ECTREE: AN EXTENDED TREE INDEX STRUCTURE FOR ATTRIBUTED SUBGRAPH QUERIES

Jun Luo

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science(Software Engineering)
Concordia University
Montréal, Québec, Canada

May 2012

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	Jun Luo
Entitled:	ECTree: An Extended Tree Index Structure for Attributed Subgraph Queries

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science(Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. B. Jaumard

______________________________ Examiner
Dr. T. Eavis

______________________________ Examiner
Dr. N. Shiri

______________________________ Supervisor
Dr. G. Butler

Approved ______________________________
Chair of Department or Graduate Program Director

__________ 20 _____ ______________________________
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

ECTree: An Extended Tree Index Structure for Attributed Subgraph Queries

Jun Luo

Graphs are popular data structures for modeling complex data types, especially graphs with attributes for gene sequences, protein structures, chemical compounds, protein interaction networks, social networks, etc. There is a need for managing such graph data and providing efficient querying tools. In the graph mining realm, the problem lies in indexing a large number of graphs for fast retrieval. Indexing attributed graphs and using attributed queries can provide faster response time and more refined results.

This thesis focuses on extending an existing index to support attributed graph indexing and providing subgraph querying access to the extended index. The aim is to find a way such that the labels of the graphs as well as the attributes of the graphs are indexed at the same time. A query format is provided to query the extended index on the attributes with flexibility which allows intervals to be used. In addition, regular expressions and label groups are used as query labels so that multiple queries that have similar structures can be combined as a single query. This also benefits in that a query graph does not have to use fixed labels. We also introduce a vertex degree-attribute based vector to capture both the features of a data graph and a query graph. A novel pruning method is proposed and implemented so that the pruning based on the degree-attribute vectors can still be adopted even when it is not clear how to define a histogram pruning for the query graphs that use non-fixed labels. All the techniques presented in our work are validated through experiments on both real and synthetic datasets.

Dedication

The completion of this thesis would not have been possible without the help of a lot of people. It is all those people who offered support when I was in need that bring me this far. I would like to express my most sincere gratitude to them.

First and foremost, I want to thank my supervisor Dr. Gregory Butler for his support, guidance and encouragement, as well as his knowledge and patience, which allowed me to see a clearer picture of my study and to avoid costly mistakes throughout my Master's program.

I offer my deep appreciation to the members of my advisory committee: Dr. Brigitte Jaumard, Dr. Nematollaah Shiri and Dr. Todd Eavis.

I also give special thanks to my wonderful colleagues: Dr. Stephen C. Barrett for his help, with whom I learned what research really means, and also his demonstration of how to write a "beautiful" thesis; Dr. Yue Wang for her help in leading me into the graph mining realm; Faizh Aplop who always helps me discuss and solve the difficulties that I have in my study.

Moreover, I am truly thankful to my friend Dayal Dasanayake who helped me correct the thesis, as well as his persistent support on my way to obtain a Master's degree; My grateful thanks to my uncle Guohui Zheng and my aunt Ying Zhao for taking care of me in every possible way in life; I thank my good friend Qinghuan Li, who always tells me I will make it and cheers me up whenever I feel down.

Last but not least, I give my deepest gratitude to my parents for their continuous encouragement and unquestioning love.

Contents

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

1.1 Graph Data 2

1.2 The Problem 3

1.3 Contributions 3

1.4 Thesis Organization 4

2 Background **5**

2.1 Graph Databases 5

2.2 Subgraph Query and Matching 6

2.2.1 Subgraph and Subgraph Query 6

2.2.2 Graph Isomorphism and Subgraph Isomorphism 7

2.3 The Filtering-and-Verification Framework 9

2.3.1 The False-Positive Rate in Candidate Answer Set 11

2.4 Dead Space in Graph Merging 11

2.5 Closure-Tree Index 13

2.5.1 Graph Representation 13

2.5.2 The Graph-Closure Method 14

2.5.3 Building the CTree and Querying 16

2.5.4 Histogram-Based Pruning in CTree 18

2.6 Regular Expressions 20

3 ECTree: the Extended Closure-Tree for Subgraph Queries 22

3.1 Extending the Vertices with Attributes 22

3.1.1 An Example in the SDF Format 23

3.1.2 Data Representation in ECTree 24

3.2 Vertex Merging 25

3.2.1 Merging Two Vertices 26

3.2.2 Merging A Vertex and A Vertex-Closure 28

3.2.3 Merging Two Vertex-Closures 29

3.3 Building the ECTree 30

3.3.1 Data Graph Mapping 30

3.3.2 An ECTree Building Example 31

3.4 A Query Graph Format for ECTree 31

3.5 Query Matching 34

3.5.1 Matching a Query Vertex and a Data Vertex 34

3.5.2 Matching a Query Graph and a Data Graph 36

3.6 Conclusion 38

4 Optimizations for ECTree 39

4.1 A More Flexible Query Format 39

4.1.1 Using Regular Expressions on Query Labels 40

4.1.2 Label Groups 42

4.1.3 Implementation 44

4.2 A Novel Pruning Method for ECTree 46

4.2.1 The Degree-Attribute Feature Vector 46

4.2.2 The Pruning Strategy 49

4.3 Conclusion 54

5 Validation 55

5.1 Experiment Setup 55

5.2 Effectiveness of the Index 58

5.3 The New Query Label 60

5.4 The Pruning Power 62

6 Related Work 68

7 Conclusion **72**

7.1 Summary of Contributions 72

7.2 Limitations 73

7.3 Future Work 74

Bibliography **75**

Appendices **82**

A Meaning of Values in SDF Atom Block **83**

B Related Algorithms **85**

B.1 Counting Different Labels and Finding New Bindings 85

B.2 Adding Attributes to Non-Attributed Graphs 87

B.3 Generating Query Sets for Synthetic Datasets 88

List of Figures

1 Varieties of graph data . . . 2
2 An example of a graph and its subgraph . . . 7
3 Two isomorphic graphs . . . 8
4 The filtering-and-verification framework as a pipe and filter system . . . 10
5 An example showing the dead space . . . 12
6 The graph data and query format in CTree . . . 13
7 Computing a graph-clousre . . . 15
8 A CTree . . . 16
9 The label pruning in CTree . . . 20
10 Alanine in SDF format . . . 23
11 Data structure of ECTree . . . 25
12 Merging two label-matched vertices . . . 26
13 Merging two non-label-matched vertex . . . 28
14 Merging a vertex and a vertex-closure . . . 29
15 Merging two vertex-closures . . . 30
16 Computing a graph-closure in ECTree . . . 32
17 Examples of interval sets . . . 33
18 Matching a query vertex and a non-closure data vertex . . . 34
19 Two intervals showing the concept of Minimum-Absolute values . . . 35
20 Two chemical structures as queries . . . 41
21 The merged query Q_3 . . . 42
22 A query Q_4 showing the label groups . . . 43
23 A matching example for label groups . . . 44
24 DA feature vectors for data graphs . . . 48
25 DA feature vectors for query graphs . . . 49

26 A graph and its supergraph 50
27 The DA-Pruning examples 52
28 The testing process as a pipe-and-filtering framework 57
29 Filtering time cost 59
30 NCI subset verification statistics using CTree and ECTree 60
31 Overall time cost 61
32 Four queries and a combined query 62
33 DA-Pruning on different attribute densities 63
34 Overall time cost enhance rate using DA(d)-Pruning 64
35 DA-Pruning on different maximum attribute values 66

List of Tables

1 Graph matching comparison using different matching strategies 12
2 Notations 37
3 Feature vector comparison 51
4 Query sets statistics 57
5 Time cost to build index 58
6 Comparison of querying methods 1 61
7 Comparison of querying methods 2 65
8 Enhancement rates using DA(d)-Pruning on synthetic datasets 67
9 Meaning of values in the atom block [Sym] 84

Chapter 1

Introduction

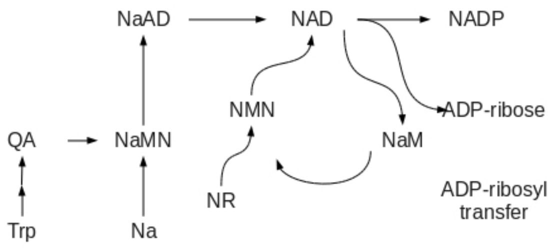
Along with the improvement of technology in biology and its related realms comes large volumes of different kinds of data. The Human Genome Project (HGP) is one of the projects with a primary goal of determining the sequence of chemical base pairs which make up DNA and to identify and map the approximately 20,000 to 25,000 genes of the human genome [Gen] [Kru01]. Each gene contains a long sequence made up of hundreds or even thousands of 'genetic words', which means the HGP also brings about a fast growing need for the analysis of biological data. The focus of recent biology related realms has shifted from the technology itself, via generating data, to the management of the huge amount of data. By management we mean collecting the data into databases, organizing them according to needs and finally finding the information needed from the databases.

There are multiple types of data that are involved in biology-related realms: gene sequences, protein structures, chemical compounds, protein interaction networks, etc. They naturally have the basic features of a graph: a set of vertices and a set of edges, and thus can be easily modeled as graphs. More broadly speaking, graphs have been widely used to model various types of data, such as flight networks [AEP09], XML data and queries [FGMP09], and social networks [BK09]. No matter what type it is, since graphs form a complex and expressive data structure, we need efficient and effective methods for representing graphs in databases, manipulating and querying them [AW10]. A key issue is that the data management software needs to be easy–to–use, yet provides fast response time [BWWZ06].

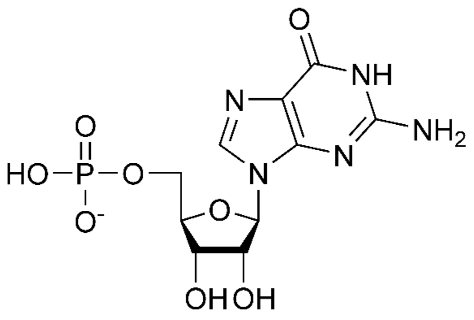
1.1 Graph Data

In bioinformatics and cheminformatics, graphs have been used to represent complex data types [HS06], such as protein structures (Figure 1(a)), metabolic pathways (Figure 1(b)), chemical compounds (Figure 1(c)) and gene structures (Figure 1(d)).

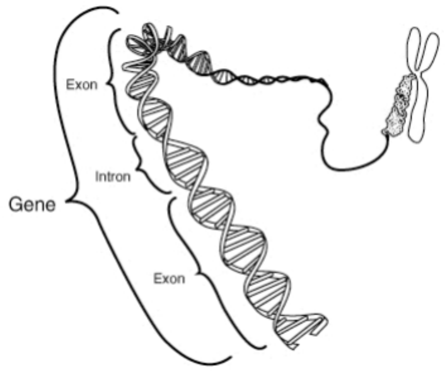
(a) Protein structure of S. pombe pop2p deadenylation subunit [Deb]



(b) NAD metabolism pathway [Vic]



(c) Guanosine monophosphate [Cac]



(d) Gene [Rav]

Figure 1: Varieties of graph data

The graphs that model different types of data can have different meanings on the vertices and edges. For example, a metabolic pathway is modeled as a set of enzymes, chemicals (also called metabolites) and reactions, where the edges are the connections between metabolites. The vertices for a single chemical compound could be the atoms in the compound, and the edges are the bonds between the atoms.

1.2 The Problem

There has been a lot of work related to graph database management. We review some of the major and/or latest works in Chapter 6. By studying varieties of graph data indexes that are effective and efficient for the graph queries, we find that most graph database indexes focus only on vertex-labeled and/or edge-labeled graphs, while in general, there has been lack of studies on graph data indexes which can handle vertices with attributes on index building, querying, as well as studies on flexible query formats and corresponding optimization strategies.

Although the essential components of a graph are the vertices and the edges, in many cases we can find that a graph may contain more information on the vertices and the edges. For example, the transformation between two metabolites may have certain conditions; an atom may have isotopes and a charge on it. This thesis aims at developing an effective graph data index to organize small graphs with both label indexing and attribute indexing, and to provide graph queries that have extended features such as attributes on vertices. A small graph is defined as a graph with less than 100 nodes [BV99]. In this thesis, we define it as a graph with less than 15 nodes.

In previous work, the query graphs have fixed labels which allows no flexibility. Querying two similar graphs would be more efficient if there is a way to combine those queries into one query. Thus we aim at providing a flexible query format for our new index and developing a pruning method for the new query format as well.

1.3 Contributions

The contribution of this thesis can be classified as follows:

- Study of merging numerical attributed graphs and extension of the tree index structure for additional attributes. We implemented the CTree index in C++ and extended the index to support numerical attributes in both indexing and querying. Our approach reveals that the indexing on both labels and attributes is faster in terms of querying response time than indexing on only labels for attributed graphs.

- A query format to query the extended index with flexibility on the attribute part, with non-fixed query labels supported which uses regular expression. The new query format also allows queries with similar graph structure to be combined into a single query.

- A pruning method that works when the vertex labels are non-fixed or unknown. The new pruning is based on the degree and attribute information of vertices on data and query graphs. We proposed the method and implemented it to test its pruning power.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces the background knowledge necessary for understanding the work in this thesis. Chapter 3 illustrates the extension of the Closure-Tree [HS06] graph index. Chapter 4 describes a query format combined with regular expression and a pruning method based on the vertex degree and attributes. Chapter 5 provides validations of our work in Chapters 3 and 4. Chapter 6 discusses the related work to this thesis. Chapter 7 gives a conclusion and discussions for the future work.

Chapter 2

Background

This chapter gives the background knowledge which is necessary to understand the described problem and related solutions in the following chapters.

2.1 Graph Databases

In past decades, the focus of the genome project has shifted from technology development to data management. As more data is generated, data analysis becomes more essential. A graph database supports data analysis with appropriate database management [GBL95].

By definition, a graph database model is a model such that data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors [AG08]. An SQL database [Mel96] addresses a many-to-many relationship with a join table that could be huge when the relationship is complex and might lead to unwieldy, long, incomprehensible SQL statements as well as unpredictable performance [Wig]. Comparing to relational databases, a graph database is designed to represent and query this type of information, so it models the data more naturally.

The essence of a graph database lies in what qualifies as a node, and what qualifies as an edge [Wan10]. A typical graph data model allows simple labels on the nodes. The labels are usually numbers or strings and are the main recognition of the nodes. Still, a node can contain some other attributes which are encoded by means of "additional attributes" that are data values of various types. For example, a person has a name, as well as other information such as sex, age, nationality, and so on.

