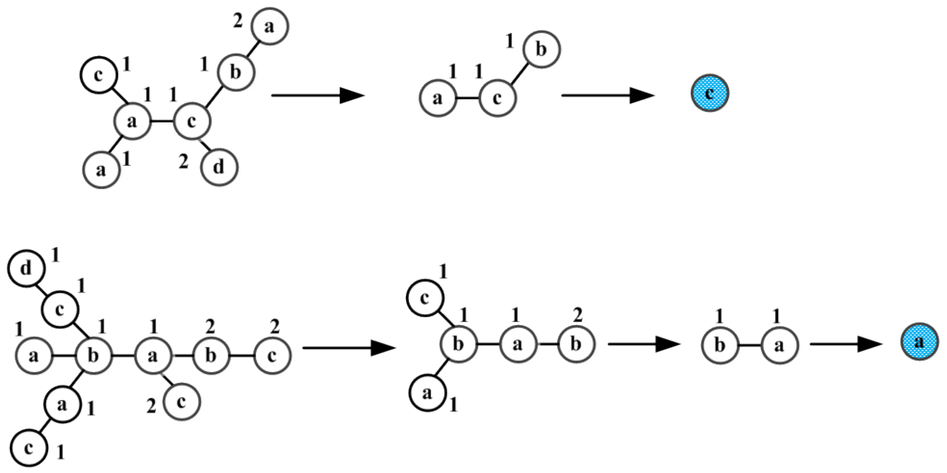


Figure 3: Process of finding a root node in free trees

Canonical Form and String: After the free tree is normalised to a rooted unordered tree, the balanced optimal canonical form can be defined as follows:

Definition 1 (*Balanced Optimal Canonical Form*): For a rooted labelled unordered tree, the balanced optimal canonical form is its optimal order of node labels along with corresponding weights.

A *canonical string representation* for labelled trees is equivalent to, but simpler than, canonical forms which facilitates frequency counting of trees in a database. For a balanced optimal canonical string encoding, we introduce four unique symbols +1, -1, +2 and -2 to specify directions on depth and breadth. More specifically, +1 and -1 are used to represent forward and backward travel towards depth between child and parent nodes; +2 and -2 are used to represent forward and backward travel towards breadth between sibling nodes respectively. We assume that none of these symbols are included in the alphabet of node labels. The canonical string representation of the rooted unordered tree is achieved by a guided record of sibling nodes,–"*under a parent node, a new node will always be recorded in a breadthwise direction from the existing rightmost sibling node.*"

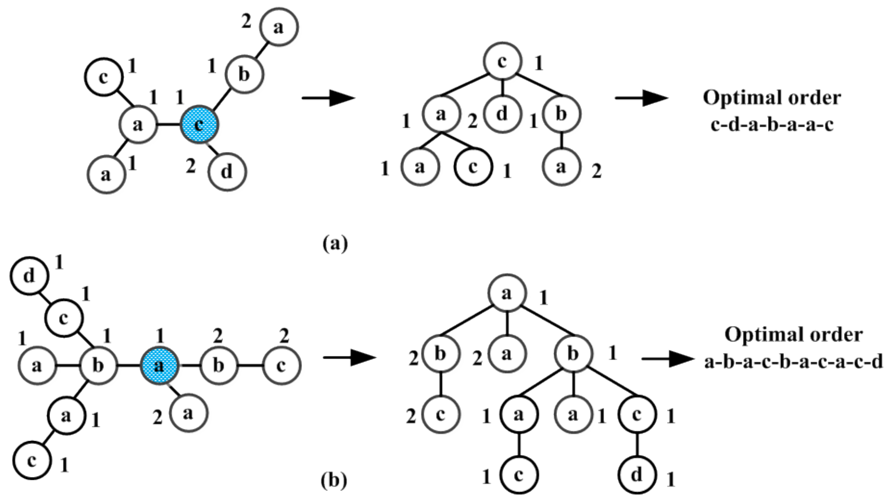


Figure 4: Balanced optimal canonical form of free tree

Example: For all the equivalent trees in Figure 2 with the unique optimal order "v_a, v_b, v_c, v_d, v_c, v_f , the balanced optimal string representation of these trees will be "1v_a, +1, 2v_b, +1, 2v_c, -1, +2, 1v_d, +1, 2v_c, -2, 1v_f". Similarly, the optimal canonical string of the free tree in Figure 4(a) will be "1v_c, +1, 2v_d, +2, 1v_a, +2, 1v_b, +1, 2v_a, -1, -2, +1, 1v_a, +2, 1v_c" and for the tree in Figure 4(b) will be "1v_a, +1, 2v_b, +2, 2v_a, -2, +1, 2v_c, -1, +2, +2, 1v_b, +1, 1v_a, +2, 1v_c, +2, 1v_a, -2, -2, +1, 1v_c, -1, +2, +1, 1v_d".

The isomorphic free trees can be successfully tracked because of having the same balanced optimal string representation. This ensures correct frequency counting for the processing of frequent subtrees. During the mining process, tree structural information such as level, weight, fan-out is stored that allows to differentiate the same alphabet appearing in different position. For sorting the optimal order it requires $O(|T| \log |T|)$ complexity, where $|T|$ is the number of nodes in a tree.

The balanced optimal canonical forms of free tree and rooted unordered tree embrace an interesting relationship which is described under Lemma 1. This relation is a fundamental step for growing the enumeration tree of free trees

Lemma 1: *Balanced optimal canonical form of a free tree is always the balanced optimal canonical form of a rooted unordered tree; however, the reverse is not true.*

PROOF: Consider a free tree *T*, with v_1, v_2, …,v_n nodes, with its balanced optimal canonical form t_{v1} that has a normalised root v_1. The *n*-number of different rooted unordered trees can be derived in their balanced optimal canonical forms t_{v1}; t_{v2}; …;

t_{vn} by changing the position of root in *T*. Only one of the balanced optimal canonical forms of these rooted unordered trees will have the same balanced optimal canonical form as the free tree, e.g. t_{v1}.

Prior to detailing our *FreeS* algorithm, we add following two lemmas that introduce important conditions which are essential to hold true during candidate free subtree enumeration through the balanced optimal canonical form representation. Fist we give the definitions of tree dimensions including depth, height and level [84].

Definition 2 (*Depth, Height, Level of Node*): For node v_i of a tree *T*, depth is the length of the unique path from that node towards the root node, denoted by $d(T, v_i)$. The height $h(v_i)$ of node v_i is the longest path from that node to a leaf. The height *H* of a tree is the height of root node, $h(v_0)$. The level of a node v_i in a tree *T* is defined as $Lv(T, v_i) = H - d(T, v_i)$.

Lemma 2: *Balanced optimal canonical form of a rooted unordered tree T with two nodes is balanced optimal canonical form of a free tree iff the root node has lexicographically minimum label.*

PROOF: *T* is a rooted unordered tree with two nodes, where v_0 is root and v_1 is its child. The optimal canonical form will be generated based on its optimal order, i.e., "v_0, v_1". Let us consider *case* 1, where root node v_0 has lexicographically minimum label. In this case treating *T* as free tree will end up having same canonical form as the rooted unordered tree, since a free tree considers the node with lexicographically minimum label as the center. Now consider *case* 2, where label of root node v_0 is higher than v_1. In this case the canonical form of free tree will be different than the rooted unordered tree, since v_1 will be the center instead of v_0.

Lemma 3: *Balanced optimal canonical form of a rooted unordered tree, T with* 3 *or more nodes and height H is balanced optimal canonical form of a free tree iff the following conditions hold:*

1. *The root has at least 2 children;*
2. *The root node has lexicographically smaller label than the labels of its children; and*

3. *One branch or subtree induced by a child of the root has a leaf node, v_i positioned at level $Lv(T, v_i) = 0$ (bottom level of the tree) and at least another branch or one subtree induced by another child of the root has a leaf node, v_j positioned at level $Lv(T, v_i) \leq 1$ (at most one level up than the last level).*

PROOF: For a rooted unordered tree *T* in its balanced optimal canonical form, we denote the root of *T* by v_0 and the children of v_0 by $v_1; \ldots; v_m$. Let us consider *case* 1. Tree *T* has 3 or more nodes and v_0 has only one child. It indicates that the rest of the nodes are appeared in that tree as child nodes of the immediate child of the root node. The node v_0 will be removed in the first step of finding center/bicenter. Consequently, v_0 cannot be the center or one of the bicentres. Therefore condition 1 will be held in this case. Let us consider *case* 2 when the root node v_0 has more than one child. This indicates that the leaf node of a subtree induced by one of $v_1; \ldots; v_k$ is at the bottom level of tree *T*. Assume this child to be v_j. If none of the subtrees induced by other child node of v_0 has a leaf node at the bottom level or second last level of tree *T*, then v_0 cannot be the center or one of the bicentres. This is because the center (or the bicenter) must be a node (or nodes) of the subtree induced by v_j. Without the loss of generality, we assume the subtree t_{v1} induced by v_1 has a leaf node at the bottom level of tree for which the path from root is *H*. The subtree t_{v2} induced by v_2 has a leaf node either at the last level or second last level. Therefore the path of that leaf node from root is either *H* or *H*-1. Now 2*H* or 2*H*-1 will be the length of path considering from the bottom-level leaf of t_{v1} to the bottom-level leaf of t_{v2} which makes v_0 as the center or one of the bicenters of the free tree. Therefore, condition 3 holds. Besides in case 2, it is essential to hold the condition 2 true, when *T* turns out to a bicentral tree and v_0 will only become the center if it has lexicographically minimum label.

4. FREQUENT FREE SUBTREE MINING ALGORITHM: FREES

FreeS consists of two main steps: (1) candidate subtree generation using the enumeration tree; and (2) frequency counting to determine frequent subtrees.