A graph database may contain many small graphs, such as a graph database for chemical compounds [NCI], or, a graph database can contain a small number of large complex graphs. For example, each of the Carbohydrate Metabolism pathway dataset [KEG] could be a large graph that contains hundreds of metabolites and connections. Thus, the main purpose of the graph databases is to find a proper way to model the graph data so as to better serve the consequential graph data mining phase.

2.2 Subgraph Query and Matching

2.2.1 Subgraph and Subgraph Query

A commonly asked type of question mentioned in [CG70] is to find whether a given chemical compound is a subcompound of a further specified compound, given the structural formulas. In graph theory, this type of problem is generalized as to find whether a given graph is a subgraph of another graph. We denote a graph G as $G(V, E)$ where V is the vertex set and E is the edge set. The formal definition for a **Subgraph** is given here [BL]:

Definition 1 Subgraph

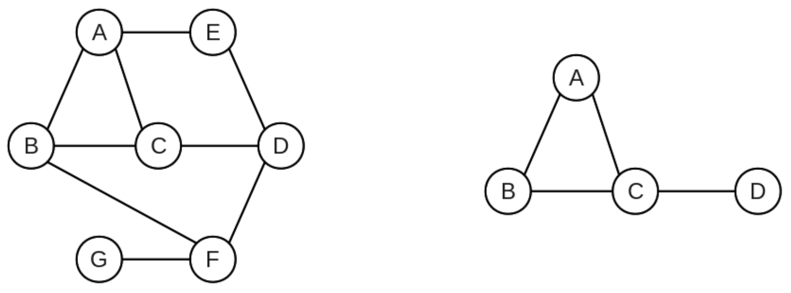
A graph $G'(V', E')$ is a subgraph of another graph $G(V, E)$ if and only if

- $V' \subseteq V$ and
- $E' \subseteq E$

According to **Definition 1**, we can see that a graph $G(V, E)$ is also a subgraph of itself.

The task of a **Subgraph Query** is to find graphs which contain a specified graph as a subgraph. Subgraph queries are widely applied for graph data mining and are important in bioinfomatics and cheminfomatics because scientists frequently ask questions such as: is a specified metabolism pathway contained in this set of metabolism pathways? Or, how many chemical compounds that develops potential cancer contain this particular chemical substructure? A considerable amount of research has been done into finding a subgraph in a set of graphs of a graph database. Some recent researches on subgraph mining algorithms can be found in [PLM08], [WC10] and [FB08].

Figure 2 gives a simple example showing a graph G_1 (Figure 2(a)) and one of its subgraphs G_2 (Figure 2(b)). If graph G_2 is used as a subgraph query and graph G_1 is a graph in the database being queried, then G_1 should be returned as one of the query results.



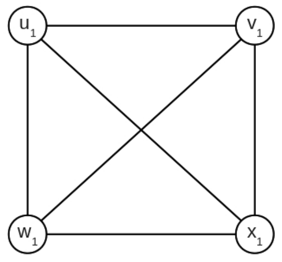
(a) G_1 (b) G_2

Figure 2: An example of a graph and its subgraph

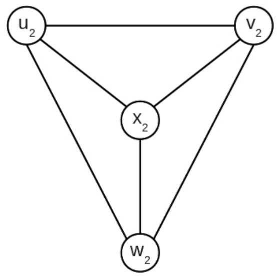
2.2.2 Graph Isomorphism and Subgraph Isomorphism

Two graphs, G_1 and G_2, are the same if it is possible to redraw one of them, say G_2, so it appears identical to G_1. In graph theory, the term **isomorphic** is used to describe graphs that are the same but are models of different situations. If two graphs G_1 and G_2 are equivalent graphs, they are referred as **isomorphic graphs** [Cha85]. The process of graph matching is to determine whether two graphs are isomorphic or not. For instance, the graph G_1 and G_2 in Figure 3 (extracted from [Cha85]) are isomorphic. If two graphs G_1 and G_2 are isomorphic, we say G_1 is isomorphic to G_2 and that G_2 is isomorphic to G_1.

Attributes of vertices are denoted by $attr(v)$. The definition for **Graph Mapping**, **Graph Isomorphism** and **Isomorphic Graphs** are given below [HS06]



(a) G_1



(b) G_2

Figure 3: Two isomorphic graphs

[Cha85]:

Definition 2 Graph Mapping

A mapping between two graphs G_1 and G_2 is a bijection $\phi : G_1 \rightarrow G_2$, where (i) $\forall v \in V_1$, $\phi(v) \in V_2$, and (ii) $\forall e = (v_1, v_2) \in E_1$, $\phi(e) = (\phi(v_1), \phi(v_2)) \in E_2$.

Definition 3 Graph Isomorphism and Isomorphic Graphs

A graph isomorphism from G_1 to G_2 is a graph mapping ϕ from G_1 to G_2 such that (i) $\forall v \in V_1, \phi(v) \in V_2$ and $attr(v) = attr(\phi(v))$ and (ii) $\forall e \in E_1, \phi(e) \in E_2$ and $attr(e) = attr(\phi(e))$. Two graphs G_1 and G_2 are isomorphic if a graph isomorphism exists from G_1 to G_2. Graph isomorphism is symmetric.

Given the definition of isomorphic graphs, the definition of **Subgraph Isomorphic** and **Subgraph Isomorphism** can then be given as follows.

Definition 4 Subgraph Isomorphic and Subgraph Isomorphism

A graph G_1 is subgraph isomorphic to another graph G_2 if G_1 is isomorphic to a subgraph of G_2. A subgraph isomorphism exists between G_1 and G_2 if G_1 is subgraph isomorphic to G_2. Subgraph isomorphism is not symmetric.

The subgraph isomorphism problem has long been proved to be NP-complete using a reduction from 3-SAT involving cliques [Coo71]. There has been a lot of research into finding a fast algorithm for subgraph isomorphism decision. Early research in [Ull76] proposed exact subgraph isomorphism algorithms which are devised for both graph isomorphism and subgraph isomorphism and are still some of the most commonly used algorithms for exact graph matching today. (The subgraph isomorphism algorithm described in [Ull76] is used in this thesis.) [Epp95] proposed a linear time method to solve the subgraph isomorphism problem in planar graphs for any pattern of constant size. [CFSV04] demonstrated an algorithm, namely VF2, for isomorphism problems when matching large graphs. A state-of-the-art subgraph isomorphism algorithm called QuickSI is introduced in [SZLY08].

Having the definition of subgraph isomorphism, when referred to a subgraph of a graph, we can simply say: a graph $G(V, E)$ is a subgraph of another graph $G'(V', E')$ if G is subgraph isomorphic to G' under graph mapping ϕ.

In this thesis, $SubgraphIsomorphism(G_1, G_2)$ is used to refer to the subgraph isomorphism test to determine whether or not a graph G_1 is a subgraph of another graph G_2.

2.3 The Filtering-and-Verification Framework

Though much research has been done into accelerating the process of subgraph isomorphism test, the time cost for this process is still heavy. Thus, a direct comparison of a query graph to each of the graphs in the dataset using a subgraph isomorphism algorithm is very inefficient. Different graph indexes for graph databases are used to decrease the number of subgraph isomorphism tests for graph queries and therefore gain a better performance in terms of time cost.

Most graph database indexes follow a common framework called **Filtering-and-Verification** [ZCZ+08], as shown in Figure 4.

The filtering step firstly builds an index using the graphs in the dataset. Then the index is used to eliminate some false results (usually most of the false results) and produces a candidate set. The last step is to use the expensive subgraph isomorphism test to verify the candidate set and obtain the final result set [CKNL07]. Since the candidate set is much smaller than the original dataset, it is more efficient to use an

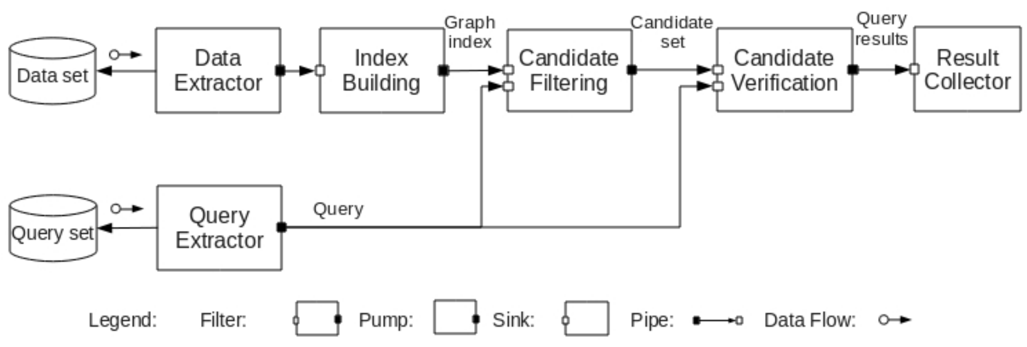


Figure 4: The filtering-and-verification framework as a pipe and filter system

index than naive sequential scan.

There has been a lot of research about graph indexes using the filtering-and-verification framework. [HLPY10] categorized them into two main types depending on whether they use frequent graph mining. A frequent graph mining algorithm is an algorithm that discovers all subgraphs which occur frequently in the database, with the motivation that these are the subgraphs with a statistically significant amount of occurrences in the database [NK04].

Mining Based Approaches is the first category. The main process of building the index in this category is to extract subgraphs as features in order to obtain a set of feature graphs. Each feature graph is associated with a list of graph IDs which contain this feature graph as a subgraph. In the query process, the candidate graphs are obtained by finding features associated with the posting lists and intersecting the lists. A subgraph isomorphism test is used to refine the candidates to get the final result [HLPY10].

Non-Mining Based Approaches is the second category. Graph indexing and query processing methods in this category do not share the same features. GraphGrep [SWG02], Summarization graph [ZCZ+08] and Closure-Tree [HS06] are in this category. We will introduce the Closure-Tree in Section 2.5 which forms a basis for our work.

2.3.1 The False-Positive Rate in Candidate Answer Set

In the filtering phase, the false-positive rate is the probability of categorizing a negative result into the candidate set. Statistically, assume S_C is the candidate answer set for a query Q after filtering, the result set S_R is obtained after verifying Q with S_C, we have the false-positive rate $FP = \frac{|S_C|-|S_R|}{|S_C|}$. In contrast, the true-positive rate $TP = \frac{|S_R|}{|S_C|}$. The false-positive rate can be reduced by selecting some proper filtering methods, thus to reduce the size of the candidate answer set to speed up the verification process.

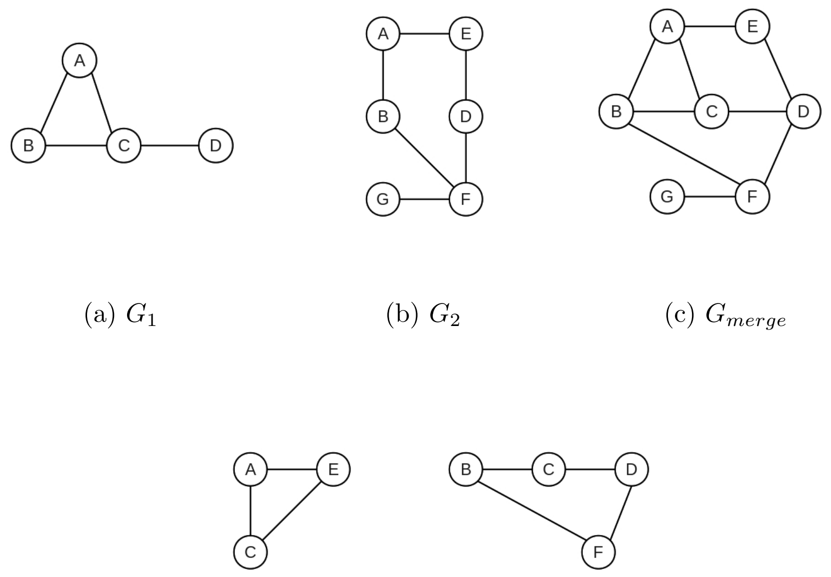
2.4 Dead Space in Graph Merging

Graph merging is an operation for a set of graphs, say S, to merge them into a single graph G_{Merge}, in a way that all the graphs in S are subgraphs of the generated graph G_{Merge}: $\forall g \in S, g$ is subgraph-isomorphic to G_{Merge}.

Graph merging has an obvious feature for subgraph mining: if a query graph G_Q is a subgraph of one of the graphs in the set S, say G_1, then G_Q is a subgraph of the merged graph G_{Merge}. This is equal saying that, if G_Q is not a subgraph of the merged graph G_{Merge}, then G_Q is not a subgraph of any of the graphs in the set S. This kind of organization of graphs benefits the subgraph mining process because if G_Q is not a subgraph of G_{Merge}, then no further test is needed to examine each of the graphs in S, and thus the execution times of the expensive subgraph-isomorphism test is reduced to 1 from up to $|S|$.

To use the benefits of graph merging there is a price to be paid in terms of **Dead Space**. The dead space of a merged graph G_{merge} is the space inside G_{merge} but contains no graphs that are in the graph set S [HLPY10]. We use an example in Figure 5 to explain the concept of dead space. G_{merge} can be obtained by merging G_1 (Figure 5(a)) and G_2 (Figure 5(b)). It is obvious that G_{merge} satisfies the condition that both G_1 and G_2 are its subgraphs. There are two simple query graphs G_{Q1}(Figure 5(d)) and G_{Q2}(Figure 5(e)), both of which are not subgraphs of any of the graphs in S. Table 1 shows a comparison of numbers of subgraph isomorphism tests executed using G_1 and G_2 as query graphs in different graph matching strategies.

Query G_{Q1} benefits from the graph merging pre-process, but for G_{Q2}, the number



(a) G_1 (b) G_2 (c) G_{merge}

(d) G_{Q1} (e) G_{Q2}

Figure 5: An example showing the dead space

of subgraph-isomorphism test is even larger than naive sequential scan of the dataset. Because G_{Q2} is a subgraph of G_{merge} but is not a subgraph of G_1 or G_2, thus we consider G_{Q2} to be a graph in the dead space of G_{merge}. There are many more sample graphs that are in the dead space of G_{merge} which make the merged graph somehow inefficient. The larger the dead space of G_{merge} is, the less graph matching can benefit from graph merging.