4.1 Candidate Subtree Generation using Enumeration Tree

Using the proposed balanced optimal canonical form of free trees and other tree structural information from a database, we define an enumeration tree that lists all subtrees in T_{db}, in their balanced optimal canonical forms. Since the underlying tree structure information is used for defining the enumeration tree, it is called tree-structure guided scheme based enumeration. To the best of our knowledge, *FreeS* is the first algorithm where this enumeration approach is used to generate candidate free trees.

Tree-Structure Guided Scheme based Enumeration Tree: The task here is enumerating a complete and non-redundant list of candidate subtrees from a given database. A candidate enumeration technique can generate both valid and invalid candidates. A candidate subtree is called valid if it exists in the considered database [58]. It is desirable to enumerate only the valid subtrees in order to reduce the computational efforts, instead of generating all possible candidates and prune invalid subtrees later. The tree-structure guided scheme based enumeration allows invalid subtrees, which will never be significant in spite of being frequent, to be excluded from counting the number of candidate trees. It utilises the tree structural information such as level, weight and fan-out of nodes, which are learned from a given database, in determining a valid subtree. This information is obtained after the free trees are normalised to rooted unordered trees. Instead of testing whether a tree actually exist in the database that is computationally expensive, a subtree is considered valid if it conforms to the tree structural information

Extending the Enumeration Tree: The right-path extension and join operations have been used to grow the enumeration tree. Previous research has shown that the right-path extension produces a complete and non-redundant candidate generation [38, 90, 96]. However, the use of extension alone for growing enumeration tree can be inefficient because the number of potential growth may be very large, especially when the cardinality of alphabets for node labels is large [90, 96]. This shortcoming necessitates of using a join operation; however, it often generates invalid subtrees. *FreeS* controls it by using the tree-structure guided scheme based enumeration. The basis of growing the enumeration tree of free trees is as follows: *By removing the last leg (node along with edge), i.e., the rightmost leg at the bottom level, of a* $(n+1)$*-free tree BOCF will result in the BOCF for another n-free tree.* The definitions of two operations for extending the enumeration tree are as follows.

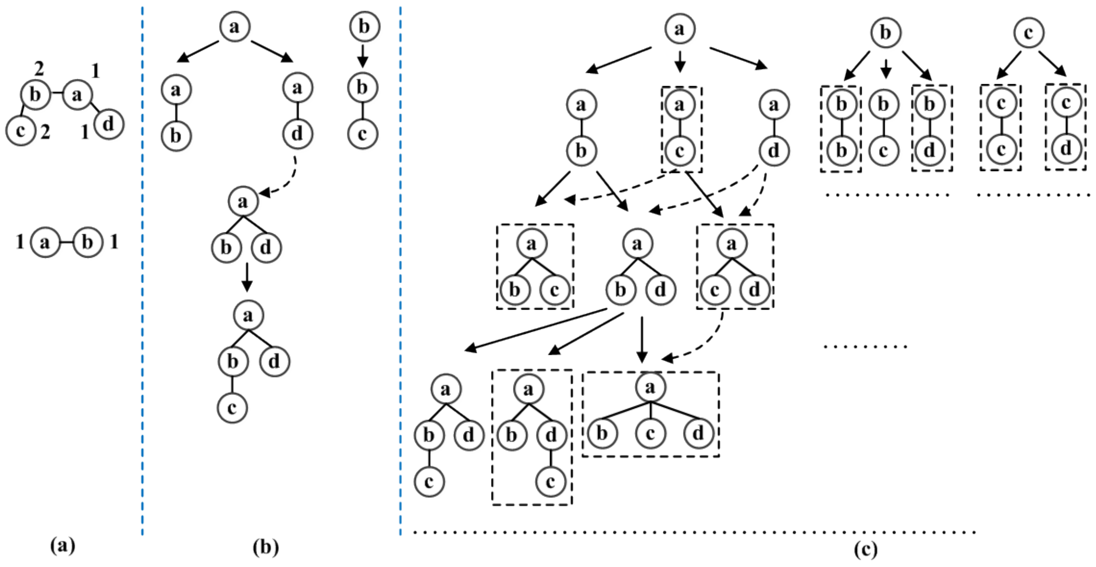


Figure 5: Sample database of labelled free trees (a), enumeration tree for free trees using tree structure guided scheme in *FreeS* (b) enumeration tree using the approach from HBT algorithm (c) (the dotted line with arrow is showing the candidates that are generated using join operations in HBT, and the dotted rectangle is showing the invalid candidate tree)

Definition 3 (*FreeS-extension*): For node v_i (fan-out ≠ 0) of a *n*-free tree in its balanced optimal canonical form t_v, an extension is possible by applying every frequent node label v_j that has a level equal to $Lv(t_v, v_i)$-1. This extension operation will result in another balanced optimal canonical form t'_v of a new (*n*+1)-free tree, with v_j child of $v_{i,}$ in the enumeration tree iff conditions of Lemma 2 and 3 are held. Further extension is possible from this new right-most node v_j iff conditions are fulfilled again.

Before giving the definition of *FreeS*-join operation, we define *equivalent group.*

Definition 4 (*Equivalent Group*): If two balanced optimal canonical forms t_v and t'_v of two *n*-free trees that have equal height *H* and common first *n*-1 nodes (along with labels and weights), they are considered as equivalent group, denoted by $t_v \cong t'_v$. Only the n^{th} node of each of these trees that appear last in their canonical forms are different.

Definition 5 (*FreeS-join*): Join operation is a guided extension between two free trees in balanced optimal canonical forms t_v and t'_v, that are members of an equivalent group, $t_v \cong t'_v$. Assume, v_i and v_j are the corresponding right-most node of t_v and t'_v, where $w_i > w_j$ or, $w_i = w_j$ with v_i lexicographically sorts lower than v_j. By joining v_j in t_v at the position of $Lv(t_v, v_i)$-1 will result in a new (n+1) node balanced optimal canonical form of free tree, denoted by $t_v \odot t'_v$, of the same height as tree t_v.

The join operation does not change the height or the level position of leaf nodes of a newly generated candidate tree, therefore Lemma 2 and 3 are not considered. As in the tree-structure guided approach, the enumeration tree growth is guided by the prior learned tree structure information. Therefore only valid subtrees are expected to be generated as candidate trees.

Consider an example database in Figure 5(a), where for minimum support 1, we compare the enumeration tree (Figure 5(b)) used by *FreeS* with the enumeration tree (Figure 5(c)) used by the *HybridTreeMiner* (HBT) method [96]. HBT also uses the right-path extension and join operations for growing the enumeration tree, but, these are defined using a different canonical form (Breadth First Canonical Form) [90], whereas we use BOCF and the tree-structure guided scheme for growing the enumeration tree. The dotted rectangles in (Figure 5(c)) show the generation of invalid subtrees in HBT. We only show a small part of the enumeration tree for HBT. If it is continued, it will grow in a much bigger size and will result in much higher numbers of invalid subtrees. In contrast, Figure 5(b) is the complete enumeration tree of the considered database for *FreeS*.

It can be clearly seen that the *FreeS* enumeration tree generates much less candidates in comparison to HBT enumeration tree because of producing only valid subtrees. Generation of invalid subtrees causes extra memory space and then, pruning of these subtrees causes additional computational cost for existing methods.

4.2 Frequency Counting

For counting frequency we modified the method described in [90, 96], which is basically an apriori like frequency counting that gives the exact support measure of each candidate subtree by maintaining an occurrence list. We used a catching technique to make the process of keeping occurrence list more efficient, which is "*stopped counting tree when the ID counter reaches the min support*", therefore the occurrence list becomes smaller than usual.

FreeS Algorithm

Input: Balanced optimal canonical form strings of labelled free trees present in a database T_{db}; level, weight and fan-out information of each node, minimum support (*minsup*) threshold.

Output: All frequent free subtrees.

```
1.  Result ←∅;
2.  Frq1 ← the set of all frequent subtrees of size 1;
3.  Frq2 ←∅;
4.  while Frq1 ≠ ∅ do
5.      for all c ∈ Frq1 do
6.          if fan-out(c) != 0
7.              Candidate ← Enumeration (c, Frq1, level, weight, fan-out );
8.          end if
9.          for all Ɛ′∈ Candidate do
10.             if support (Ɛ′) ≥ minsup then
11.                 Frq2 ← Frq2 ∪ Ɛ′;
12.             end if
13.         end for
14.     end for
15.     Frq1← Frq2;
16.     Result ← Result ∪ Frq1;
17.     Frq2 ← ∅;
18. end while
19. return Result
```

Figure 6: High level pseudo code of *FreeS* algorithm

Enumeration (l_k, *Frq*1, level, weight, fan-out)

```
1.  Output ←∅;
2.  for all Ɛ ∈ Frq1do
3.      Enumerate candidate $l_{k+1}$ by adding Ɛ;      /* Using FreeS-extension */
4.      Output ← Output ∪ $l_{k+1}$;
5.  end for
6.  for all equivalent groups in Output do
7.      $l_{k+2} \leftarrow l_{k+1} \odot l'_{k+1}$;      /* Using FreeS-join and $l_{k+1} \cong l'_{k+1}$ */
8.      Output ← Output ∪ $l_{k+2}$;
9.  end for
10. return (Output)
```

Figure 7: High level pseudo code of candidate generation

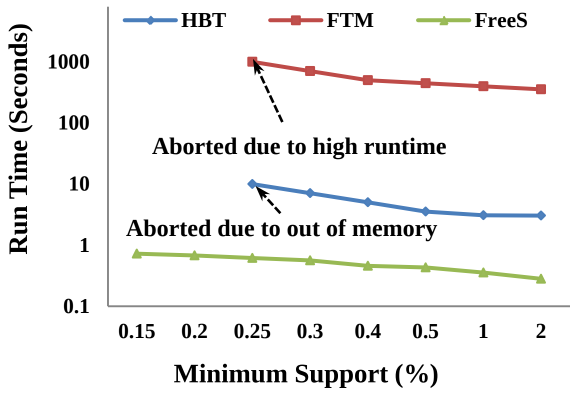
Figures 6 and 7 list the overall enumeration approach and the *FreeS* algorithm. The process of frequent subtree mining is initiated by scanning the database T_{db}, where free trees are stored as BOCF strings along with weight, level and fan-out information of each node. The set of frequent subtrees of size 1 is generated and the *Enumeration* method (in Figure 7) is called recursively for generating the candidates of larger sized subtrees. The frequency of every resultant candidate tree is computed. The full pruning is also performed to ensure downward-closure lemma [140]. But full pruning is expensive; therefore to accelerate this process we cease the frequency checking for a subtree belong to (*K*-1) set as soon as the *K* subtree is found frequent.