The solution to reducing the dead space is to merge only the graphs that are similar in terms of vertices, edges and the structures. The definition for the term

Query Graph	Naive Sequential Scan	Graph Merging Pre-process
G_{Q1}	2	1
G_{Q2}	2	3

Table 1: Graph matching comparison using different matching strategies

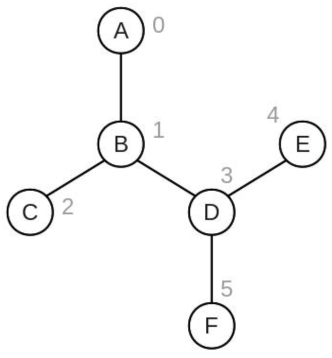
Graph Similarity as well as related calculation methods are given in [He07].

2.5 Closure-Tree Index

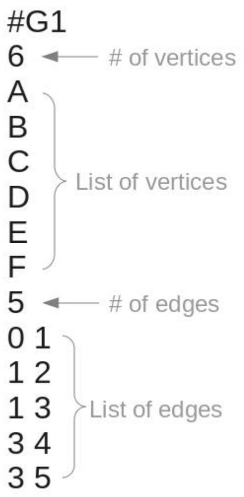
In this section, we introduce the previous work in [HS06].

2.5.1 Graph Representation

The Closure-Tree (CTree) index is a graph database index for a large number of small graphs which adopts a plain text format for representations of both data graphs and query graphs. Figure 6(a) shows a simple graph G_1 that contains six vertices and five edges. Figure 6(b) shows the plain text format representation of graph G_1.



(a) A simple graph G_1



```
#G1
6
A
B
C
D
E
F
5
0 1
1 2
1 3
3 4
3 5
```

# of vertices
List of vertices
# of edges
List of edges

(b) Plain text format of G_1

Figure 6: The graph data and query format in CTree

Each graph entry starts with a '#', and is followed by the components: *graph name, number of vertices, list of vertex labels, number of edges, list of edges.* The vertices are numbered starting from 0 in the internal representation, which correspond to the numbers in the edge list. The labels of the vertices are limited to alphabetic letters. In the edge list, the order of the two numbers appearing in a line shows the direction of an edge. However, throughout the thesis, we use only undirected graphs for both data and query graphs, thus each line in the edge list is recognized as an undirected edge.

An undirected graph G is denoted as $G(V, E)$, where V is the vertex set and E is the edge set.

2.5.2 The Graph-Closure Method

In the CTree index, every tree node is a graph or a special kind of graph which is called a graph-closure. Every CTree node contains structural information of all its descendants. *Graph-Closure* is introduced to capture the structural features of a set of graphs or graph-closures. [HS06] gives the concepts of vertex-closure, edge-closure and a new definition of graph mapping below:

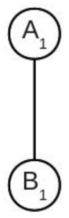
Definition 5 Vertex Closure and Edge Closure
The closure of a set of vertices is a generalized vertex whose attribute is the union of the attribute values of the vertices. Likewise, the closure of a set of edges is a generalized edge whose attribute is the union of the attribute values of the edges.

To ensure each vertex and edge have a corresponding element in the mapped graph, dummy vertices/edges are introduced and have a special label ε as their attribute.

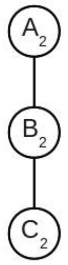
Definition 6 Graph Mapping
A mapping between two graphs G_1 and G_2 is a bijection $\phi : G_1 \to G_2$, where (i) $\forall v \in V_1$, $\phi(v) \in V_2$, and at least one of v and $\phi(v)$ is not dummy, and (ii) $\forall e = (v_1, v_2) \in E_1, \phi(e) = (\phi(v_1), \phi(v_2)) \in E_2$, and at least one of e and $\phi(e)$ is not dummy.

Figure 7 demonstrates an example of constructing a simple CTree. The subscript of each label is used to differentiate vertices in different graphs.

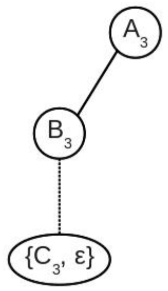
Firstly, Closure 1 is computed from Graph 1 and Graph 2. Note that vertex A, vertex B and edge (A, B) are common structures between these two graphs. Vertex C and edge (B, C) in Graph 2 are not common structures. Vertex C is mapped to a dummy vertex ε and put into a vertex closure $\{C, \varepsilon\}$. Edge (B, C) is mapped to



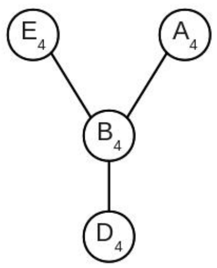
(a) Graph 1



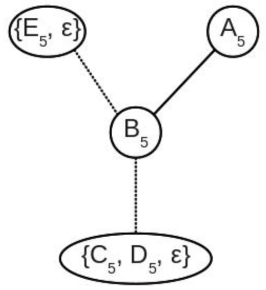
(b) Graph 2



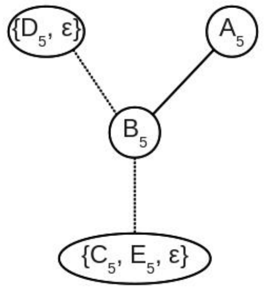
(c) Closure 1



(d) Graph 3



(e) Closure 2a



(f) Closure 2b

Figure 7: Computing a graph-clousre

a dummy edge ε and put into an edge closure $\{(B,C),\varepsilon\}$. A dotted line implies an edge-closure. Dummy vertices are used so that every vertex has a corresponding element in the other graph.

Next, in Figure 7(e), Closure 2a is computed from Closure 1 and Graph 3 using the same method. Here the concept of a graph-closure under mapping ϕ is introduced [HS06].

Definition 7 Graph Closure under Mapping ϕ
The closure of two graphs G_1 and G_2 under a mapping ϕ is a generalized graph (V,E) where V is the set of vertex closures of the corresponding vertices and E is the set of edge closures of the corresponding edges. This

is denoted by $closure(G_1, G_2)$.

In Figure 7(d), vertex E_4 is mapped to ε, vertex D_4 is mapped to $\{C, \varepsilon\}$ in (c). If D_4 is mapped to ε, then the final constructed graph-closure will be (f). [HS06] developed a graph mapping method called $Neighbor\ Biased\ Mapping$ (NBM) which maximizes the common structures between two graphs. After NBM, the obtained mapping can be used to merge two graphs, say G_1 and G_2, into a new generalized graph G_3. The process is denoted as $G_3 = Closure(G_1, G_2)$.

In the plain text representation format, a graph-closure always has an ID *null*. A vertex-closure or an edge-closure has *null* following the vertex label or edge numbers indicating that it is a closure. Following the edge list, a graph-closure further has one line indicating how many entries of graphs/graph-closures it has.

2.5.3 Building the CTree and Querying

The graph-closures for a graph dataset are computed recursively level by level. The graph-closure on the top level contains the structural information of all the graphs in the dataset and is the root of this CTree. The constructed tree structure is called a *Closure-Tree* (CTree). The CTree building algorithm is shown in Algorithm 1 [HS06].

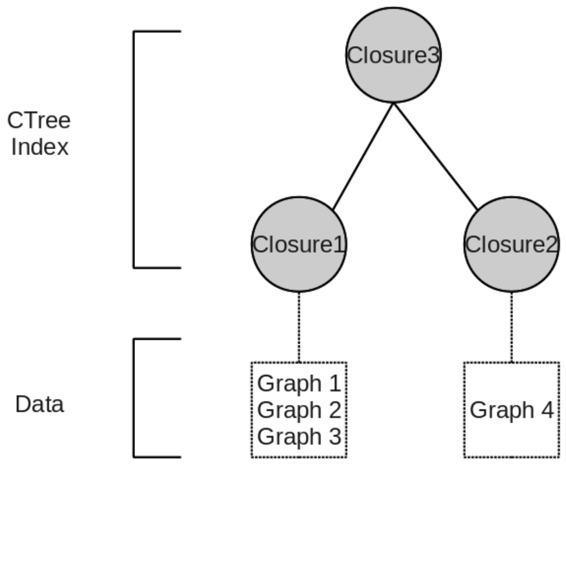


Figure 8: A CTree

Algorithm 1: BuildCTree(S, M)

```
input  : A set of graphs $S$, the maximum number of entries for each
         graph-closure $M$
output: A CTree $tree$
$ClosureSet \leftarrow Partition(S)$ ;
if $|ClosureSet| = 1$ then
    $tree$.Root $\leftarrow ClosureSet[1]$
else
    $tree$.Root $\leftarrow MakeClosure(ClosureSet)$ ;
return ctree;

Function $\textbf{\textit{Partition}}(S)$
input : A set of graphs $S$
output : A set of graph-closures $C$
if $|S| \leqslant M$ then
    return $MakeClosure(S)$;
else
    $S_1 \leftarrow \{S[i] \mid i \in [1, \lfloor \frac{|S|}{2} \rfloor ] \}$ ;
    $S_2 \leftarrow \{S[i] \mid i \in [\lfloor \frac{|S|}{2} \rfloor + 1, |S|] \}$ ;
    $C_1 \leftarrow partition(S_1)$ ;
    $C_2 \leftarrow partition(S_2)$ ;
    if $|C_1| + |C_2| \leqslant M$ then
        return $C_1 \cup C_2$ ;
    else
        $C_1 \leftarrow MakeClosure(C_1)$ ;
        $C_2 \leftarrow MakeClosure(C_2)$ ;
        return $C_1 \cup C_2$;

Function $\textbf{\textit{MakeClosure}}(C)$
input : A set of graphs $C$
output : A graph-closure $G$
if $|C| = 1$ then
    $G \leftarrow C[1]$ ;
    return $G$ ;
else
    $G \leftarrow C[1]$;
    for $i \leftarrow 2$ to $|C|$ do
        $G \leftarrow Closure(G, C[i])$ ;
    return $G$;
```

Figure 8 shows a CTree computed from the graphs in Figure 7. *Closure*3 is the root of this CTree which has a depth of 2. Each graph-closure is a node in the CTree, and each leaf CTree node (*Closure*1, *Closure*2) contains graph entries only, while each non-leaf CTree node (*Closure*3) only contains CTreeNode entries. In order to maintain that only leaf CTreeNodes contain graph entries, *Graph*4 is firstly transformed into a graph-closure and then participates in further computing.

In order to distinguish the use of 'vertex' and 'node', the term 'node' is used when we refer to the tree index, The word 'vertex' is used when we refer to the graphs in a dataset.

In the graph query process, if a CTree node fails a sub-graph isomorphism test with the query graph, then no children nodes need to be further tested, and thus this node and all its descendants are pruned. If the pruning happens at a relatively higher level of the index tree, then more descendant nodes are likely to be pruned and therefore reduce the query time significantly. CTree uses a *Pseudo Subgraph Isomorphism* test to prune unwanted nodes and uses the subgraph isomorphism test of [Ull76] to further filter the candidate graph list. The query graphs have the same structure and format as the data graphs. The query process is shown in Algorithm 2 from [HS06].

2.5.4 Histogram-Based Pruning in CTree

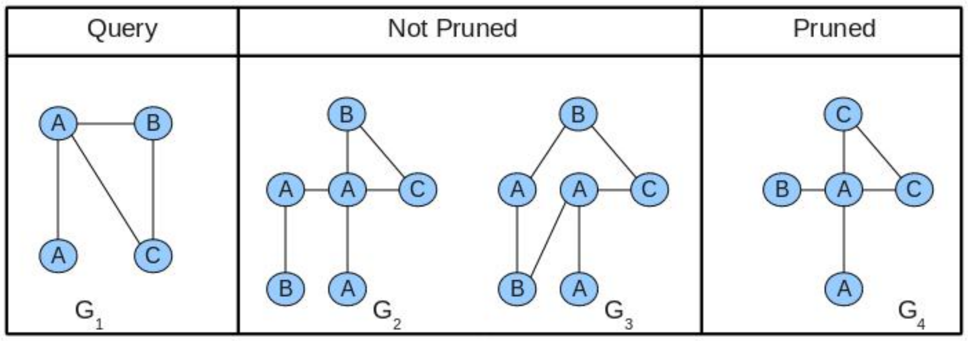
In CTree, a histogram-based method is used as a simple pruning before the structural pruning *Pseudo-Subgraph Isomorphism* test [HS06]. The pruning starts by calculating the histogram-feature of each graph in the dataset. Assume a query graph G_1 and a data graph G_2 need to be tested. The pruning proceeds in the following steps:

1. Record the number of appearance of each different vertex labels in an array T_L to get $T_L(G_1)$ and $T_L(G_2)$. Also record the number of appearance of edges in an array T_E to get $T_E(G_1)$ and $T_E(G_2)$. Note that $|T_L(G_1)| = |T_L(G_2)| = |T_L|$, and $|T_E(G_1)| = |T_E(G_2)| = |T_E|$.
2. if $\exists i \in [1, |T_L(G_1)|]$, such that $T_L(G_1)[i] > T_L(G_2)[i]$, prune G_2.
3. if $\exists i \in [1, |T_E(G_1)|]$, such that $T_E(G_1)[i] > T_E(G_2)[i]$, prune G_2.

This pruning can also be applied to prune graph-closures because a graph-closure is a generalized graph.

```
Algorithm 2: QueryCTree($query$, $tree$)
    input  : A query graph $query$, a CTree $tree$
    output: A set of graphs that contain $query$ as a subgraph
    $CS \leftarrow Visit(query, tree.Root)$ ;
    $Ans \leftarrow empty$ ;
    foreach $G \in CS$ do if SubIsomorphic(query, G) then
        $Ans \leftarrow Ans \cup \{G\}$;
    return $Ans$ ;

    Function Visit($query$, $node$)
    input : A query graph $query$, a CTree node $node$
    output : A graph set $CS$
    $CS \leftarrow empty$;
    foreach child c of node do
        $G \leftarrow$ the graph or graph closure at $c$ ;
        if PseudoSubIsomorphic(query,G) then
            if G is a database graph then
                $CS \leftarrow CS \cup \{G\}$;
            else
                $CS \leftarrow CS \cup Visit(query, c)$;
    return $CS$;
```

A pruning example is shown in Figure 9. G_2 and G_3 both survived the pruning with G_1 as a query graph. But G_1 is not a subgraph of G_3 and G_3 is not pruned, which means the pruning is conservative. G_4 is pruned because $T_E(G_4)$ does not satisfy the requirements.



$T_L(G_1)$

A	2
B	1
C	1

$T_E(G_1)$

A-A	1
A-B	1
A-C	1
B-B	0
B-C	1
C-C	0

$T_L(G_2)$

A	3
B	2
C	1

$T_E(G_2)$

A-A	2
A-B	2
A-C	1
B-B	0
B-C	1
C-C	0

$T_L(G_3)$

A	3
B	2
C	1

$T_E(G_3)$

A-A	2
A-B	2
A-C	1
B-B	0
B-C	1
C-C	0

$T_L(G_4)$

A	2
B	1
C	2

$T_E(G_4)$

A-A	1
A-B	1
A-C	2
B-B	0
B-C	0
C-C	1

Figure 9: The label pruning in CTree

2.6 Regular Expressions

A regular expression is the term used to describe a codified method that provides a concise and flexible means for matching strings of text [ZYT]. It is usually used to give a concise description of a set of strings without having to list all elements. The IEEE POSIX [IEE] released the Basic Regular Expressions (BREs) along with an alternative standard called Extended Regular Expressions (EREs). BREs provided a common standard which is adopted as the default syntax of many Unix regular expression tools. Most of such tools also provide additional features.

The GNU C Library [GNU] provides regular expression tools which follow the POSIX.2 standard with support of EREs. We adopt this tool and use EREs in Section 4.1. Detailed introduction of regular expressions can be found in [Goo05] and [Fri97].

Chapter 3

ECTree: the Extended Closure-Tree for Subgraph Queries

Using a tree structure in a graph database index allows efficient subgraph structure mining. Section 2.5 introduced Closure-Tree (CTree) which adopts a tree structure in substructure graph mining. However, considering that the labels may not be the only attribute that a vertex can have, CTree is not able to index vertices with additional attributes other than vertex labels.

We adopt the CTree index structure to capture common structures in graphs and focus on subgraph queries, meanwhile, we extend it with additional integer attributes on the vertices.

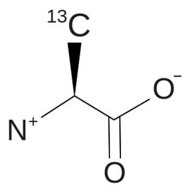
This chapter is organized as follows: Section 3.1 presents our extensions on the vertex attributes. Section 3.2 discusses the vertex merging methods after the extensions. Section 3.3 shows the building process of the ECTree. Section 3.4 gives a query format for the ECTree. Section 3.5 discusses the matching between a query graph and a data graph.

3.1 Extending the Vertices with Attributes

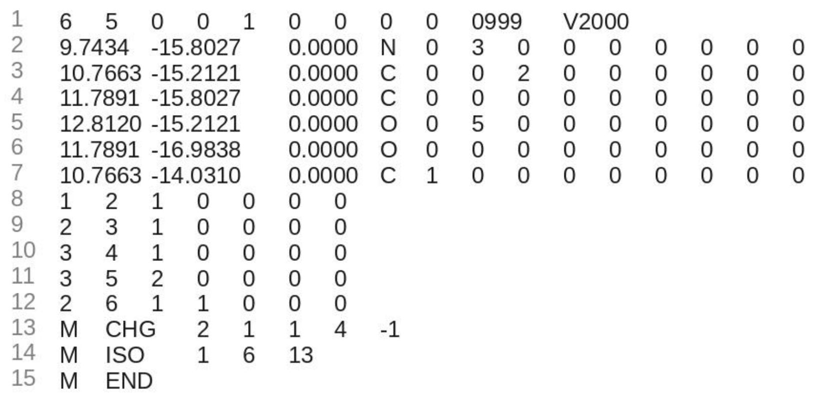
The existing CTree provides access to subgraph mining focusing on vertex labels and edge relations. However, actual graphs could contain more information on the vertices than only the labels. For example, chemical compounds, social networks, etc. have additional information of different types on each vertex. In this section, we firstly demonstrate an example of a chemical structure that has both label attributes and numerical attributes. Then we show our method to represent it as a graph in plain text format.

3.1.1 An Example in the SDF Format

SDF stands for structural-data file. It is one of a family of chemical-data file formats developed by MDL [DNH+92]. The SDF format is a text-based format for representing chemical compounds, in which the structural information and associated data items for one or more compounds are contained. An SDF format chemical consists of some header information, a connection table containing the atom info, the bond connections and types, followed by some sections of more complex information. The file format can be V2000, V3000 or a combination of both [Sym].



(a) ISIS/Draw version of Alanine



```
1   6  5  0  0  1  0  0  0  0  0999  V2000
2   9.7434  -15.8027   0.0000 N  0  3  0  0  0  0  0  0  0
3   10.7663 -15.2121   0.0000 C  0  0  2  0  0  0  0  0  0
4   11.7891 -15.8027   0.0000 C  0  0  0  0  0  0  0  0  0
5   12.8120 -15.2121   0.0000 O  0  5  0  0  0  0  0  0  0
6   11.7891 -16.9838   0.0000 O  0  0  0  0  0  0  0  0  0
7   10.7663 -14.0310   0.0000 C  1  0  0  0  0  0  0  0  0
8   1  2  1  0  0  0  0
9   2  3  1  0  0  0  0
10  3  4  1  0  0  0  0
11  3  5  2  0  0  0  0
12  2  6  1  1  0  0  0
13  M  CHG   2  1  1  4  -1
14  M  ISO   1  6  13
15  M  END
```

(b) Connection table of Alanine [Sym]

Figure 10: Alanine in SDF format

Figure 10(a) shows a chemical compound *Alanine*[NC05] drawn from a chemical structure drawing program called ISIS/Draw [LWSO04]. Figure 10(b) shows the connection table (CTab) of *Alanine*.

The first line is the counts line which specifies the number of atoms, bonds, and atom lists, the chiral flag setting, and the CTab version. Line 2-7 shows the atom block which specifies the atomic symbol and any mass difference, charge, stereochemistry and associated hydrogens for each atom. Line 8-12 shows the bond block which specifies the two atoms connected by the bond, the bond type and any bond stereochemistry and topology (chain or ring properties) for each bond. The last 3 lines are the property block that is provided for future expandability of CTab features, while maintaining compatibility with earlier CTab configurations. Detailed meaning of the values of the atom properties can be found in Appendix A.

We adopt the atom names as vertex labels, the property lists as vertex attributes and the bonds as edges in our ECTree index.

3.1.2 Data Representation in ECTree

We begin by transforming *Alanine* into the graph shown in Figure 11(a). Besides the vertex label, each vertex further has three more integer attributes adopted from 10(b). In order to store these additional attributes in text format, three positions after each vertex label are added, separated by a " | ", shown in Figure 11(b). The numbers of integer attributes on each vertex are the same and can be defined according to actual use of the dataset.

Here we give the definitions of data attribute vector, data vertex and data graph:

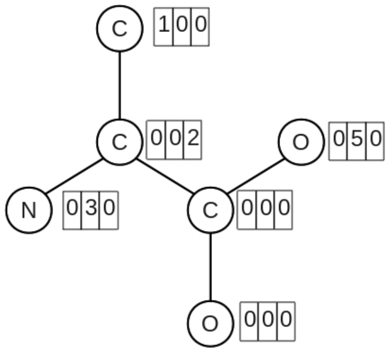
Definition 8 Data Attribute Vector, Data Vertex and Data Graph
A data attribute vector is a vector that contains only integers as its elements. A data vertex is a vertex that has a data attribute vector as its attribute vector. A data graph is a graph that contains only data vertices.

The data attribute vector Λ_v of a data vertex v is formally defined here:

$$\Lambda_v = \langle a_1, a_2, ..., a_n \rangle, a_i \in \mathbb{Z}, \forall i \in [1, n] \tag{1}$$

Each element a_i of Λ_v is an attribute value of the data vertex v. All the graphs from a dataset are always used as data graphs. Since a graph-closure is treated as a generalized graph, all graph-closures are data graphs.

There are some reasons that we define the data attribute vector to contain only integer numbers but not other data types such as string or boolean. Firstly, if the attribute contains strings, then the strings have to be matched exactly in the vertex



(a) The graph format of *Alanine*

```
#Alanine
6
N|0|3|0
C|0|0|2
C|0|0|0
O|0|5|0
O|0|0|0
C|1|0|0
5
0 1
1 2
2 3
2 4
1 5
```

(b) Text representation of (a)

Figure 11: Data structure of ECTree

mapping phase, which is not any different than having an extra label. Secondly, a pruning method can be developed with the numbers as attributes, this is discussed later in Section 4.2. Thirdly, in the validation chapter (Chapter 5), the datasets use only integers as attributes of the vertices. Besides, it is very simple to extend the attributes to real numbers if there is need.

3.2 Vertex Merging

The vertex matching phase in building a CTree is very simple: to check whether two vertices have the same label or not, if they do, then the two vertices match, otherwise they do not match. We define the label-matching for two data vertices in ECTree as follows: two data vertices label-match if these two data vertices have the same label.

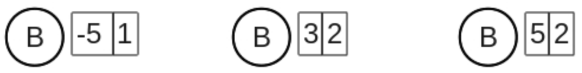
Now we consider the attribute vector of a vertex. In CTree, if two vertices are label-matched, there is no additional operations for merging them. In ECTree, we need to modify the vertex merging strategy so that the label-matched vertices or the non-label-matched vertices which are made into a vertex closure still capture necessary information for the later querying phase. CTree uses each graph-closure as a "bounding box" of constituent graphs which contains discriminative information of their descendants [ZYY07]. We want our index to inherit the feature of "bounding box" on the attributes as well, thus the merging methods of vertex/vertex-closures are taken into account.

There are three different cases to discuss in the vertex merging phase: *i*) merging two vertices, *ii*) merging a vertex and a vertex-closure, *iii*) merging two vertex-closures. We firstly ignore the integer attributes and map two graphs using *Neighbor Biased Mapping*, then we consider each of the merging cases separately in detail. In this section, we only discuss the merging methods, the graph mapping method *Neighbor Biased Mapping* is adopted from [HS06].

3.2.1 Merging Two Vertices

The main purpose of the closure-tree structure is to reuse common sub-structures. Therefore the merging of two or multiple vertices is a very frequent operation. Merging multiple vertices can be composed of a series of operations of merging two vertices, so we only discuss binary operations here.

Figure 12 shows two vertices v_a and v_b that have the same label. The merged vertex from two label-matched vertices is a vertex with the same label. v_a and v_b are merged into v_{merged} shown in Figure 12(c). For each attribute a_i in v_{merge}, we set it to the maximum absolute value at the same position of the merging vertices. $\Lambda_{v_{merged}}[1] = Max\{|\Lambda_{v_a}[1]|, |\Lambda_{v_b}[1]|\} = Max\{|-5|, |3|\} = 5$.



(a) v_a (b) v_b (c) v_{merged}

Figure 12: Merging two label-matched vertices

If a vector Λ_v needs to be merged from multiple vectors $\Lambda_{v_1}, ..., \Lambda_{v_M}, M > 1$, we denote it by $\Lambda_v = Merge(\Lambda_{v_1}, ..., \Lambda_{v_M})$. The $Merge$ operation is given below in

Equation 2: (Note that when the notation $|A|$ is used, if A is a set, then $|A|$ is the cardinality of A; if A is a number, then $|A|$ is the absolute value of A.)

$$Merge(\Lambda_{v_1}, ..., \Lambda_{v_M}) = \langle Max\{|\Lambda_{v_i}[1]| \mid i \in [1, M]\}, ..., Max\{|\Lambda_{v_i}[n]| \mid i \in [1, M]\}\rangle \quad (2)$$

Since the attribute vector of the merged vertex is the absolute ceiling of the all merging vertices, we denote it by a comparison operator "$\geqslant$": $\Lambda_{v_{merged}} \geqslant \Lambda_{v_i}$, $\forall i \in [1, M]$. The definition for the operator "$\geqslant$" between two data attribute vectors are given in Equation 3. (Note that we have $|\Lambda_{v_a}| = |\Lambda_{v_b}| = |\Lambda|$)

$$\Lambda_{v_a} \geqslant \Lambda_{v_b} \Leftrightarrow |\Lambda_{v_a}[i]| \geqslant |\Lambda_{v_b}[i]|, \forall i \in [1,\ |\Lambda|] \quad (3)$$

According to Equation 2 and 3, it can be inferred that:

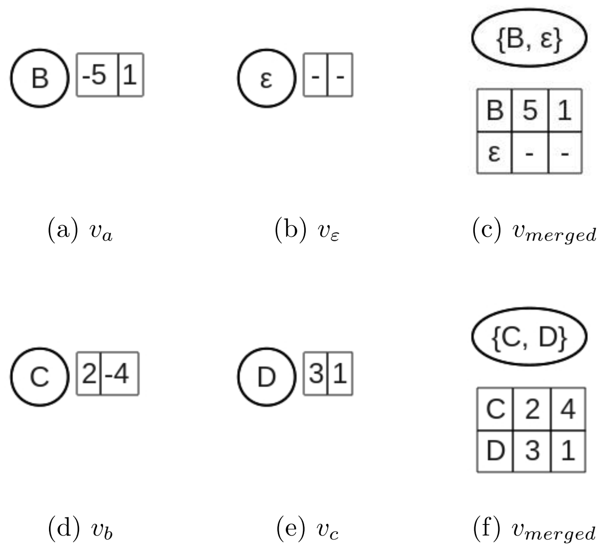
$$Merge(\Lambda_{v_1}, ..., \Lambda_{v_M}) \geqslant \Lambda_{v_i},\ \forall i \in [1,\ M] \quad (4)$$

Next we take a look at the merging of two non-label-matched vertices. The merged vertex from two non-label-matched vertices is a vertex-closure which has a list of attribute vectors of each of the vertex included in the merging. The structure of a vertex-closure in ECTree index has two parts: a label set and an attribute vector list. We further discuss two subcases in vertex merging.

The first subcase is that a vertex v is mapped to a dummy vertex v_ε. Figure 13(a)(b)(c) shows that a vertex v_a is mapped to a dummy vertex v_ε and they are both added to a vertex-closure v_{merged}. A dummy vertex in ECTree has the same label as in CTree: 'ε', and has an empty attribute vector $\Lambda_\varnothing$. When a vertex v is added to a vertex-closure v_{merged}, the label of v is added to the label set of v_{merged}, each attribute in the vertex attribute vector of v is transformed into the corresponding absolute value, and then the attribute vector is added to the attribute vector list of v_{merged}. If the vertex to be added is a dummy vertex, the label 'ε' is added to the label set of v_{merged} and $\Lambda_\varnothing$ is added to the attribute vector list of v_{merged}.

The second subcase is that a vertex v_1 is mapped to another vertex v_2 which has a different label. Both labels of v_1 and v_2 are added to the label set of v_{merged}, and all attributes of vectors of v_1 and v_2 are transformed into the corresponding absolute value and added to the attribute vector list of v_{merged}. Figure 13(d)(e)(f) shows an example of this case.