5. EMPIRICAL ANALYSIS

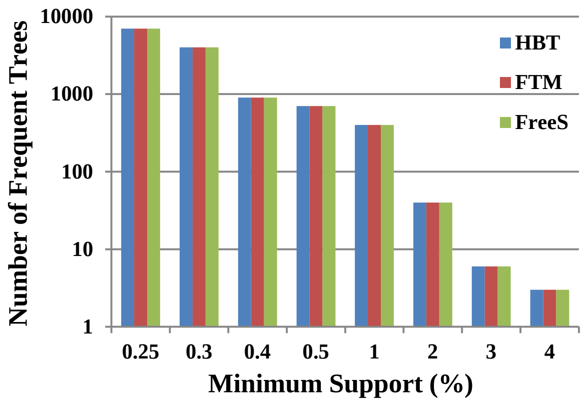
The efficacy of *FreeS* is shown by conducting systematic experiments using both real-life and synthetic datasets. *FreeS* is benchmarked with the most relevant and leading algorithms *FreeTreeMiner* (FTM) [90] and *HybridTreeMiner* (HBT) [96] which are designed to mine frequent free subtrees from a database of labelled free trees. All experiments have been done on a 2.8GHz Intel Core i7 PC with 8GB main memory and running the UNIX operating system.

CSLOGS: This real-life dataset has been widely used in evaluating various tree mining algorithms. CSLOGS [38, 70] contains web access trees of the CS department of Rensselaer Polytechnic Institute during one month. There are a total of 59,691 transactions and 13,209 unique node labels (corresponding to the URLs of the web pages).

Figure 8(a) shows that *FreeS* can find the same amount of subtrees in significant lesser time than its counterparts. Results show that below a certain support threshold (0.25%) the number of frequent trees explodes that causes huge memory consumption for HBT and consequently, the software automatically aborts the process. For calculating support of free trees, HBT uses occurrence list that makes the process faster, but, it is responsible for high memory usage too. *FreeS* performs this step within the memory size even for smaller minimum support threshold such as 0.15% because of using modified occurrence list. FTM does not suffer from the memory exhaustion problem though; however the run time increases drastically for smaller supports due to the lack of efficient frequency counting and inclusion of the expensive apriori candidate generation.



(a)

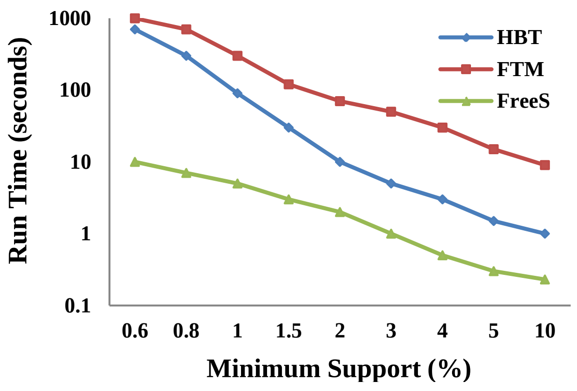


(b)

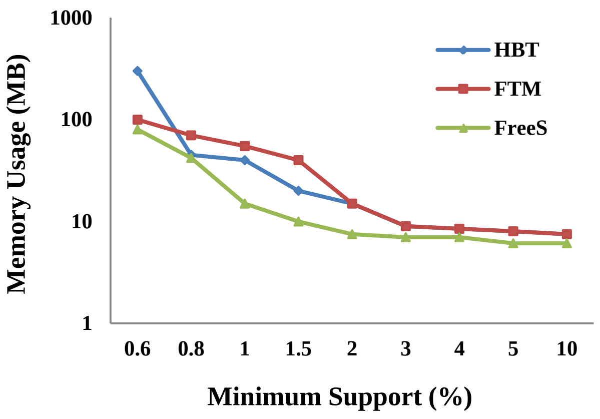
Figure 8: Run time comparison (a) and completeness test (b) using CSLOGS data (a $\log_{10}$ scale is used in Y axis)

The runtime performance of *FreeS* is few orders of magnitude better than HBT and FTM due to several reasons. (1) *FreeS* uses tree-structure guided based enumeration tree that allows enumerating only valid subtrees. (2) BOCF is defined to enumerate only one free tree for either of central or bicentral free trees, hence the occurrence list only keeps record of one tree. (3) A catching technique assists in keeping the occurrence list shorter. On the other hand, HBT can't avoid generating invalid candidate subtrees during enumeration, which results in extra memory consumption. HBT may also enumerate two free trees from a bicentral tree because of the supplementary canonical form concept [96]. Consequently, it will keep record of both trees which increases the size of the occurrence list.

Results in Figure 8(b) show that *FreeS* extracts the same amount of frequent patterns as the other state-of-the-art methods. The tree model guided enumeration employed in *FreeS* does not generate any invalid trees but does not miss on any valid trees. All three algorithms satisfy the completeness property and do not miss any frequent patterns since they all used full pruning (downward closure lemma), not an opportunistic pruning. This shows the accuracy of *FreeS* in finding subtrees.



(a)

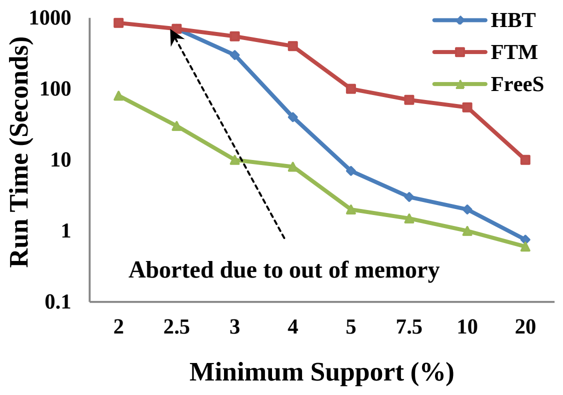


(b)

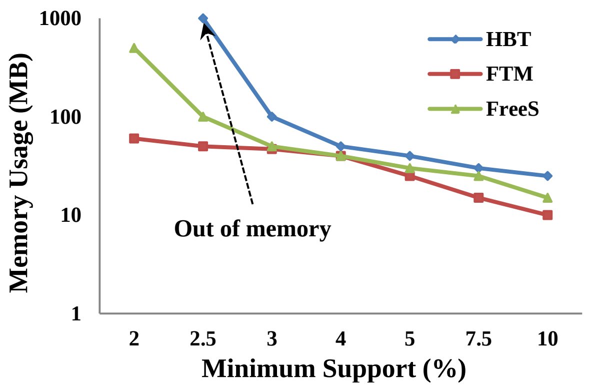
Figure 9: Memory usage comparison using dataset D1 (a $\log_{10}$ scale is used in Y axis)

Synthetic Data Sets: We conducted few more experiments using synthetic datasets with varied properties to support all of the above findings. The synthetic data sets were generated by a tree generator as described in [38]. The dataset called D1 is created using following parameters: the number of labels L = 10, the number of vertices in the master tree M = 100, the maximum depth D = 10, the maximum fan-out F = 5 and the total number of subtrees T = 5000. Such characteristics reflect the properties of web-browsing but not of very large databases. Result in Figure 9(a) shows that *FreeS* requires less runtime than HBT and FTM as expected. The memory consumption is also low for *FreeS*, whereas for being the small dataset the other two can also perform within the given memory size, Figure 9(b).

The dataset called D2 is generated using high fan-out, F = 20 with low number of labels L =10 and a moderate size dataset T = 10,000. The rest of the parameters are kept the same. This makes D2 having wider trees than the deep trees. The isomorphic problem is known to occur more commonly when trees have several siblings at same label. This facet of experiment will support the claim that *FreeS* can handle isomorphism more effectively than any other algorithms due to the use of BOCF.



(a)



(b)

Figure 10: Runtime (a) and memory (b) comparison using dataset D2 (a $\log_{10}$ scale is used in Y axis)

As shown in Figure 10, *FreeS* consumes much less processing time in comparison to other methods. It happens as *FreeS* does not generate a candidate tree multiple times because of using BOCF that ensures same identity for all isomorphic trees. Therefore, no additional test is required for checking the presence of isomorphism during frequency counting. In contrast, the state-of-the-art algorithms perform a mandatory isomorphism checking which makes them more expensive (Figure 10(a)).

Figure 10(b) shows that HBT consumes larger memory space than FTM and *FreeS*, and it becomes worse for smaller support thresholds. As explained before, FTM does not use occurrence list for frequency counting but computes the occurrences of each free tree. Therefore, it saves memory but consumes additional computational time. The usage of occurrence list becomes a pressing concern in terms of memory for large data, especially when the support threshold is low, but allows fast and efficient frequency checking. The catching mechanism employed in *FreeS* makes it consume less memory as well as the enumeration strategy does not generate any invalid subtrees, therefore *FreeS* can offer a good trade-off between memory usage and runtime.

6. CONCLUSION

In this paper, we consider an important problem of mining frequent free subtrees from a collection of free trees. We proposed a computationally efficient algorithm *FreeS* to discover all frequent subtrees in a database of free trees. A novel balanced optimal canonical form is introduced that ensures unique identity of frequent free trees even in presence of isomorphism. Because of this canonical form the isomorphism problem can be handled, that is responsible for computational complexity in this process. Moreover, the proposed tree-structure guided scheme based enumeration enables *FreeS* to reduce the cost for candidate generation by enumerating only valid subtrees. We modified the efficient apriori like occurrence list based frequency counting method that ensures less memory consumption.