Assume the number of distinctive labels in a graph database is D, the upper bound for the size of a vertex-closure in the ECTree index is $D + 1$ because of a possible dummy vertex.



(a) v_a (b) v_ε (c) v_{merged}

(d) v_b (e) v_c (f) v_{merged}

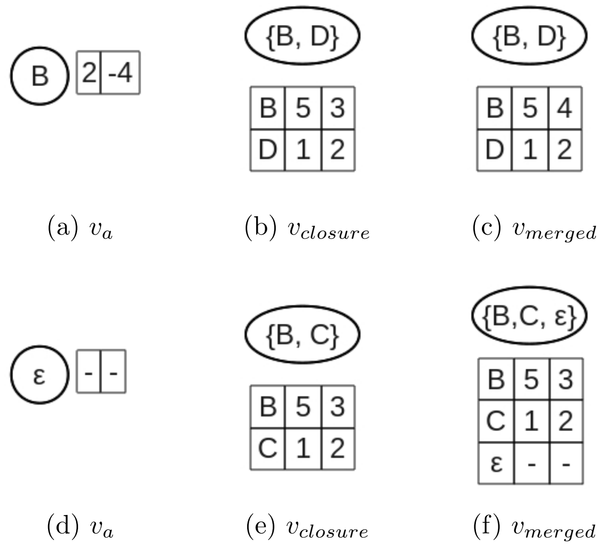
Figure 13: Merging two non-label-matched vertex

3.2.2 Merging A Vertex and A Vertex-Closure

In this case, we merge a vertex v and a vertex-closure v_c into a new vertex-closure. There are two possible subcases: 1) the vertex-closure $v_{closure}$ contains a vertex label that is the same as the label of v, 2) the vertex-closure $v_{closure}$ does not contain any vertex label that is the same as the label of v.

In subcase 1), the merging vertex v has a same label l (l =‘B’ in Figure 14(a)) of one of the labels in the vertex-closure $v_{closure}$, we update the vertex attribute vector Λ_l in the vertex-closure using the attribute vector Λ_v according to Equation 2: $\Lambda_{l-new} = Merge(\Lambda_l, \Lambda_v)$. If the label of the merging vertex v is ε, then the merged vertex-closure v_{merged} is the same as the merging vertex-closure $v_{closure}$, no operations are needed. An example of the merging is shown in Figure 14(a)(b)(c).

In subcase 2), the process is similar to merging two unmatched vertices but to replace one vertex by a vertex-closure in this case. An example is shown in Figure



(a) v_a (b) $v_{closure}$ (c) v_{merged}

(d) v_a (e) $v_{closure}$ (f) v_{merged}

Figure 14: Merging a vertex and a vertex-closure

14(d)(e)(f).

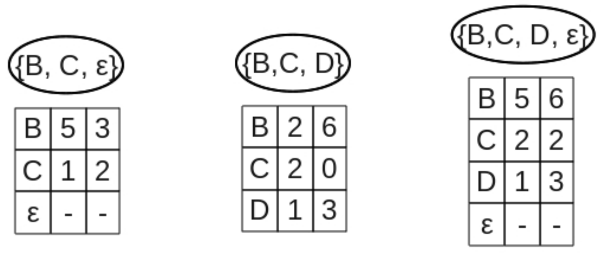
3.2.3 Merging Two Vertex-Closures

In this case, two vertex-closures $v_{closure1}$ and $v_{closure2}$ are merged into a new vertex-closure v_{merged}. We still have two possible subcases: 1) the two vertex-closure label sets do not have any vertex labels in common, 2) the two vertex-closure label sets have some vertex labels in common. The first subcase is similar to merging two unmatched vertices, we simply add the labels and corresponding attribute vectors into a new vertex-closure, no further operations are needed. We mainly discuss the two steps in merging two vertex-closures that have same labels. Assume L_1 is the label set of $v_{closure1}$, L_2 is the label set of $v_{closure2}$.

Step 1:

- if the common label $l \neq$ 'ε': add l to the label set of v_{merge}, add $\Lambda_{l-new} = Merge(\Lambda_{l_1}, \Lambda_{l_2})$ to the attribute vector list, $\forall l_1 = l_2 \in L_1 \cap L_2$, $l_1 \in L_1$, $l_2 \in L_2$.
- if the common label $l =$ 'ε': add ε to the label set of v_{merge}, add $\Lambda_{\varnothing}$ to the attribute vector list.

Step 2, $\forall l \in (L_1 \cup L_2 - L_1 \cap L_2)$, add l to the label set of v_{merge} and add Λ_l to the attribute vector list.



(a) $v_{closure1}$ (b) $v_{closure2}$ (c) v_{merged}

Figure 15: Merging two vertex-closures

An example of this case is shown in 15, $v_{closure1}$ and $v_{closure2}$ have two labels in common: l_1 = 'B' and l_2 = 'C', while 'ε' and 'D' are not common labels. Thus we use Equation 2 to calculate the new attribute vector in v_{merge}: $\Lambda_{merged-l_1} = Merge(\Lambda_{closure1-l_1}, \Lambda_{closure2-l_1})$, $\Lambda_{merged-l_2} = Merge(\Lambda_{closure1-l_2}, \Lambda_{closure2-l_2})$. '$\varepsilon$' and 'D' are directly added to the list of attribute vectors without any operations.

3.3 Building the ECTree

We modified the data representation of a graph in the dataset, thus the structure of a graph-closure and its computing method have to be modified accordingly. Since the structure of edges and edge closures are not modified, the corresponding methods do not need modifications. We focus on our new methods of computing graph closures in this section.

3.3.1 Data Graph Mapping

In CTree, there is no distinguishment between mapping two data graphs and mapping a query graph and a datagraph. We will introduce the mapping between a query graph and a data graph for ECTree in Section 3.4.

In order to make two graphs G_1 and G_2 into a graph-closure C, it is necessary to find a possible graph mapping ϕ initially from G_1 to G_2. In ECTree, we get the graph mapping ϕ by using *Neighbor Biased Mapping*, ignoring the attributes of the vertices; then each mapped pair of vertices/vertex-closures are merged using previous described methods.

We do not do exact matching for data graph mapping for the following reasons. Firstly, the possibility for two vertices in two graphs to have the exact same label and attributes is very low; secondly, we still maintain the clustering feature of the tree index using the vertex merging methods.

3.3.2 An ECTree Building Example

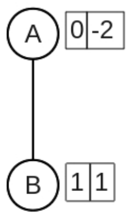
The process of building graph-closures is the process of building an ECTree. We show an example of building a graph-closure from three graphs in a dataset in Figure 16.

Firstly Closure 1 is computed from Graph 1 and Graph 2. For vertex C in Graph 2, there is no corresponding vertex so it is mapped to a dummy vertex ε, both of them are put into a vertex-closure $\{C, \varepsilon\}$.

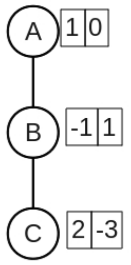
Next we compute Closure 2a from Closure 1 and Graph 3 using the same method, as shown in Figure 16(e). Figure 16(f) shows Closure 2b which is computed from Closure 1 and Graph 3 as well but using a different graph mapping. Graph mappings can be obtained using different graph mapping algorithms, but for a chosen algorithm, the mapping obtained between two graphs is unique. Ideally, the mapping method should maximize the common structure and attributes of the two graphs. Since we did not find a proper mapping algorithm for attributed graphs, the NBM [HS06] method is used.

3.4 A Query Graph Format for ECTree

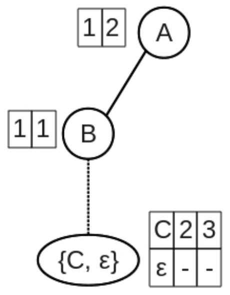
Just like CTree, we can also use a data graph as a query graph. For instance, to query the substructure as shown in Figure 11(a), the representation in Figure 11(b) can simply be used as a query. However, in ECTree, we focus on the graph data mining of the attributes of the vertices, the use of the format of the data graphs as query graphs is too simple and does not make full use of the new index. We introduce



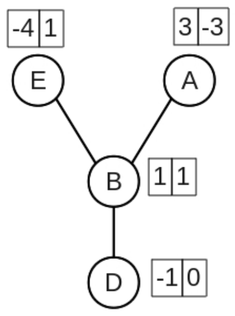
(a) Graph 1



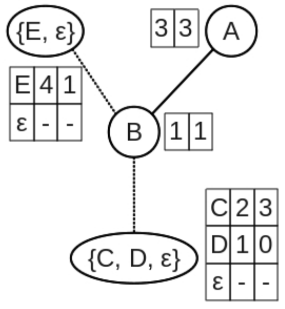
(b) Graph 2



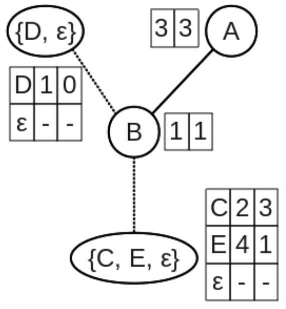
(c) Closure 1



(d) Graph 3



(e) Closure 2a



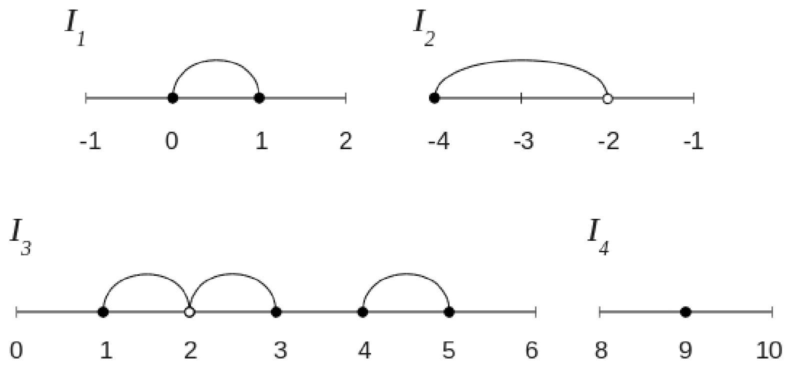
(f) Closure 2b

Figure 16: Computing a graph-closure in ECTree

a query format for ECTree that allows more flexible queries in terms of the attributes of vertices.

Firstly, we allow the use of intervals to appear in the query graphs. An integer interval is made up of two integers and two brackets. An open or close bracket means the integer at that side is not included in the interval, while a square bracket means the integer at that side is included in the interval.

Two or more intervals can be used together to make up more complex intervals. Some examples of interval sets are shown in Figure 17(a): $I_1 = [0,1]$, $I_2 = [-4,-2)$, $I_3 = [1,2) \cup (2,3] \cup [4,5]$, $I_4 = 9$. There is only one relational operator "$\cup$" union supported between the intervals. Since there is only one relational operator, we omit it in the representation and by default, it is implied that the union operation is used between all the intervals using together.



(a) Four intervals

```
#Q
6
A|0|0
B|[-2,2](5,7]|0
C|0|0
D|0|0
E|0|[0,0][1,4)(4,7)
F|[-2,-2]|3
5
0 1
1 2
1 3
3 4
3 5
```

(b) A graph with intervals

Figure 17: Examples of interval sets

Therefore, [-2,2]∪(5,7] is written as [-2,2](5,7] in plain text format. I_3 is written as $I_3 = [1,2)(2,3][4,5]$. I_4 can both be written as 9 or $I_4 = [9,9]$. But if a single integer value i is unioned with other intervals, it has to be written in the form of $[i,i]$. E.g, 5∪[6.8] should be written as [5,5][6,8]. Figure 17(b) shows a query graph Q which uses the intervals in the graph attribute vectors.

In addition to the integer intervals described above, we use the symbol '*' to replace the interval $(-\infty,+\infty)$. The symbol '*' cannot be unioned with any other interval(s). E.g., the plain text format of the query vertex 'D$|(-\infty,+\infty)|$[-2,-2]' is written as 'D$|*|$[-2,-2]'.

In contrast to **Definition 8**, we give the definitions for Query Attribute Vector, Query Vertex and Query Graph here:

Definition 9 Query Attribute Vector, Query Vertex and Query Graph

A query attribute vector is a vector that contains integer interval sets as its elements. A query vertex is a vertex that has a query attribute vector as its attribute vector. A query graph is a graph that contains query vertices.

Let $I_i, i \in [1, n]$ be a set of integer intervals, we denote the query attribute vector Λ_v^* as follows:

$$\Lambda_v^* = \langle I_1, I_2, ..., I_n \rangle \tag{5}$$

3.5 Query Matching

In this section, we discuss the query matching for ECTree graph index. We have demonstrated different formats of the query graphs and the data graphs as well as the methods to make the graph-closures in ECTree. There are two different cases in the query matching: matching a query with a non-closure graph and a graph-closure. In both cases, we assume the mapping ϕ between a query graph and a data graph is obtained by using the graph mapping algorithm *Neighbor Biased Mapping* from [HS06].

3.5.1 Matching a Query Vertex and a Data Vertex

There are two cases to discuss: 1) matching a query vertex and a data vertex in a non-closure graph, 2) matching a query vertex and a data vertex in a graph-closure.

In the first case, a query vertex v_Q matches a data vertex v_D in a non-closure graph if $\Lambda_{v_D}[i] \in \Lambda_{v_Q}^*[i], \forall i \in [1, |\Lambda_{v_Q}^*|]$. This is easy to understand because a non-closure data vertex is in the vertex set of a non-closure graph which is in the original dataset. An example is shown in Figure 18. v_Q matches v_{D1} but does not match v_{D2}.

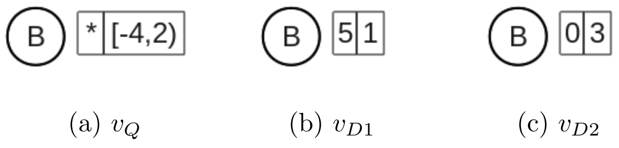


Figure 18: Matching a query vertex and a non-closure data vertex

For simplification, we define the operator "⊒" between a query attribute vector and a data attribute vector of a non-closure vertex in Equation 6:

$$\Lambda^* \supseteq \Lambda \Leftrightarrow \Lambda[i] \in \Lambda^*[i]\,, \forall i \in [1, |\Lambda^*|] \tag{6}$$

In the second case, since the attribute vector of a vertex-closure is the absolute ceiling of all the attribute vectors in the closure, we only compare the values in the data attribute vector to the minimum absolute values of the intervals in the query vector. We denote the minimum absolute value within a interval set I by *Min-Absolute*$(I) = Min\{|\ i\ |\ \mid\ i \in I\}$. E.g., in Figure 19(a) *Min-Absolute*$(I) = i_2$, in Figure 19(b) *Min-Absolute*$(I) = 0$. Here the definition of the operator "$\supseteq$" between a query attribute vector and a data attribute vector of a graph-closure is given in Equation 7.

$$\Lambda^* \supseteq \Lambda \Leftrightarrow |\Lambda[i]| \geqslant \textit{Min-Absolute}(\Lambda^*[i])\,, \forall i \in [1, |\Lambda^*|] \tag{7}$$

We take a look at the two examples in Figure 19. Assume $I = [i_1, i_2)[i_3, +\infty)$ is an interval set in a query attribute vector $\Lambda^* = \langle I \rangle$, $i_1 < i_2 < i_3$, $|i_2| < |i_3|$. $d_1 > 0, d_2 > 0, d_2 > d_1$ are two values in two different data attribute vector $\Lambda_1 = \langle d_1 \rangle$ and $\Lambda_2 = \langle d_2 \rangle$. In Figure 19(a), the query interval requires the value of the data attribute falls in the interval set I, but for Λ_1, the value can only fall within $[-d_1, d_1]$ which means all the values of the attribute of the vertices in the vertex-closure are within $[-d_1, d_1]$ thus cannot match I. In this case, only Λ_2 satisfies the condition. In Figure 19(b), the value '0' is included in the interval set I, therefore the minimum absolute value within I is '0'. Since $d_1 > 0$ and $d_2 > 0$, they both satisfy the condition.

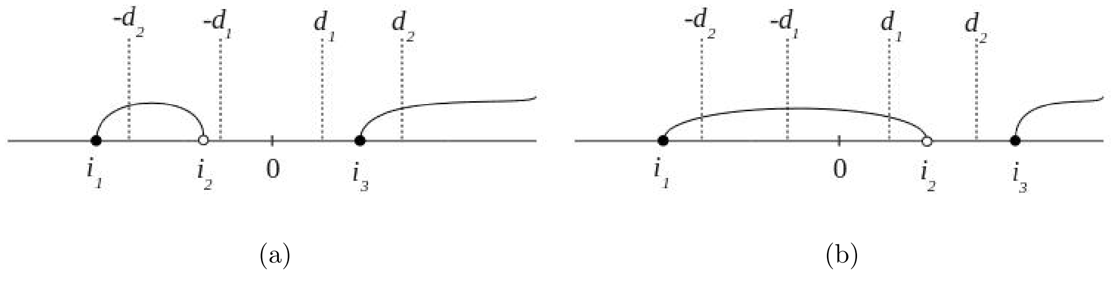


Figure 19: Two intervals showing the concept of Minimum-Absolute values

We sum the matching between a query attribute vector $\Lambda^*_{v_Q}$ and a data attribute vector Λ_{v_D} as follows:

$$\Lambda^* \supseteq \Lambda \Leftrightarrow \begin{cases} \Lambda[i] \in \Lambda^*[i]\,, \forall i \in [1, |\Lambda^*|] & (v_Q \textit{ is non-closure}) \\ |\Lambda[i]| \geqslant \textit{Min-Absolute}(\Lambda^*[i])\,, \forall i \in [1, |\Lambda^*|] & (v_Q \textit{ is a closure}) \end{cases}$$

The modified vertex mapping algorithm is shown in Algorithm 3.

Algorithm 3: VertexMapping_Modified(v_1, v_2)

```
input  : A query vertex $v_1$, a data vertex $v_2$
output: A boolean value, true if mappable, false if not
if label of $v_1$ does not match label of $v_2$ then
    return false;
$\Lambda^* \leftarrow$ attribute vector of $v_1$ ;
$\Lambda \leftarrow$ attribute vector of $v_2$ ;
if $v_2$ is from a graph-closure then
    for $i \leftarrow 1$ to $|\Lambda^*|$ do
        $k \leftarrow$ Min-Absolute$(|\Lambda^*[i]|)$ ;
        if $\Lambda[i] < k$ then
            return false ;
else
    for $i \leftarrow 1$ to $|\Lambda^*|$ do
        boolean $b \leftarrow$false;
        interval set $I \leftarrow \Lambda^*[i]$;
        for $j \leftarrow 1$ to $|I|$ do
            if $\Lambda[i] \in I[j]$ then
                $b \leftarrow$ true;
                break;
        if $b$ is false then
            return false;
return true;
```

3.5.2 Matching a Query Graph and a Data Graph

Based on our discussion in the previous subsection, we define the subgraph matching as follows:

A query graph $G_Q(V_Q, E_Q)$ matches a data graph $G_D(V_D, E_D)$ if G_Q is subgraph-isomorphic to G_D under graph mapping ϕ and $\forall v \in V_Q, \Lambda^*_v \supseteq \Lambda_{\phi(v)}$.

When a query graph is compared to a data graph, the histogram pruning is used firstly, followed by the Pseudo Subgraph Isomorphism test, and finally the subgraph isomorphism test. Assume G_Q is a query graph and G_D is a data graph, Table 2 shows the descriptions of the symbols to be used.

Symbol	Description
n_1	the number of vertices in G_Q
n_2	the number of vertices in G_D
d_1	the maximum vertex degree in G_Q
d_2	the maximum vertex degree in G_D
l	the pseudo compatibility level defined for the PSI pruning [HS06]
$M()$	the time complexity of maximum cardinality matching for bipartite graphs
a	the number of vertex attributes

Table 2: Notations

The worst case complexity for the histogram pruning in ECTree is the same as CTree: $O(n_1^2)$, since all calculations are one-time-cost and pre-processed. The PSI pruning algorithm for ECTree has a worst case complexity of $O(ln_1n_2(d_1d_2 + M(d_1, d_2)) + aM(n_1, n_2))$ [HS06]. The pseudo compatibility level is defined before querying and can be adjusted. Hopcroft and Karp's algorithm [HK71] finds a maximum cardinality matching in $O(n^{2.5})$ time.

Here we do some further discussions about **Exact Matching** and **Inexact Matching**. When we match a query vertex v_Q and a data vertex v_{D1} from a non-closure graph, we use exact matching. That is, the attribute values of Λ_{D1} need to be included in the corresponding intervals of Λ^*_Q. When we match a query vertex v_Q and a data vertex v_{D2} from a graph-closure, we use inexact matching which means the attribute values of Λ_{D2} do not have to exactly match the corresponding intervals of Λ^*_Q, but just be larger than the corresponding absolute minimum values. There are a few reasons why we do not use exact matching:

- On memory consumption, storing the exact numerical value in every one of the graph-closures at each level will cost a huge amount of space and thus make the index very large.

- On time consumption, if we store a list for each position in an attribute vector, then matching a query attribute vector and a data attribute vector will be very trivial and the time cost can be unacceptable. At a relatively higher level, the lists in the attribute vectors can be very long and hard to update.

- The dead space in graph merging needs to be considered as well. Even if a query graph and a graph-closure are matched using exact matching, there is no guarantee that there are answer graphs under the branch of this graph-closure because the matched graph could be in the dead space.

3.6 Conclusion

In this chapter, we have provided description of our extended closure-tree that manages graph data with integer attributes on the vertices, based on the original CTree index. ECTree is a graph index that specifies the queries on the vertex attributes and at the same time provides very flexible formats of queries in terms of the attributes. We discussed the new merging and matching methods in ECTree for the new structure of both data and query graphs which is very different from the CTree index.

The ECTree index inherits the feature of CTree: graph clustering hierarchically. An ECTree node is the clustering of both the structure and the attribute of its descendents.

Chapter 4

Optimizations for ECTree

According to the features of ECTree, we are able to query a graph dataset with more flexible query graphs and extended attributes. However, we find it yet not flexible enough to query the ECTree index on the vertex labels. There are also methods that we can improve to make the ECTree more efficient, such as the pruning algorithm. This chapter presents our approaches to make the ECTree more useful.

The rest of the chapter is organized as follows: Section 4.1 introduces a new query format for ECTree. Section 4.2 presents our novel pruning method which is named Degree-Attribute pruning.

4.1 A More Flexible Query Format

In CTree, alphabetic letters are used on the vertex labels. As we can see the two parts from Algorithm 3: 1) the mapping of the vertex label, 2) the mapping of the attributes of the vertex. At line 1, the mapping of the vertices is simply checking whether one label matches the other label. Since both labels contain only alphabetic letters, the check is only string matching.

However, in many cases, fixed labels can lead to time cost redundancy. For instance, a group of query graphs may share some common information on the structure and labels, and are different in just a few labels. In this case, the fixed label index will have to run each of the queries separately, but a more flexible query format may need to combine them into one query and run it only once and save time from the expensive subgraph-isomorphism test. There are also cases that a query graph contains some vertices with unknown labels.

We propose our approach to use Regular Expressions in ECTree, with some query examples using regular expressions to demonstrate the usefulness. Then we show our modifications on the mapping algorithms.

4.1.1 Using Regular Expressions on Query Labels

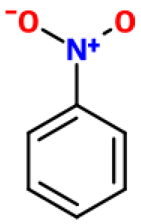
In Figure 20(a) and (c) (drawing using *eMolecules* [eI]), substructures 1 and 2 are two real chemical substructures extracted from NCI subset [NCI]. Since they are substructures, other possible bonds are not shown and the hydrogen atoms are omitted. Figure 20(b) and (d) shows two queries Q_1 and Q_2 which use substructures 1 and 2. By observing Q_1 and Q_2, it is obvious that they have similar structures. If we can find a way to merge those two queries into one query such that the results from both queries can be returned as the results of the merged query, then the querying time may be reduced.

Considering this, if the vertex labels are flexible then multiple queries could be combined into one query if they have similar structures. Benefits also exists in that a query graph with undetermined labels can be queried. For example, different metal atoms. Here we give the definitions for Data Label and Query Label:

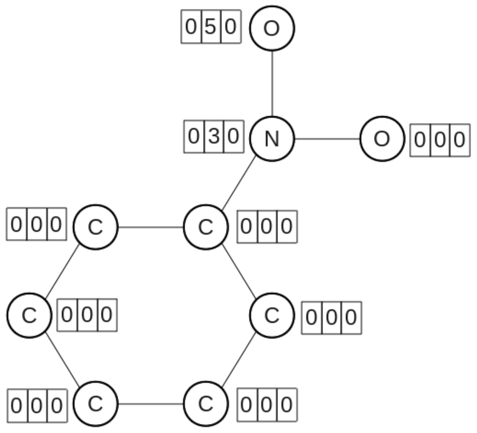
Definition 8 Data Label and Query Label

A data label is a string composed of alphabetic letters only. A query label is either a data label or a regular expression.

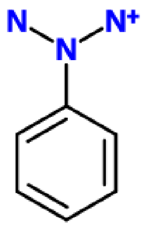
In the definition of query label, we use the regular expressions because regular expressions are very powerful in expressing different types of combination of strings. Having the regular expressions, queries Q_1 and Q_2 can be replaced by a single query Q_3 (Figure 21). In some cases, the graphs returned may contain unwanted results because there are graphs contained in Q_3 but are neither Q_1 nor Q_2. When the regular expression is used as a query label, we should be careful because if too many unwanted results are returned, the idea of combining queries becomes inefficient. Moreover, there is another limitation for using regular expressions to combine queries: the candidate queries must have the same structures (structurally isomorphic) so as



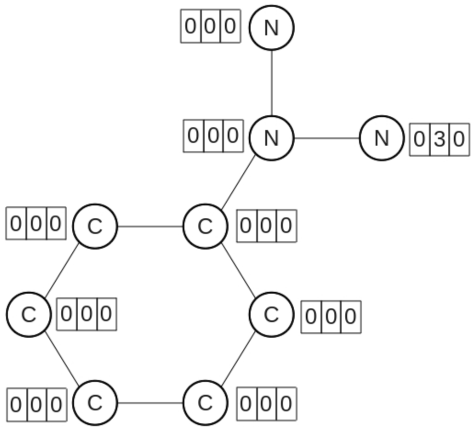
(a) Substructure 1



(b) Q_1



(c) Substructure 2



(d) Q_2

Figure 20: Two chemical structures as queries

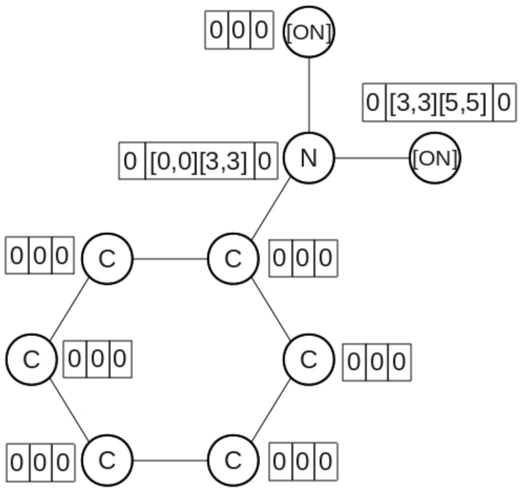


Figure 21: The merged query Q_3

for the vertices to be combined. If two query graphs are not structurally isomorphic, then they cannot be combined.

Moreover, to enhance the usefulness of the system, the wildcard * for vertex labels is supported. Vertices with the wildcard $*$ may be mapped to vertices with arbitrary labels. Some approaches such as CT-Index [KKM11] discard all vertices and edges with wildcard labels from the query graph for filtering (CT-Index supports edge labels). This method leads to possible loss of information of query graph structure and may increase the false-positive rate, therefore in our approach we keep the vertices with wildcard labels and their related edges. The matching condition for wildcard query vertices is directly verified with the subgraph isomorphism algorithm.

4.1.2 Label Groups

In Figure 21, we replace queries Q_1 and Q_2 by a single query Q_3. We find a problem that both Q_1 and Q_2 have vertices that have the same labels. For example, both are 'O' or both are 'N'. However, the query results returned from Q_3 may not have the same label: one of the vertices can be 'O' and the other one can be 'N'. Thus we introduce a way to guarantee two or more labels to be the same in the query: label

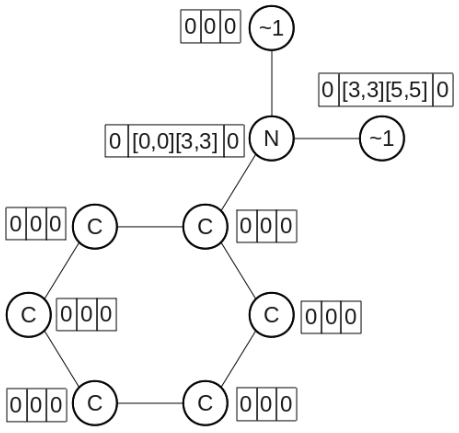


Figure 22: A query Q_4 showing the label groups

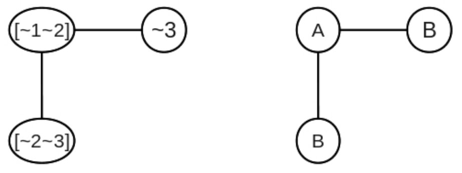
groups.