Our empirical analyses show *FreeS* is scalable to mine frequent free trees in a large database of free trees with low support thresholds. In future we are planning to extend our algorithm for mining free trees in graph database.

Chapter 6: Conclusions

The omnipresence of tree data is noticeable in multitudinous domains such as web, computational biology, pattern recognition, XML databases and computer networks [14, 17, 19, 22, 61]. Mining tree databases is non-trivial and arises many issues in discovering knowledge due to the presence of hierarchical relationships, structural flexibility and enormous data expansion. This thesis focuses on mining the databases of labelled unordered trees, which is more challenging than mining the popular ordered tree type databases. This is because the flexibility in unordered tree structure causes issues with their representation, which affects their further processing.

The broad research objective of this study was discovering knowledge from the databases of labelled unordered trees in an efficient and scalable manner. In order to achieve the objective, this thesis presented algorithms for frequent subtree mining and tree matching, using novel and effective tree representation.

6.1 SUMMARY OF CONTRIBUTIONS

Based on the literature review as presented in Chapter 2, the following shortcomings were noted:

- Lack of current tree representation methods including tree traversing, canonical form and adjacency matrix for rooted unordered and free trees.
- Lack of an efficient and scalable tree matching algorithm for unordered trees.
- Lack of efficient frequent rooted unordered subtree algorithms.
- Lack of an efficient frequent free subtree mining algorithm.

This thesis has aimed to overcome these shortcomings by proposing novel tree representations, frequent subtree mining and tree matching algorithms. Firstly, in this thesis a novel balance-optimal-search traversing algorithm is proposed that provides an optimal traversal order for trees without relying on sibling orders. The canonical string-based representation, called balanced optimal canonical form, is proposed for rooted unordered trees and free trees. These canonical forms ensure one-to-one mapping between a labelled tree and a string by ensuring unique identity of isomorphic trees. This thesis also explored the matrix representation of trees and proposed two adjacency based matrix representations with information embedded in tree structures. These matrix representations ensure unique identity for the variations of the same unordered tree that is lacking in the traditional adjacency matrix representation.

Secondly, a tree matching algorithm is proposed for finding similarities between unordered tree pairs. One of the algorithms uses the Augmented Adjacency Matrix (AAM) for representing unordered trees and a cosine similarity metric is used for calculating the pairwise similarity. The cosine metric is modified for making it compatible with matrix computation. The other algorithm uses Extended Augmented Adjacency Matrix (EAAM)-based comparison for measuring similarities between trees. The EAAM matrix uses not only the embedded tree information along with adjacency but also uses knowledge of the considered database for representing the tree data. The similarity scores obtained by the matching algorithm based on each representation are utilised in clustering the unordered tree data. Empirical analysis shows the efficacy of this algorithm in clustering and establishes that the matrix-based comparison method is more computationally efficient than the traditional edit string operation based method, i.e., tree edit distance-based method.

Thirdly, algorithms for mining frequent subtrees for databases of labelled unordered and free trees are introduced, based on the canonical form BOCF. BOSTER is a tree structure guided scheme-based enumeration tree for systematically enumerating all frequent rooted unordered induced subtrees. BEST is a frequent rooted unordered embedded subtree mining algorithm using the tree structure guided scheme-based enumeration tree with the extension and join operations defined with changed level conditions of nodes. FreeS has been designed to extract frequent induced subtrees from the databases of free trees. Considering the literature review, FreeS is the first algorithm that has used the tree structure guided scheme-based enumeration for mining frequent free trees. These algorithms have addressed three different frequent subtree mining problems focusing on different tree types as inputs and different types as subtrees. The extensive experimental studies have been conducted to demonstrate the performance of the proposed algorithms as well as to compare the performance with the state-of-the-art algorithms.

6.2 SUMMARY OF FINDINGS

This section presents the main findings derived from this thesis:

- In response to Research Question 1, an optimisation-based tree traversal approach, new canonical forms and adjacency matrices are proposed in this thesis and the main findings are the following:
    - The BOS traversal ensures identical encoding of isomorphic rooted unordered and free trees. Unlike the BFS and DFS traversal, BOS is using optimisation for traversing trees, therefore the structural flexibilities (i.e. sibling ordering) does not impact the traversing order as well as encoding.
    - The AAM and EAAM adjacency matrices ensure identical representation for all variations of an unordered tree. Moreover, these matrices include more tree structural information in addition to adjacency information. With the proposed tree matching algorithm, these matrices showed improved accuracy performance over the traditional adjacency matrix in finding the trees pairwise matching (16% improvement in the value of FScore).
    - BOCFs ensure a common identity to a rooted unordered tree or a free tree in the presence of isomorphism without performing an expensive operation for finding the representative canonical form of the isomorphic trees from the sorted BFCF or DFCF string encodings. This unique characteristic of BOCFs allows it to save a significant amount of time during processing of the frequent unordered and free subtree mining algorithms.
    - An optimisation based representation that does not depend upon the sibling ordering can produce better results in any further manipulation like tree matching and frequent subtree mining.
- In response to Research Question 2, the proposed tree matching algorithm uses a matrix (e.g., AAM and EAAM)-based comparison instead of string

edit operations for measuring similarities. The following findings can be summarised.

  - The proposed AAM matrix-based tree matching algorithm requires significantly less computational time than the tree edit distance based methods, without compromising the accuracy of output. Incorporating optimal encoding and matrix calculation into the proposed method allows saving a significant amount of computation time. Any matrix-based computation is very fast and requires almost no time for processing, which motivated to represent trees in equivalent matrices and avoid the expensive edit string operation for calculating the approximate similarity score between a pair of trees. The optimal order allows the proposal of a matrix form that ensures identical representation of isomorphic unordered trees; also the additional tree information in AAM form offers more accuracy while processing the similarity calculation. The baseline algorithms [79, 112, 124] showed exponential complexity after reaching a tree size in the range of 60~65 nodes, while the proposed method yields a fraction of second runtime to determine pairwise similarity.
  - The proposed EAAM-based similarity measure method led to more accurate clustering results than the benchmark methods [77] through incorporating additional database specific knowledge in tree representation. A tree database, in which hierarchical relations of tree structures are frequent, can be found using a frequent subtree mining algorithm; adding this piece of information during the representation of tree structured data ensures more accuracy in its further manipulating processes, like tree matching and clustering. The results show that the proposed algorithm gives more accurate (on an average 10-15% improvement in FScore value) tree matching than the baseline as well as ensuring better clustering output.

– In response to Research Question 3, the proposed frequent subtree mining algorithms use the BOCF canonical form and an effective tree structure guided scheme-based enumeration tree for improving computational efficiency. All of these algorithms are compared against the popular and relevant benchmark algorithms using both synthetic and real life datasets. The experimental results indicate the following findings.

- All these algorithms can handle isomorphism issues more effectively than the state-of-the-art methods due to using BOCF representation. For example, while performing on a synthetic dataset with presence of isomorphic trees, the BOSTER algorithm is able to save 73.55% runtime in comparison to the UNI3 [98] algorithm for a small support threshold of 0.15%, while the HBT [96] algorithm could not even perform due to high memory usage. BEST has also been shown to save reasonable amounts of time and memory because of using BOCF. In FreeS algorithm, significant runtime improvement is achieved with reasonable memory use while performing on a synthetic dataset that has high probability of the presence of isomorphic trees.

- The BOSTER algorithm has shown consumption of less memory and less runtime for mining frequent rooted unordered induced subtrees, in comparison to the benchmarks, even in the presence of large data. Using the CSLOGS data, BOSTER is able to extract all induced subtrees a lot faster and with one order of magnitude less memory consumption than HBT [96], whereas UNI3 [98] could not even finish extracting the complete list of frequent subtrees due to excessive usage of memory.

- BEST outperforms SLEUTH and U3 algorithms [70, 97] in terms of runtime and memory usage without missing any frequent subtree generation. The tree structure guided scheme-based enumeration tree of BEST uses both join and extension operations to grow, which ensures faster computation compared to other algorithms. BEST also successfully avoids generating invalid subtrees as well as it does not need to save an additional data structure (e.g., embedding list) for checking isomorphism like U3. These improvements allow BEST to perform computations in memory.

- FreeS is the first algorithm that utilises a tree structure guided scheme-based enumeration tree for its candidate generation. This enumeration approach allows shortening the list of candidate trees, resulting in less memory utilisation in processing. FreeS has been found very fast using both real and synthetic datasets. It outperforms HybridTreeMiner [96] and FreeTreeMiner [63] by reducing the runtime expense (few orders of magnitude) without missing any frequent subtrees. The algorithm has also performed in memory for a large real life data while the other benchmark algorithms show an out of memory problem. The usage of a scalable enumeration approach, BOCF and modified occurrence list allows saving a lot of memory for FreeS.
- The runtime of FreeS in comparison to BOSTER and BEST with their corresponding benchmarking is found much lower. This may indicate that the optimal order is more beneficial for less constrained trees[9].

6.3 FUTURE WORKS

There is always room for work to be done to improve the scalability of existing methods. Besides working in this direction, some of the most promising other directions for future research relate to concise subtree mining; frequent subgraph mining; social network analysis and clustering. This section provides some hints and brief descriptions of these potential future research areas, which can be built upon from the base research that has been carried out in this thesis.

A. Further Scope of Improving The Proposed Method

The sensitivity behaviours of the proposed frequent subtree mining algorithms across various tree parameters are not completely known yet. This discovery will help to make these algorithms more generic. The scalability performance of an algorithm may vary depending upon various domain properties. An extensive sensitivity analysis can be considered for making the algorithm more efficient, regardless of any domain.