We use the notation '~' followed by a non-negative integer number to stand for a group of the labels that are logically bound to a string value. The label groups can be simply understood as 'variables'. Note that two different label groups cannot be bound to the same string. That means, for example, if '~1' is bound to 'A', then '~ 2' cannot be bound to 'A' anymore and has to be bound to another string.

Though the purpose of using the label groups is to provide queries so that vertices which have the same label but these labels are not determined can be queried, the label groups can also be used to distinguish labels. If a label group appears only once in a query graph, that means this label group is used to distinguish itself from any other label groups.

We use the query Q_4 in Figure 22 to replace of the queries Q_1 and Q_2 so it is guaranteed that both vertices have the same label. So that the results returned from Q_4 will include the union of the results returned by Q_1 and Q_2.

However, there is still a limitation to the label groups: we bind each of the label groups to a different label in the graph mapping process, that means the number of different labels in the data graph must be no smaller than the number of label groups in the query graph. We show an example in Figure 23 (the attributes are ignored). G_Q has three different label groups, G_D has only two distinguished labels but obviously G_Q is still mappable to G_D.



(a) G_Q (b) G_D

Figure 23: A matching example for label groups

In our later discussions and experiments, we always assume that the number of distinguished labels in a data graph is no smaller than the number of different label groups in a query graph. In addition, since the label groups are treated as variables, they can be used in regular expressions. For example, '[~1N]' either matches the string that is bound to '~1', or 'N'.

4.1.3 Implementation

The GNU C Library [GNU] offers a Regular Expressions interface declared in the header file *regex.h*. This POSIX standard Regular Expressions tool *pseudo compiles* the regular expression and produces a special data structure which enables fast execution of the pattern using the function $regcomp(RegExp)$ which returns the pseudo compiled pattern *pattern*. Then the boolean function $regexec(pattern, S)$ can be called to compare *pattern* and a string S.

Since the query format has been modified, several related algorithms need to be modified as well.

Algorithm 4: GraphMapping(G_Q, G_D)

```
input  : A query graph $G_Q$ and a data graph $G_D$
output: A boolean value, *true* if $G_Q$ and $G_D$ are mappable, *false* if not
$V'_D \leftarrow CountVertex(V_D)$;
$n \leftarrow$ number of label groups in $G_Q$;
if $n \neq 0$ then
    if $n > |V'_D|$ then
        return false;
    // If the number of label groups is bigger than the number
       of different labels in $V_D$, then $G_Q$ and $G_D$ are not
       mappable.
    for $i \leftarrow 1$ to $n$ do
        $B[i] \leftarrow i$ ;
    // $B[x] = y$ means that label group $x$ is bound to label
       $V'_D[y].label$
    while true do
        Graph $G'_Q(V'_Q, E'_Q) \leftarrow G_Q(V_Q, E_Q)$;
        foreach $v \in V'_Q$ do
            if $\forall j \in [1, n], \exists$ label group $j$ in $v.label$ then
                Replace label group $j$ by $V'_D[B[j]].label$;
        if $subgraphIsomorphism(G'_Q, G_D)$ is true then
            return true;
        $B \leftarrow FindNewBinding(B, |V'_D|)$ ;
        if $B$ is null then
            break while ;
    return false;
else
    return $subgraphIsomorphism(G_Q, G_D)$;
```

Algorithm 4 shows the modified graph mapping algorithm. In Algorithm 4, functions *CountVertex*(V) and *FindNewBinding*$(B,\ c)$ can be found in Appendix B.1. Function *CountVertex*(V) returns a vertex vector which has different vertex labels from V. Function *FindNewBinding*$(B,$ c) returns a new binding solution according to the current binding B and the total number of different labels to bind.

According to the new format, the query label is not only alphabetic letters, therefore the vertex mapping algorithm Algorithm 3 needs to be modified as well. As is shown in Algorithm 5, whether or not the query label maps to the data label is tested first, if false is returned then the attribute vector testing is bypassed. If the label mapping test succeeds, then the attributes will be tested further.

4.2 A Novel Pruning Method for ECTree

After we change the query label format, the previous histogram-based label pruning method from CTree can not be adopted anymore. We discuss the pruning method in CTree and the reason why it can not be used for the ECTree index. Next we propose a method to generalize the feature of a graph which we name 'Degree-Attribute Feature'. Then a novel pruning method for the ECTree based on the 'Degree-Attribute Feature' is proposed.

This pruning method is based on that all the graphs, both in the dataset and the queryset, have only fixed alphabetic letters as the vertex labels, so that the number of appearance of different vertex labels can be calculated as well as for the edges. Our new query format allows the label of a vertex to be flexible in the query graphs with wildcard labels allowed and thus it is not clear how to define a histogram for the query graphs in this case. If the histogram-based feature cannot be calculated, the pruning cannot work for our ECTree index.

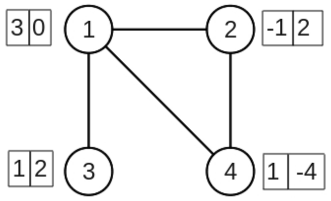
4.2.1 The Degree-Attribute Feature Vector

Though a query vertex no longer has a fixed label, it still has other fixed features: its attribute vector and its degree. We start by introducing the calculation method of the **Degree-Attribute(DA)** feature vector for a data graph. We ignore the labels of a data graph and the numbers on the vertices are only marks of the vertices. Two

Algorithm 5: VertexMapping(v_Q, v_D)

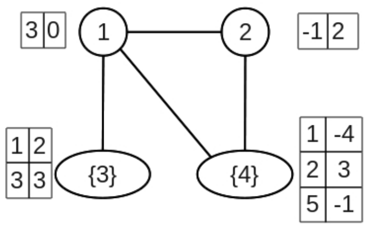
```
input  : A query vertex $v_Q$, a data vertex $v_D$
output: A boolean value, true if mappable, false if not

$l_D \leftarrow v_D.label$;
$l_Q \leftarrow v_Q.label$;
if $l_Q$ is a regular expression then
    $pattern \leftarrow regcomp(l_Q)$;
    $bool \leftarrow regexec(pattern, l_D)$;
    if bool is false then
        return false;
else
    if $l_Q$ does not match $l_D$ then
        return false;
$\Lambda^* \leftarrow$ attribute vector of $v_Q$, $\Lambda \leftarrow$ attribute vector of $v_D$ ;
if $v_D$ is a vertex-closure then
    for $i \leftarrow 1$ to $|\Lambda^*|$ do
        foreach element $I$ in $\Lambda^*[i]$ do
            $k \leftarrow Min\text{-}Absolute(I)$ ;
            if $\Lambda[i] < k$ then
                return false ;
else
    for $i \leftarrow 1$ to $|\Lambda^*|$ do
        boolean $b \leftarrow$ false, interval set $I \leftarrow \Lambda^*[i]$;
        for $j \leftarrow 1$ to $|I|$ do
            if $\Lambda[i] \in I[j]$ then
                $b \leftarrow true$;
                break;
        if not $b$ then
            return false;
return true;
```

data graphs $G_{D1}(V_{D1}, E_{D1})$ and $G_{D2}(V_{D2}, E_{D2})$ are shown in Figure 24(a) and (c). The numbers in braces mean that the vertex is a vertex-closure.



d	$F_d(G_D)$
1	<1,2>
2	<2,6>
3	<3,0>

(a) G_{D1} (b) $F_d(G_{D1})$



d	$F_d(G_D)$
1	<4,5>
2	<9,10>
3	<3,0>

(c) G_{D2} (d) $F_d(G_{D2})$

Figure 24: DA feature vectors for a non-closure data graph and a graph closure

We calculate the DA feature vector F_d of a data graph as shown in Equation 8 and 9 ($n = |\Lambda_{v_a}| = |\Lambda_{v_b}|$). If the data graph is a graph-closure, then all the vertices within a vertex-closure have the same degree. $MaxDegree(V)$ represents the maximum degree in the vertex set V. Figure 24(b) and (d) show the DA feature vectors of a non-closure data graph G_{D1} and a graph-closure G_{D2}.

$$\Lambda_{v_a} + \Lambda_{v_b} = \langle |\Lambda_{v_a}[1]| + |\Lambda_{v_b}[1]|, ..., |\Lambda_{v_a}[n]| + |\Lambda_{v_b}[n]| \rangle \tag{8}$$

$$F_i(G_D) = \sum_{degree(v)=i} \Lambda_v, \ i \in [1, MaxDegree(V_D)], \ v \in V_D \tag{9}$$

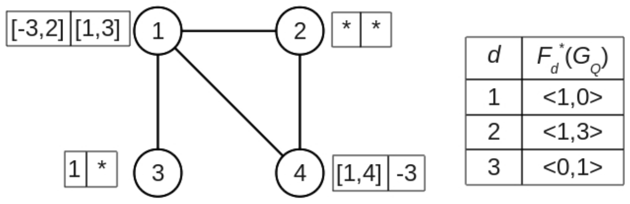
Next we discuss the method to calculate the DA feature vector for a query graph. Since there are integer intervals in a query attribute so the equation to add two query attribute vectors is slightly different than Equation 9. We show the calculation of the DA feature vector F_d^* for a query graph in Equation 10, 11 and 12 :

$$\Lambda^*_{v_a} + \Lambda^*_{v_b} = \langle \Lambda^*_{v_a}[1] + \Lambda^*_{v_b}[1], ..., \Lambda^*_{v_a}[n] + \Lambda^*_{v_b}[n] \rangle \tag{10}$$

$$\Lambda^*_{v_a}[j] + \Lambda^*_{v_b}[j] = \mathit{Min\text{-}Absolute}(\Lambda^*_{v_a}[j]) + \mathit{Min\text{-}Absolute}(\Lambda^*_{v_b}[j]),\ j \in [1, |\Lambda^*|] \tag{11}$$

$$F^*_i(G_Q) = \sum_{degree(v)=i} \Lambda^*_v,\ i \in [1, MaxDegree(V_Q)],\ v \in V_Q \tag{12}$$

We also show an example to calculate the DA feature vector for a query graph G_Q in Figure 25:



d	$F^*_d(G_Q)$
1	<1,0>
2	<1,3>
3	<0,1>

(a) G_Q (b) $F^*_d(G_Q)$

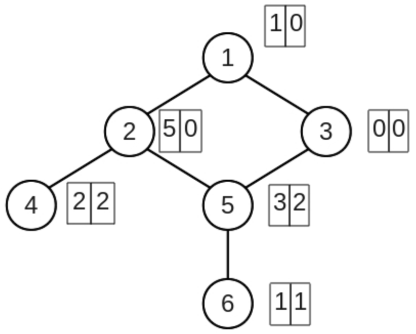
Figure 25: DA feature vector for a query graph

4.2.2 The Pruning Strategy

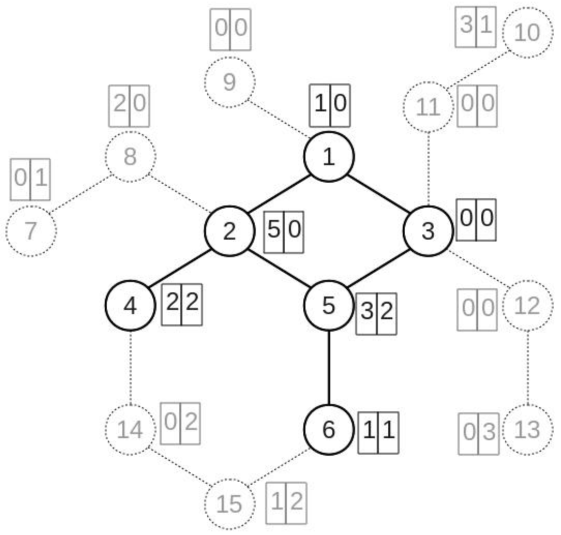
Having the DA feature vectors to represent the features of the degrees and attribute vectors, we introduce the pruning method which we name DA-Pruning. We start by looking at a an example. Figure 26 shows a query graph G_{Q1} and a data graph G_{D1}. For simplification, each vertex attribute has a length of two.

We can observe from Figure 26, if query graph $G_Q(V_Q, E_Q)$ is a subgraph of data graph $G_D(V_D, E_D)$ under graph mapping ϕ, then the condition $\forall v \in V_Q, Degree(v) \leqslant Degree(\phi(v))$ must be satisfied. The degree of a vertex $Degree(v)$ in the query graph must be no bigger than $Degree(\phi(v))$ under mapping ϕ. If this condition does not stand, then G_Q cannot be a subgraph of G_D.

Now we take a look at Table 3, which shows the DA feature vectors for both G_Q and G_D.



(a) Query graph G_Q



(b) Data graph G_D

Figure 26: A graph and its supergraph

By observing Table 3, there is no mathematical relation between $F_d^*(G_Q)$ and $F_d(G_D)$. But $F_d^*(G_Q) \leqslant \sum_{j=d}^{MaxDegree(V_D)} F_j(G_D), \forall d \in [1, MaxDegree(V_Q)]$. **Lemma 1** and its proof are given below.

Lemma 1

If $G_Q(V_Q, E_Q)$ is a subgraph of $G_D(V_D, E_D)$ under graph mapping ϕ, then $F_d^*(G_Q) \leqslant \sum_{j=d}^{MaxDegree(V_D)} F_j(G_D), \forall d \in [1, MaxDegree(V_Q)]$.