[9] This may be just a coincidence, as the research in the area of free tree mining is still underway and the state-of-the-art benchmarking algorithms are not that efficient as other benchmarking algorithms of rooted unordered tree.

Although the proposed algorithms performed well with large datasets in regards to both runtime and memory usage in comparison to benchmarking methods, the space efficiency of these algorithms can be improved further by improvising the data structures. For example, the possibility of proposing a more compact representation than AAM and EAAM can be checked, which will maintain the same level of accuracy performance but within reduced usage of space.

Further, the possibility of adding some new conditions to support free embedded candidate tree generation can be checked for the frequent free subtree mining algorithm, which may lead to proposing a new algorithm for mining frequent free embedded subtrees.

B. Concise Subtree Mining

Due to the large number of frequent subtrees generated, several researchers have focused their attention on finding the condensed representations of frequent patterns such as concise subtrees (e.g., closed and maximal subtrees). The efficiency of the algorithm for finding a condensed representation depends largely on the efficiency of the base algorithms that have been addressed in this thesis. The frequent concise subtree mining algorithms can be developed for mining patterns from unordered and free trees based on the efficient base algorithms as proposed in this thesis. For extracting these subtrees, the candidate generation process should be designed according to their definitions based on the BOCF canonical forms and incorporating pruning conditions. The tree structure guided scheme based-enumeration tree has rarely been used in designing frequent concise subtree mining algorithms in the literature. The enumeration process can be implemented in this area to improve performance. If it can be ascertained that the BOCF representation would be advantageous, in finding concise subtrees, this will be advantageous as this is an area that still requires an efficient algorithm to be developed.

C. Graph Mining

Since this thesis has presented an algorithm for mining frequent free trees, which can be seen as an acyclic version of graph data, therefore this algorithm can be helpful in the research field of graph mining. The proposed canonical form can be extended in order to deal with the graph data. In future this research can be carried out to check whether is it is possible to come up with other canonical forms tailored to graph morphisms like homomorphism or bisimulation, in the spirit of classic Web graph query languages like WG-log [198].

D. Implementation in Other Application Domains

Some important application domains like Social Network Analysis (SNA) and process mining in Business Process Management (BPM) can be considered as a future study. In some of these areas, the proposed contributions have direct implementation or have scope to evolve further in a way that could be useful in analysing these domains. Besides, the applicability of available SNA techniques in mining tree data especially in finding frequent free trees can be investigated, such as using the method of finding betweenness centrality, can be used in finding the root node for a free tree.

Lastly, evidence was given in Paper 3 included in this thesis that the knowledge gained from frequent subtree mining and from the tree matching can be combined for a better outcome; an effective integration would be an interesting avenue for future exploration.

Bibliography

1. Fayyad, U., G. Piatetsky-Shapiro, and P. Smyth, *From data mining to knowledge discovery: an overview*, in *Advances in Knowledge Discovery and Data Mining*, U.M. Fayyad, Piatetsky-Shapiro, G., Smyth, P. And Uthurusamy, R. , Editor. 1996, AAAI Press/MIT Press. p. 1-34.
2. Piateski, G. and W. Frawley, *Knowledge discovery in databases*. 1991: MIT press.
3. Chen, M.-S., J. Han, and P.S. Yu, *Data mining: an overview from a database perspective*. IEEE Transactions on Knowledge and Data Engineering, 1996. **8**(6): p. 866-883.
4. Nica, A., F.M. Suchanek, and A.S. Varde, *New Research Directions in Knowledge Discovery and Allied Spheres*. ACM SIGKDD Explorations Newsletter, 2015. **16**(2): p. 46-49.
5. Han, J. and M. kamber, *Data Mining Concepts And Techniques*. 2006, Morgan Kaufmann.
6. Neaga, E.I. and J.A. Harding*, *An enterprise modeling and integration framework based on knowledge discovery and data mining*. International Journal of Production Research, 2005. **43**(6): p. 1089-1108.
7. Choudhary, A.K., J.A. Harding, and M.K. Tiwari, *Data mining in manufacturing: a review based on the kind of knowledge*. Journal of Intelligent Manufacturing, 2009. **20**(5): p. 501-521.
8. Casillas, J. and F.J.M. Lopez, *A knowledge discovery method based on genetic-fuzzy systems for obtaining consumer behaviour patterns. An empirical application to a Web-based trust model*. International Journal of Management and Decision Making, 2009. **10**(5/6).
9. Nagabhushana, S., *Data Warehousing OLAP and Data Mining*. 2006: New Age International.
10. Han, J., X. Yan, and P.S. Yu, *Mining, Indexing, and Similarity Search in Graphs and Complex Structures*, in *International Conference on Data Engineering*. 2006.
11. Hadzic, F., H. Tan, and T.S. Dillon, *Mining of Data with Complex Structures*. Vol. 333. 2010: Springer Berlin Heidelberg.
12. Aggarwal, C.C. and H. Wang, *Managing and Mining Graph Data*. 2010, Boston/Dordrecht/London: Springer.
13. Cao, L., *Actionable Knowledge Discovery and Delivery*, in *Metasynthetic Computing and Engineering of Complex Systems*. 2015, Springer. p. 287-312.
14. Yan, X., P.S. Yu, and J. Han, *Substructure similarity search in graph databases*, in *International Conference on Management of Data*. 2005. p. 766-777.
15. Wang, J., et al., *Research on a frequent maximal induced subtrees mining method based on the compression tree sequence*. Expert Systems with Applications, 2015. **42**(1): p. 94-100.
16. Washio, T. and H. Motoda, *State of the art of graph-based data mining*. IEEE Transactions on Knowledge and Data Engineering, 2003. **5**(1): p. 59-68.

17. Tan, H., et al., *State of the art of data mining of tree structured information.* 2008.
18. Balcázar, J.L., A. Bifet, and A. Lozano, *Mining frequent closed rooted trees.* Machine Learning, 2010. **78**(1-2): p. 1-33.
19. Tekli, J., et al., *Approximate XML structure validation based on document–grammar tree similarity.* Information Sciences, 2015. **295**: p. 258-302.
20. Brisaboa, N.R., S. Ladra, and G. Navarro, *Compact representation of Web graphs with extended functionality.* Information Systems, 2014. **39**: p. 152-174.
21. Kuboyama, T., K. Hirata, and K. Aoki-Kinoshita, *An Efficient Unordered Tree Kernel and Its Application to Glycan Classification*, in *Advances in Knowledge Discovery and Data Mining*, T. Washio, et al., Editors. 2008, Springer Berlin Heidelberg. p. 184-195.
22. Kertész-Farkas, A., et al., *PTMTreeSearch: a novel two-stage tree-search algorithm with pruning rules for the identification of post-translational modification of proteins in MS/MS spectra.* Bioinformatics, 2014. **30**(2): p. 234-241.
23. Romanowski, C.J. and R. Nagi, *On comparing bills of materials: a similarity/distance measure for unordered trees.* IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 2005. **35**(2): p. 249-260.
24. Kashkoush, M. and H. ElMaraghy, *Matching Bills of Materials Using Tree Reconciliation.* Procedia CIRP, 2013. **7**: p. 169-174.
25. Jansson, J. and W.-K. Sung, *Maximum Agreement Supertree*, in *Encyclopedia of Algorithms*, M.-Y. Kao, Editor. 2015, Springer US. p. 1-5.
26. Zhang, S. and J.T.-L. Wang, *Discovering frequent agreement subtrees from phylogenetic data.* IEEE Transactions on Knowledge and Data Engineering, 2008. **20**(1): p. 68-82.
27. Hadzic, F., H. Tan, and T. Dillon, *TMG Framework for Mining Unordered Subtrees*, in *Mining of Data with Complex Structures*. 2010, Springer Berlin Heidelberg. p. 139-174.
28. Balcázar, J., A. Bifet, and A. Lozano, *Mining Frequent Closed Unordered Trees Through Natural Representations*, in *Conceptual Structures: Knowledge Architectures for Smart Applications*, U. Priss, S. Polovina, and R. Hill, Editors. 2007, Springer Berlin Heidelberg. p. 347-359.
29. Wang, Y., D.J. DeWitt, and J.-Y. Cai. *X-Diff: An Effective Change Detection Algorithm for XML Documents*. in *Proceedings of the 19th International Conference on Data Engineering*. 2003. Vienna: IEEE.
30. Zhao, Q., et al., *Discovering frequently changing structures from historical structural deltas of unordered XML*, in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 2004, ACM: Washington, D.C., USA. p. 188-197.
31. Zhao, Q., et al., *XML structural delta mining: Issues and challenges.* Data & Knowledge Engineering, 2006. **59**(3): p. 627-651.
32. Boiret, A., et al., *Logics for Unordered Trees with Data Constraints on Siblings*, in *Language and Automata Theory and Applications*. 2015, Springer. p. 175-187.
33. Punin, J.R., M.S. Krishnamoorthy, and M.J. Zaki, *LOGML: Log markup language for web usage mining*, in *WEBKDD 2001—Mining Web Log Data Across All Customers Touch Points*. 2002, Springer. p. 88-112.

34. Chehreghani, M.H. *Efficiently Mining Unordered Trees*. in *Proceedings of the 11th IEEE International Conference on Data Mining*. 2011. Vancouver, BC.
35. Shasha, D., J.T.-L. Wang, and S. Zhang. *Unordered tree mining with applications to phylogeny*. in *Proceedings on the 20th International Conference on Data Engineering, (ICDE' 04).* . 2004. IEEE.
36. Zhang, S., Z. Du, and J.T. Wang, *New Techniques for Mining Frequent Patterns in Unordered Trees*. 2015.
37. Deepak, A., et al., *EvoMiner: frequent subtree mining in phylogenetic databases.* Knowledge and Information Systems, 2014. **41**(3): p. 559-590.
38. Zaki, M.J., *Efficiently Mining Frequent Trees in A Forest: Algorithms and Applications.* IEEE Transactions on Knowledge and Data Engineering, 2005. **17**(8): p. 1021-1035.
39. Chowdhury, I.J. and R. Nayak, *Measuring Similarity between Unordered Trees with the Balanced-Optimal-Search Traversal Algorithm (Under Review).* Knowledge and Information Systems (KAIS).
40. Fukagawa, D., et al., *A Clique-based Method for the Edit Distance between Unordered Trees and Its Application to Analysis of Glycan Structures.* BMC Bioinformatics, 2011. **12**(1): p. 1-9.
41. Pavel Zezula , F.M., Federica M , Riccardo Martoglia, *Unordered XML Pattern Matching with Tree Signatures.* SOFSEM Conference, 2004.
42. Bille, P., *A survey on tree edit distance and related problems.* Theoretical Computer Science, 2005. **337**(1-3): p. 217-239.
43. Jiang, T., L. Wang, and K. Zhang, *Alignment of trees — an alternative to tree edit.* Theoretical Computer Science, 1995. **143**(1): p. 137-148.
44. Kilpelainen, P. and H. Mannila, *Ordered and Unordered Tree Inclusion.* SIAM Journal on Computing, 1995. **24**(2): p. 340-17.
45. Hirata, K., Y. Yamamoto, and T. Kuboyama. *Improved MAX SNP-Hard Results for Finding an Edit Distance between Unordered Trees*. in *The 22nd Annual Conference on Combinatorial Pattern*. 2011. Palermo, Italy: Springer Berlin Heidelberg.
46. Yamamoto, Y., K. Hirata, and T. Kuboyama, eds. *On Computing Tractable Variations of Unordered Tree Edit Distance with Network Algorithms*. New Frontiers in Artificial Intelligence, ed. M. Okumura, D. Bekki, and K. Satoh. Vol. 7258. 2012, Springer Berlin Heidelberg. 211-223.
47. Zhang, K. and T. Jiang, *Some MAX SNP-hard results concerning unordered labeled trees.* Information Processing Letters, 1994. **49**(5): p. 249-254.
48. Zhang, K., R. Statman, and D. Shasha, *On the Editing Distance between Unordered Labeled Trees.* Information Processing Letters, 1992. **42**(3): p. 133-139.
49. Chi, Y., et al., *Frequent Subtree Mining - An Overview.* Fundamental Informatic., 2004. **66**(1-2): p. 161-198.
50. Zaki, M.J. and C.C. Aggarwal. *XRules: An Effective Structural Classifier for XML Data*. in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2003. Washington, D. C.: ACM.
51. Nayak, R., *Fast and effective clustering of XML data using structural information.* Knowledge and Information Systems, 2008. **14**(2): p. 197-215.
52. Han, J., et al., *Frequent pattern mining: current status and future directions.* Data Mining and Knowledge Discovery, 2007. **15**(1): p. 55-86.

53. Chehreghani, M.H., *Efficiently Mining Unordered Trees*, in *IEEE 11th International Conference on Data Mining (ICDM)*. 2011: Vancouver, BC. p. 111-120.
54. Rückert, U. and S. Kramer. *Frequent free tree discovery in graph data*. in *Proceedings of the 2004 ACM symposium on Applied computing*. 2004. ACM.
55. Tan, H., et al., *IMB3-Miner: mining induced/embedded subtrees by constraining the level of embedding*, in *Advances in Knowledge Discovery and Data Mining*. 2006, Springer. p. 450-461.
56. Hadzic, F., H. Tan, and T. Dillon, *Algorithm Development Issues*, in *Mining of Data with Complex Structures*. 2010, Springer Berlin Heidelberg. p. 41-65.
57. Chowdhury, I.J. and R. Nayak. *BOSTER: An Efficient Algorithm for Mining Frequent Unordered Induced Subtrees*. in *WISE 15th International Conference on Web Information Systems Engineering*. 2014. Athens, Greece: Springer Berlin Heidelberg.
58. Chowdhury, I.J. and R. Nayak. *BEST: An Efficient Algorithm for Mining Frequent Unordered Embedded Subtrees* in *PRICAI 13th Pacific Rim International Conference on Artificial Intelligence*. 2014. Gold Coast, Australia: Springer Berlin Heidelberg.
59. Chowdhury, I.J. and R. Nayak. *FreeS: Fast Algorithm to Discover Frequent Free Subtrees Using a Novel Canonical Form (Accepted)*. in *WISE 16th International Conference on Web Information Systems Engineering*. 2015. MIami, Florida, USA: Springer Berlin Heidelberg.
60. Yang, R., P. Kalnis, and A.K. Tung. *Similarity Evaluation on Tree-Structured Data*. in *the Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2005. Maryland, USA: ACM.
61. Hadzic, F., *A Structure Preserving Flat Data Format Representation for Tree-Structured Data*, in *New Frontiers in Applied Data Mining*, L. Cao, et al., Editors. 2012, Springer Berlin Heidelberg. p. 221-233.
62. Cormen, T.H., et al., *Introduction to Algorithms*. 2001, Cambridge: MIT Press and McGraw-Hill.
63. Chi, Y., Y. Yang, and R.R. Muntz. *Indexing and mining free trees*. in *In Third IEEE International Conference on Data Mining, 2003, (ICDM'03)* 2003. IEEE.
64. Zhao, P. and J. Yu, *Fast Frequent Free Tree Mining in Graph Databases*. World Wide Web, 2008. **11**(1): p. 71-92.
65. Scholl, A., *Balancing and Sequencing of Assembly Lines*. 1999, Heidelberg: Physica-Verlag.
66. Nijssen, S. and J.N. Kok, *A quickstart in frequent structure mining can make a difference*, in *International conference on Knowledge discovery and data mining (Proceedings of the tenth ACM SIGKDD)*. 2004, ACM: Seattle, WA, USA. p. 647-652.
67. Anderson, R., *Professional XML*. 2000, Birmingham, England: Wrox Press Ltd.
68. Romanowski, C.J. and R. Nagi, *A data mining approach to forming generic bills of materials in support of variant design activities*. Journal of Computing and Information Science in Engineering, 2004. **4**(4): p. 316-328.
69. Watts, F.B., *Configuration Management Metrics*. 2009, William Andrew: Burlington.

70. Zaki, M.J., *Efficiently Mining Frequent Embedded Unordered Trees.* Fundamental Informatic, 2004. **66**(1-2): p. 33-52.
71. Møller, A. and M.I. Schwartzbach, *An Introduction to XML and Web Technologies*. 2006: Addison-Wesley.
72. Nayak, R. and S. Xu, *XCLS: A Fast and Effective Clustering Algorithm for Heterogenous XML Documents*, in *Advances in Knowledge Discovery and Data Mining*, W.-K. Ng, et al., Editors. 2006, Springer Berlin Heidelberg. p. 292-302.
73. Cooley, R., B. Mobasher, and J. Srivastava. *Web mining: Information and pattern discovery on the world wide web*. in *Ninth IEEE International Conference onTools with Artificial Intelligence*. 1997. IEEE.
74. Cooley, R., B. Mobasher, and J. Srivastava, *Data Preparation for Mining World Wide Web Browsing Patterns.* Knowledge and Information Systems, 1999. **1**(1): p. 5-32.
75. Punin, J.R., M.S. Krishnamoorthy, and M.J. Zaki, *Web usage mining—Languages and algorithms*, in *Exploratory Data Analysis in Empirical Research*. 2003, Springer. p. 266-281.
76. Hopp, W.J. and M.L. Spearman, *Factory Physics*. 2011: Waveland PressInc.
77. Shih, H.M., *Product structure (BOM)-based product similarity measures using orthogonal procrustes approach.* Computers & Industrial Engineering, 2011. **61**(3): p. 608-628.
78. Aoki-Kinoshita, K.F., et al., *ProfilePSTMM: capturing tree-structure motifs in carbohydrate sugar chains.* Bioinformatics, 2006. **22**(14): p. e25-e34.
79. Akutsu, T., et al., *An Improved Clique-Based Method for Computing Edit Distance between Unordered Trees and Its Application to Comparison of Glycan Structures*, in *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*. 2011. p. 536-540.
80. Kanehisa, M., *KEGG for representation and analysis of molecular networks involving diseases and drugs.* Nucleic acids research, 2010. **38**(suppl 1): p. D355-D360.
81. Li, G., et al., *Efficient Similarity Search for Tree-Structured Data*, in *Scientific and Statistical Database Management*, B. Ludäscher and N. Mamoulis, Editors. 2008, Springer Berlin Heidelberg. p. 131-149.
82. Diestel, R., *Graph theory {graduate texts in mathematics; 173}*. 2000: Springer-Verlag Berlin and Heidelberg GmbH & amp.
83. Valiente, G., *Algorithms on trees and graphs*. 2013: Springer Science & Business Media.
84. Ullman, J.D., A.V. Aho, and J.E. Hopcroft, *The design and analysis of computer algorithms*. Addison-Wesley, Reading. Vol. 4. 1974. 1.2-4.3.
85. Suciu, D., *Semistructured data and XML Information.*, in *Information Organization and Databases: Foundations of Data Organization.*, K. Tanaka, Ghandeharizadeh, S., Kambayashi, Y., Editor. 2000, Kluwer Academic Publishers: Dordrecht. p. 9–30.
86. Feng, L., et al. *An XML-enabled association rule framework*. in *Database and Expert Systems Applications*. 2003. Springer.
87. Han, J. and M. Kamber, *Data mining: concepts and techniques (the Morgan Kaufmann Series in data management systems).* 2000.
88. Cormen, T.H., et al., *Representations of graphs*, in *Introduction to Algorithms*. 2009, MIT Press and McGraw-Hill: Cambridge. p. 524-531.

89. Hopcroft, J.E. and R.E. Tarjan, *Efficient algorithms for graph manipulation.* 1971.
90. Chi, Y., Y. Yang, and R.R. Muntz, *Canonical Forms for Labelled Trees and Their Applications in Frequent Subtree Mining.* Knowledge and Information System, 2005. **8**(2): p. 203-234.
91. Nijssenm, S. and J.N. Kok, *Efficient Discovery of Frequent Unordered Trees*, in *First International Workshop on Mining Graphs, Trees and Sequences*. 2003, Springer Berlin Heidelberg: Croatia.
92. Asai, T., et al., *Discovering Frequent Substructures in Large Unordered Trees*, in *Discovery Science*, G. Grieser, Y. Tanaka, and A. Yamamoto, Editors. 2003, Springer Berlin Heidelberg. p. 47-61.
93. Luccio, F., et al., *Exact Rooted Subtree Matching in Sublinear Time*, in *Universita Di Pisa Technical Report TR-01*. 2001.
94. Luccio, F., et al., *Bottom-up subtree isomorphism for unordered labeled trees.* International Journal of Pure and Applied Mathematics, 2007. **38**(3): p. 325.
95. Nijssen S, K.J., *Efficient discovery of frequent unordered trees.* First international workshop on mining graphs, trees and sequences, 2003.
96. Chi, Y., Y. Yang, and R.R. Muntz. *HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms*. in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*. 2004. Santorini: IEEE.
97. Hadzic, F., H. Tan, and T.S. Dillon, *U3 - Mning Unordered Embedded Subtrees Using TMG Candidate Generation*, in *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*. 2008, IEEE Computer Society. p. 285-292.
98. Hadzic, F., H. Tan, and T.S. Dillon, *UNI3 - efficient algorithm for mining unordered induced subtrees using TMG candidate generation*, in *IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. 2007: Honolulu, Hawaii. p. 568-575.
99. Huan, J., W. Wang, and J. Prins. *Efficient mining of frequent subgraphs in the presence of isomorphism*. in *Third IEEE International Conference on Data Mining (ICDM), 2003.* . 2003.
100. Inokuchi, A., T. Washio, and H. Motoda, *An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data*, in *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*. 2000, Springer-Verlag. p. 13-23.
101. Choi, S.-S., S.-H. Cha, and C.C. Tappert, *A survey of binary similarity and distance measures.* Journal of Systemics, Cybernetics and Informatics, 2010. **8**(1): p. 43-48.
102. Chen, Y. and D. Cooke. *Unordered Tree Matching and Strict Unordered Tree Matching: The Evaluation of Tree Pattern Queries*. in *the 2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 2010. Huangshan: IEEE Computer Society.
103. Shasha, D., et al., *Exact and approximate algorithms for unordered tree matching.* Systems, Man and Cybernetics, IEEE Transactions on, 1994. **24**(4): p. 668-678.
104. Lu, S.Y., *A tree-to-tree distance and its application to cluster analysis* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1979.

105. Hackman, S.T., M.J. Magazine, and T.S. Wee, *Fast, Effective Algorithms for Simple Assembly Line Balancing Problems.* Operation Research, 1989. **37**(6): p. 916-924.
106. Levenshtein, V.I. *Binary codes capable of correcting deletions, insertions, and reversals*. in *Soviet physics doklady*. 1966.
107. Ogawa, H., *Labeled point pattern matching by Delaunay triangulation and maximal cliques.* Pattern Recognition, 1986. **19**(1): p. 35-40.
108. Tomita, E., et al., *A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique*, in *WALCOM: Algorithms and Computation*, M.S. Rahman and S. Fujita, Editors. 2010, Springer Berlin Heidelberg. p. 191-203.
109. Pawlik, M. and N. Augsten, *RTED: a robust algorithm for the tree edit distance.* Proceedings of the VLDB Endowment, 2011. **5**(4): p. 334-345.
110. Pawlik, M. and N. Augsten, *Efficient Computation of the Tree Edit Distance.* ACM Transactions on Database Systems (TODS), 2015. **40**(1): p. 3.
111. Arora, S., et al., *Proof verification and the hardness of approximation problems.* Journal of the ACM (JACM), 1998. **45**(3): p. 501-555.
112. Akutsu, T., et al., *Exact algorithms for computing the tree edit distance between unordered trees.* Theoretical Computer Science, 2011. **412**(4-5): p. 352-364.
113. Horesh, Y., R. Mehr, and R. Unger, *Designing an A* Algorithm for Calculating Edit Distance between Rooted-Unordered Trees.* Journal of Computational Biology, 2006 **13**(6): p. 1165-1176.
114. Demaine, E.D., et al., *An optimal decomposition algorithm for tree edit distance.* ACM Transactions on Algorithms, 2009. **6**(1): p. 1-19.
115. Kaizhong, Z., *Algorithms for the constrained editing distance between ordered labeled trees and related problems.* Pattern Recognition, 1995. **28**(3): p. 463-474.
116. Fukagawa, D., T. Akutsu, and A. Takasu. *Constant factor approximation of edit distance of bounded height unordered trees*. in *String Processing and Information Retrieval*. 2009. Springer.
117. Shasha, D. and K. Zhang, *Fast algorithms for the unit cost editing distance between trees.* Journal of Algorithms, 1990. **11**(4): p. 581-621.
118. Akutsu, T., et al., *Efficient Exponential Time Algorithms for Edit Distance between Unordered Trees*, in *Combinatorial Pattern Matching*, J. Kärkkäinen and J. Stoye, Editors. 2012, Springer Berlin Heidelberg. p. 360-372.
119. Selkow, S.M., *The tree-to-tree editing problem.* Information Processing Letters, 1977. **6**(6): p. 184-186.
120. Kilpelainen, P.T., *Tree matching problems with applications to structured text databases*. 1992, Helsingin Yliopisto (Finland): Finland. p. 110-110 p.
121. Richter, T., *A new algorithm for the ordered tree inclusion problem.* in the Proceeding of 8th Annual Symposium on Combinatorial Pattern Matching (CPM), Lecture Notes of Computer Science (LNCS), 1997. **1264**: p. 150–166.
122. Chen, W., *More Efficient Algorithm for Ordered Tree Inclusion.* Journal of Algorithms, 1998. **26**(2): p. 370-385.
123. Matoušek, J. and R. Thomas, *On the complexity of finding iso- and other morphisms for partial k-trees.* Discrete Mathematics, 1992. **108**(1–3): p. 343-364.

124. Mori, T., et al., *A clique-based method using dynamic programming for computing edit distance between unordered trees.* Journal of computational biology, 2012. **19**(10): p. 1089-1104.
125. Okamoto, Y. and T. Shoudai. *Hardness of Learning Unordered Tree Contraction Patterns*. in *Advanced Applied Informatics (IIAIAAI), 2013 IIAI International Conference on*. 2013. IEEE.
126. Yoshimura, Y. and T. Shoudai, *Learning Unordered Tree Contraction Patterns in Polynomial Time*, in *Inductive Logic Programming*, F. Riguzzi and F. Železný, Editors. 2013, Springer Berlin Heidelberg. p. 257-272.
127. Vercoustre, A.-M., et al., *A flexible structured-based representation for XML document mining*. 2006: Springer.
128. Pelillo, M., K. Siddiqi, and S.W. Zucker, *Matching hierarchical structures using association graphs.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1999. **21**(11): p. 1105-1120.
129. Torsello, A. and E.R. Hancock, *Computing approximate tree edit distance using relaxation labeling.* Pattern Recognition Letters, 2003. **24**(8): p. 1089-1097.
130. Shin, K., *Tree Edit Distance and Maximum Agreement Subtree.* Information Processing Letters, 2015. **115**(1): p. 69-73.
131. Akutsu, T., D. Fukagawa, and A. Takasu, *Improved approximation of the largest common subtree of two unordered trees of bounded height.* Information Processing Letters, 2008. **109**(2): p. 165-170.
132. Deza, M.M. and E. Deza, *Encyclopedia of distances*. 2009: Springer.
133. Cha, S.-H., *Comprehensive survey on distance/similarity measures between probability density functions.* International Journal of Mathematical Models and Methods in Applied Sciences, 2007. **1**(4): p. 300-307.
134. Kutty, S., R. Nayak, and Y. Li, *XML Documents Clustering Using a Tensor Space Model*, in *Advances in Knowledge Discovery and Data Mining*, J. Huang, L. Cao, and J. Srivastava, Editors. 2011, Springer Berlin Heidelberg. p. 488-499.
135. Nayak, R. and W. Iryadi, *XML schema clustering with semantic and hierarchical similarity measures.* Knowledge-Based Systems, 2007. **20**(4): p. 336-349.
136. Naghibzadeh, M., *Tag Name Structure-based Clustering of XML Documents.* International Journal of Computer and Electrical Engineering-IJCEE, 2010.
137. Antonellis, P., C. Makris, and N. Tsirakis. *XEdge: clustering homogeneous and heterogeneous XML documents using edge summaries*. in *Proceedings of the 2008 ACM symposium on Applied computing*. 2008. ACM.
138. Hadzic, F., T.S. Dillon, and H. Tan, *Mining of Data with Complex Structures*. 2011: Springer Berlin Heidelberg.
139. Pelillo, M., *Matching free trees, maximal cliques, and monotone game dynamics.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002. **24**(11): p. 1535-1541.
140. Agrawal, R. and R. Srikant. *Fast Algorithms for Mining Association Rules in Large Databases*. in *Proceedings of the 20th International Conference on Very Large Data Bases*. 1994. Morgan Kaufmann Publishers Inc.
141. Liu, T.-L. and D. Geiger. *Approximate tree matching and shape similarity*. in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. 1999. IEEE.

142. Chehreghani, M.H., M. Rahgozar, and C. Lucas. *Mining maximal embedded unordered tree patterns*. in *IEEE Symposium on Computational Intelligence and Data Mining, 2007 (CIDM 2007)* 2007. IEEE.
143. Aggarwal, C.C., M.A. Bhuiyan, and M.A. Hasan, *Frequent Pattern Mining Algorithms: A Survey*, in *Frequent Pattern Mining*, C.C. Aggarwal and J. Han, Editors. 2014, Springer International Publishing. p. 19-64.
144. Inokuchi, A., et al. *General framework for mining frequent patterns in structures*. in *Proceedings of the ICDM-2002 workshop on Active Mining (AM-2002)*. 2002.
145. Kuramochi, M. and G. Karypis, *Frequent Subgraph Discovery*, in *IEEE International Conference on Data Mining (ICDM), 2001* 2001, IEEE Computer Society. p. 313-320.
146. Tan, H., et al., *MB3-Miner: efficiently mining eMBedded subTREEs using Tree Model Guided candidate generation.* 2005.
147. Tatikonda, S., S. Parthasarathy, and T. Kurc. *TRIPS and TIDES: new algorithms for tree mining*. in *Proceedings of the 15th ACM international conference on Information and knowledge management*. 2006. ACM.
148. Wang, C., et al., *Efficient pattern-growth methods for frequent tree pattern mining*, in *Advances in Knowledge Discovery and Data Mining*. 2004, Springer. p. 441-451.
149. Han, J., J. Pei, and Y. Yin, *Mining frequent patterns without candidate generation.* ACM SIGMOD Record, 2000. **29**(2): p. 1-12.
150. Pavón, J., S. Viana, and S. Gómez. *Matrix Apriori: Speeding Up the Search for Frequent Patterns*. in *Databases and Applications*. 2006.
151. Ghoting, A., et al. *Cache-conscious frequent pattern mining on a modern processor*. in *Proceedings of the 31st international conference on Very large data bases*. 2005. VLDB Endowment.
152. Xiao, Y., et al. *Efficient Data Mining for Maximal Frequent Subtrees*. in *Proceedings of the Third IEEE International Conference on Data Mining*. 2003. IEEE Computer Society.
153. Termier, A., M.-C. Rousset, and M. Sebag. *Treefinder: a first step towards xml data mining*. in *In the Proceedings of IEEE International Conference on Data Mining, 2002 (ICDM'03)*. 2002. IEEE.
154. Hadzic, F., et al., *Mining of data with complex structures*. Vol. 333. 2011: Springer.
155. Chowdhury, I.J. and R. Nayak. *A Novel Method for Finding Similarities between Unordered Trees Using Matrix Data Model*. in *WISE 14th International Conference on Web Information Systems Engineering*. 2013. Springer Berlin Heidelberg.
156. Chowdhury, I.J. and R. Nayak. *Identifying product families using data mining techniques in manufacturing paradigm*. in *12th Australasian Data Mining Conference (AusDM 2013)*. 2014. Conferences in Research and Practice in Information Technology, Australian Computer Society.
157. Baybars, I., *A survey of exact algorithms for the simple assembly line balancing problem.* Management science, 1986. **32**(8): p. 909-932.
158. Sotskov, Y.N., et al., *Stability of optimal line balance with given station set*, in *Supply Chain Optimisation*. 2005, Springer. p. 135-149.
159. Ghoniem, M., J. Fekete, and P. Castagliola. *A comparison of the readability of graphs using node-link and matrix-based representations*. in *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. 2004. IEEE.

160. Rosen, K., *Discrete Mathematics and Its Applications 7th edition*. 2011: McGraw-Hill Science.
161. Kutty, S., et al., *Clustering XML Documents Using Frequent Subtrees*, in *Advances in Focused Retrieval*, S. Geva, J. Kamps, and A. Trotman, Editors. 2009, Springer Berlin Heidelberg. p. 436-445.
162. Kutty, S., et al., *Clustering XML documents using closed frequent subtrees: A structural similarity approach*, in *Focused Access to XML Documents*. 2008, Springer. p. 183-194.
163. Algergawy, A., et al., *XML data clustering: An overview.* ACM Computing Surveys (CSUR), 2011. **43**(4): p. 25.
164. Karypis, G., *CLUTO—software for clustering high-dimensional datasets*. 2007, Karypis Lab:
165. Shen, Y. and B. Wang, *Clustering schemaless XML documents*, in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. 2003, Springer. p. 767-784.
166. Aggarwal, C.C., et al. *Xproj: a framework for projected structural clustering of xml documents*. in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007. ACM.
167. Shasha, D., et al., *Exact and approximate algorithms for unordered tree matching.* IEEE Transactions on Systems, Man and Cybernetics, 1994. **24**(4): p. 668-678.
168. Zhang, S. and J.T.L. Wang, *Discovering Frequent Agreement Subtrees from Phylogenetic Data.* IEEE Transactions on Knowledge and Data Engineering, 2008. **20**(1): p. 68-82.
169. Jiang, T., et al., *A General Edit Distance between RNA Structures.* Journal of Computational Biology, 2002. **9**(2): p. 371-88.
170. Nakamura, T. and E. Tomita, *Efficient algorithms for finding a maximum clique with maximum vertex weight*, in *Tech. Rep. UEC-TRCAS3*. 2005, The University of Electro-Communications: Japan.
171. Wee, T. and M. Magazine, *Assembly line balancing as generalized bin packing.* Operations Research Letters, 1982. **1**(2): p. 56-58.
172. Manning, C.D., P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Vol. 1. 2008: Cambridge university press Cambridge.
173. Utterback, J. and M. Meyer, *The product family and the dynamics of core capability.* Sloan Management Review, 1993. **34**: p. 29-47.
174. Sawhney, M.S., *Leveraged high-variety strategies: from portfolio thinking to platform thinking.* Journal of the Academy of Marketing Science, 1998. **26**(1): p. 54-61.
175. Harhalakis, G., A. Kinsey, and I. Minis. *Automated group technology code generation using PDES*. in *Third International Conference on Computer Integrated Manufacturing*. 1992. IEEE.
176. Marion, D., J. Rubinovich, and I. Ham, *Developing a group technology coding and classification scheme.* Industrial Engineering, 1986. **18**(7): p. 90-97.
177. Romanowski, C.J. and R. Nagi, *A data mining-based engineering design support system: a research agenda*, in *Data mining for design and manufacturing*. 2002, Kluwer Academic Publishers. p. 161-178.
178. Iyer, S. and R. Nagi, *Automated retrieval and ranking of similar parts in agile manufacturing.* IIE Transactions, 1997. **29**(10): p. 859-876.

179. Romanowski, C.J., R. Nagi, and M. Sudit, *Data mining in an engineering design environment: OR applications from graph matching.* Computers & Operations Research, 2006. **33**(11): p. 3150-3160.
180. Matías, J., et al., *Automatic generation of a bill of materials based on attribute patterns with variant specifications in a customer-oriented environment.* Journal of Materials Processing Technology, 2008. **199**(1): p. 431-436.
181. Kao, Y. and Y. Moon, *A unified group technology implementation using the backpropagation learning rule of neural networks.* Computers & Industrial Engineering, 1991. **20**(4): p. 425-437.
182. Lee-Post, A., *Part family identification using a simple genetic algorithm.* International Journal of Production Research, 2000. **38**(4): p. 793-810.
183. Chen, Y., et al. *Using artificial neural networks to develop a mechanism for functional feature-based reference design retrieval*. in *IEEE International Conference on Engineering Management* 2004. IEEE.
184. Liu, F., et al., *Research on product combinatorial design based on functional similarity.* International Journal of Design Engineering (IJDE), 2008. **1**(3): p. 333–356.
185. Clement, J., A. Coldrick, and J. Sari, *Manufacturing Data Structures; Building Foundations for Excellence with Bills of Materials and.*. 1992, New York: John Wiley & Sons, Inc.
186. Ye, X. and J.K. Gershenson, *Attribute-based clustering methodology for product family design.* Journal of Engineering Design, 2008. **19**(6): p. 571-586.
187. Fogarty, D.W., J.H. Blackstone, and T.R. Hoffmann, *Production & Inventory Management*. 1991: South-Western Publishing Company.
188. Valiente, *Algorithms on Trees and Graphs*. 2002, New York: Springer, Berlin Heidelberg.
189. Rasmussen, M. and G. Karypis, *gcluto: An interactive clustering, visualization, and analysis system*, in *UMN-CS TR-04*. 2004.
190. Goutte, C. and E. Gaussier, *A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation*, in *Advances in Information Retrieval*, D. Losada, Fernández-Luna, JuanM, Editor. 2005, Springer Berlin Heidelberg. p. 345-359.
191. Hegge, H.M.H. and J.C. Wortmann, *Generic bill-of-material: a new product model.* International Journal of Production Economics, 1991. **23**(1–3): p. 117-128.
192. Hadzic, F., H. Tan, and T.S. Dillon. *UNI3 - Efficient Algorithm for Mining Unordered Induced Subtrees Using TMG Candidate Generation*. in *Proceedings of the 1st IEEE Symposium on Computational Intelligence and Data Mining*. 2007. Honolulu, Hawaii.
193. Pei, J., et al., *Mining Access Patterns Efficiently from Web Logs*, in *Knowledge Discovery and Data Mining 2000*. 2000, Springer Berlin Heidelberg: Kyoto. p. 396-407.
194. Termier, A., M.-C. Rousset, and M. Sebag. *Treefinder: a first step towards xml data mining*. in *IEEE International Conference on Data Mining, 2002 (ICDM 2002)*. 2002. IEEE.
195. Zaki, M.J. *Efficiently mining frequent trees in a forest*. in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2002. Edmonton, Alberta, Canada: ACM.

196. Hein, J., et al., *On the complexity of comparing evolutionary trees.* Discrete Applied Mathematics, 1996. **71**(1): p. 153-169.
197. Cui, J.-H., et al., *Aggregated multicast–a comparative study*. Cluster Computing, 2005. **8**(1): p. 15-26.
198. Comai, S., E. Damiani, and L. Tanca. *Flexible Queries to Semistructured Datasources: The WG-log Approach*. in *IIA/SOCO*. 1999. Citeseer.