Proof

Because G_Q is a subgraph of G_D, according to the subgraph matching definition in Subsection 3.5.2, we have:

$$\Lambda_v^* \supseteq \Lambda_{\phi(v)} \tag{13}$$

d	$F^*_d(G_Q)$	$F_d(G_D)$	$\sum_{j=d}^{MaxDegree(V_D)} F_i(G_D)$
1	< 3 , 1 >	< 3 , 5 >	< 18 , 14 >
2	< 1 , 0 >	< 6 , 7 >	< 15 , 9 >
3	< 5 , 2 >	< 4 , 2 >	< 9 , 2 >
4	< 0 , 0 >	< 5 , 0 >	< 5 , 0 >

Table 3: Feature vector comparison

From Equation 7 we have:

$$\textit{Min-Absolute}(\Lambda^*_v[i]) \leqslant |\Lambda_{\phi(v)}[i]|, \forall i \in [1, |\Lambda^*_v|] \tag{14}$$

Assume $v_{d1}, v_{d2}, ..., v_{dk}$ are vertices in V_Q that have the same degree d, according to Equation 14, $\forall d \in [1, MaxDegree(V_Q)]$,

$$\begin{aligned}
& \sum_{degree(v)=d} \Lambda^*[i] \\
= \ & \Lambda^*_{v_{d1}}[i] + \Lambda^*_{v_{d2}}[i] + ... + \Lambda^*_{v_{dk}}[i] \\
= \ & \textit{Min-Absolute}(\Lambda^*_{v_{d1}}[i]) + \textit{Min-Absolute}(\Lambda^*_{v_{d2}}[i]) + ... + \textit{Min-Absolute}(\Lambda^*_{v_{dk}}[i]) \\
\leqslant \ & |\Lambda_{\phi(v_{d1})}[i]| + |\Lambda_{\phi(v_{d2})}[i]| + ... + |\Lambda_{\phi(v_{dk})}[i]|
\end{aligned} \tag{15}$$

Since G_Q is a subgraph of G_D under graph mapping ϕ, we have

$$degree(v) \leqslant degree(\phi(v)), \forall v \in V_Q, \phi(v) \in V_D \tag{16}$$

From Equation 15 and 16, we can infer:

$$|\Lambda_{\phi(v_{d1})}[i]| + |\Lambda_{\phi(v_{d2})}[i]| + ... + |\Lambda_{\phi(v_{dk})}[i]| \leqslant \sum_{degree(v) \geqslant d} |\Lambda_v[i]| \tag{17}$$

From Equation 15 and 17,

$$\sum_{degree(v)=d} \Lambda^*[i] \leqslant \sum_{degree(v) \geqslant d} |\Lambda_v[i]| = \sum_{j=d}^{MaxDegree(V_D)} \sum_{degree(v)=j} |\Lambda_v[j]| \tag{18}$$

which can be written as:

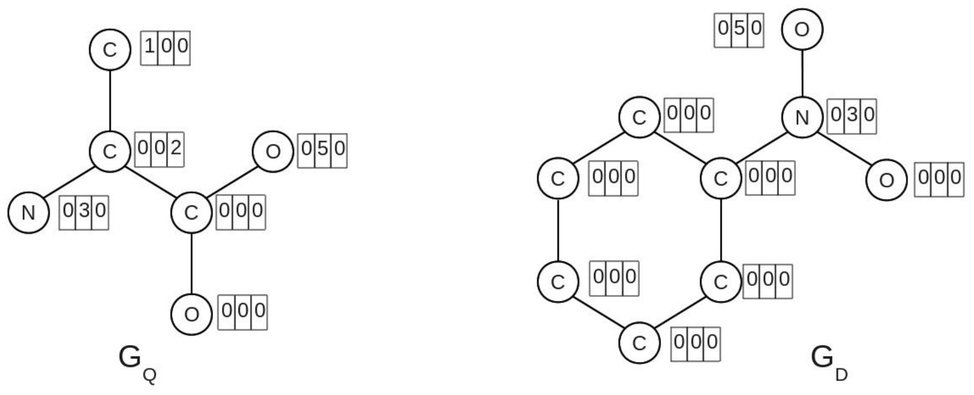
$$\sum_{degree(v)=d} \Lambda^* \leqslant \sum_{j=d}^{MaxDegree(V_D)} \sum_{degree(v)=j} \Lambda_v, \forall d \in [1, MaxDegree(V_Q)] \tag{19}$$

Also according to Equation 9 and 12, we then have:

$$F_d^*(G_Q) \leqslant \sum_{j=d}^{MaxDegree(V_D)} F_j(G_D), \forall d \in [1, MaxDegree(V_Q)] \tag{20}$$

QED

Accordingly, if the condition $F_d^* \leqslant \sum_{j=d}^{MaxDegree(V_D)} F_j, \forall d \in [1, MaxDegree(V_Q)]$ cannot be satisfied between a query graph $G_Q(V_Q, E_Q)$ and a data graph $G_D(V_D, E_D)$, the graph (graph-closure) is pruned. We name the pruning method **Degree-Attribute Pruning (DA-Pruning)**. We use the following example to demonstrate a graph which is pruned by the DA-Pruning.



d	$F_d^*(G_Q)$
1	<1,8,0>
2	<0,0,0>
3	<0,0,2>

d	$F_d(G_D)$	$\Sigma^3_{j=d} F_d(G_D)$
1	<0,5,0>	<0,8,0>
2	<0,0,0>	<0,3,0>
3	<0,3,0>	<0,3,0>

Figure 27: The DA-Pruning examples

In Figure 27 there are two chemical structures. G_Q is the graph form of *Alanine* from Figure 10(a), G_D is the graph form of a chemical *N,N-Dihydroxyaniline* [RSoC] with hydro atoms ignored. G_D is pruned using G_Q as a query by the DA-Pruning method according to Equation 20.

Algorithm 6: CalculateFeature(G)

```
input  : A graph $G(V, E)$
output: A feature vector $F$ if $G$ is a data graph, $F^*$ if $G$ is a query graph.
1  if $G$ is a data graph then
2      for $i \leftarrow 1$ to $|V|$ do
3          $\Lambda \leftarrow V[i].attributeVector$;
4          for $j \leftarrow 1$ to $|\Lambda|$ do
5              $F[V[i].degree][j] \leftarrow F[V[i].degree][j] + |\Lambda[j]|$ ;
6      return $F$;
7  else
8      for $i \leftarrow 1$ to $|V|$ do
9          $\Lambda^* \leftarrow V[i].attributeVector$;
10         for $j \leftarrow 1$ to $|\Lambda^*|$ do
11             Interval set $I \leftarrow \Lambda^*[j]$;
12             $F^*[V[i].degree][j] \leftarrow F^*[V[i].degree][j] + Min\text{-}Absolute(\mathrm{I})$ ;
13     return $F^*$;
```

Algorithm 7: DA-Pruning(G_Q, G_D)

```
input  : A query graph $G_Q$, a data graph $G_D$
output: A boolean value, true if $G_D$ not pruned, false if $G_D$ should be
        pruned
1  $F \leftarrow CalculateFeature(G_D)$, $F^* \leftarrow CalculateFeature(G_Q)$;
2  if $|F| < |F^*|$ then
3      return false;
4  for $i \leftarrow 1$ to $|F|$ do
       // To calculate sum for each element in $F$
5      $sum[i] \leftarrow \sum_{j=i}^{|F|} F[j]$;
6  for $i \leftarrow 1$ to $|F^*|$ do
7      if $sum[i] < F^*[i]$ then
8          return false;
9  return true;
```

There are two main processes in DA-Pruning: the feature vector calculating process and the pruning process. Algorithm 6 shows the process which outputs the feature vector for both a query graph and a data graph. According to Equation 8 and Equation 11, the Min-Absolute value of an attribute interval set is used in the calculation.

Algorithm 7 shows the pruning process for the DA-Pruning. In line 2, if $|F| < |F^*|$ then the node is pruned. This is because $|F| = MaxDegree(V_D)$, $|F^*| = MaxDegree(V_Q)$. If $MaxDegree(V_D) < MaxDegree(V_Q)$, then G_D can not be a supergraph for G_Q, and is thus pruned.

However, there is still a weakness in our pruning method. When the data graphs are large or the attribute values are big, but the query graph is relatively small or the absolute value of the attributes are small, the DA-Pruning might even slow down the query process because the pruning focuses on the difference of the vertex degrees and attributes. A vertex-closure in a graph-closure has a bigger chance that its degree and attribute are large since it is a bounding box of vertices.

4.3 Conclusion

In this chapter, we have provided a more flexible query format which allows the labels of the query graphs to be more complex. We also provided the modified related mapping algorithms. We develop a method to generalize a graph without know the exact labels of vertices which we call feature vectors. A novel pruning method which is developed based on the feature vector which is described as well as the proof of the correctness of the pruning. Detailed descriptions of the processing stages and algorithms are provided.

Chapter 5

Validation

In this chapter, we validate our index by real dataset and synthetic dataset experiments. We demonstrate that our index is more efficient for the new query format than CTree. We also demonstrate that our pruning method is effective when using non-fixed query labels. The validation of this research focuses on the effectiveness of ECTree itself and the pruning method.

The rest of this chapter is organized as follows: Section 5.1 introduces our experiment setup. Section 5.2 validates the ECTree index on its attribute organization. Section 5.3 validates the new query label format. Section 5.4 validates the DA-Pruning method for the ECTree index.

5.1 Experiment Setup

We implement ECTree based on the original implementation of CTree [He07] using C++. We conduct all the experiments on a Fedora 13 Linux machine with a dual processor Inter(R) Core(TM)2 1.8GHz with 2 GBytes RAM.

Datasets

For real datasets, we use the NCI Release 2 Files [NCI] in SDF file format [Sym]. The SDF format contains information of multiple chemical compounds with the name of the atoms and additional integer attributes such as the mass difference, atom charge and atom parity. We use a subset of the NCI Release 2 which contains 20,000 raw chemical structures. The average number of vertices is 17.65 and the average number of edges is 18.29. The number of distinct vertex labels is 45. We define the **attribute**

density of a graph $P = \frac{\#\ of\ non\text{-}zero\ attributes\ in\ G}{(\#\ of\ attributes\ in\ each\ vertex) \times (\#\ of\ vertices)}$. The attribute density of the NCI subset is $P_{NCI-subset} = 0.004$.

For synthetic datasets, we use the datasets which are used in [HLPY10] generated by GraphGen [CKN] in different densities. The density of a graph is defined as $\frac{\#\ of\ edges\ in\ G}{\#\ of\ edges\ in\ a\ complete\ graph}$. We use the dataset: 'Synthetic.10K.E30.D3.L50' where 10K is the size of the set, E30 means the average number of edges in the set is 30, D3 means the average density of the graphs is 0.3, L50 means there are 50 distinct vertex labels. Those graphs do not contain any additional attributes, so we use an algorithm to add random attributes to the synthetic datasets for the testing of ECTree. We vary P in testing the performance of our index against CTree, the number of attributes is always set to 3, and the maximum value of the attributes is varied as well in order to test the pruning power. The name 'Synthetic.10K.E30.D3.L50.P05.A25' is used to specify in addition that the synthetic dataset has an attribute density of 0.05 and maximum absolute attribute value of 25. Because all synthetic datasets are generated from 'Synthetic.10K.E30.D3.L50', so they all have the same number of graphs, average number of edges, graph density and number of distinct vertex labels. Thus we use the name 'Synthetic.P05.A25' instead only to specify the attribute density and maximum value of attributes. The algorithm to add the attributes to a graph can be found in Appendix B.2.

Query sets

For the NCI Release 2 subset, we use 4 query sets, each of which contains 10 substructures randomly chosen from the dataset for exact matching so they queries are ensured to be matched. For the synthetic datasets, we generate the queries from the datasets using randomly selected graphs and remove the vertices and related edges until the number of vertices meets the requirements. Thus queries for the synthetic dataset are ensured to be matched too. We also use 4 query sets for each synthetic dataset. The algorithm to generate the query set for synthetic dataset can be found in Appendix B.3. Table 4 shows statistics about the query sets.

We test ECTree against the original CTree. Both indexes are implemented using C++. In order to test both indexes fairly, we set up them as follows:

- Since CTree does not support the additional attributes, in order to get the same query results as ECTree, we use the subgraph isomorphism test and attribute

Query set name	Avg. # of vertices	Avg. # of edges	# of queries
NCI.Q1	5.9	5	10
NCI.Q2	9.1	8.5	10
NCI.Q3	12.1	11.2	10
NCI.Q4	14.8	15.1	10
Synthetic.Q1	4	3.52	50
Synthetic.Q2	6	7.16	50
Synthetic.Q3	8	9.53	50
Synthetic.Q4	10	16.78	50

Table 4: Query sets statistics

test which are used in ECTree to test the CTree **only in the verification phase**. That is, in the index building and query filtering phase, CTree still uses its own testing methods.

- Since CTree does not support the new query label format, we use fixed label queries for both indexes. The Histogram Pruning and the DA-Pruning are disabled in both indexes because we want to know the performance of the index using the new query format without pruning. Then we test them again with the same dataset enabling the Histogram Pruning. The DA-Pruning performance is tested separately.

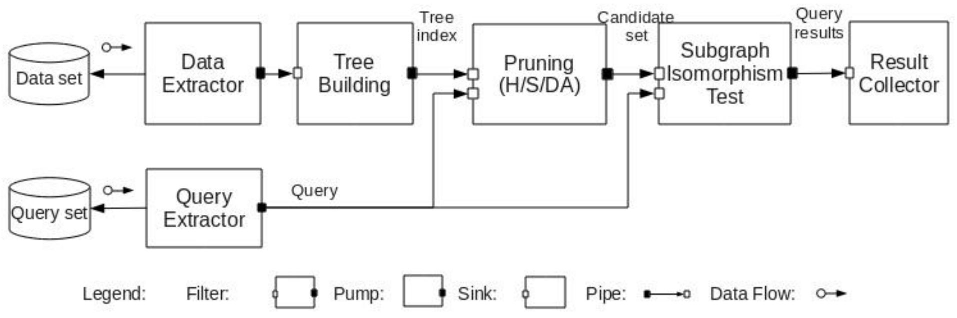


Figure 28: The testing process as a pipe-and-filtering framework

Figure 28 shows our testing framework. The letter 'H' stands for 'Histogram

Pruning', the letter 'S' stands for 'Pseudo-Subgraph Isomorphism Test', the letters 'DA' stand for 'DA-Pruning'. In the following discussions and experiments, we always enable the Pseudo-Subgraph Isomorphism Test by default. When we test the indexes using only fixed labeled queries, we enable the Histogram Pruning and denote the index by '+H'. We also test the performance of the indexes on our new query format but it is hard to generate those queries, so we still adopt the fixed labeled queries but we disable the Histogram Pruning (see Subsection 2.5.4) that cannot be used when we use the new query format. When we enable the DA-Pruning for ECTree, it is denoted as '+DA'.

5.2 Effectiveness of the Index

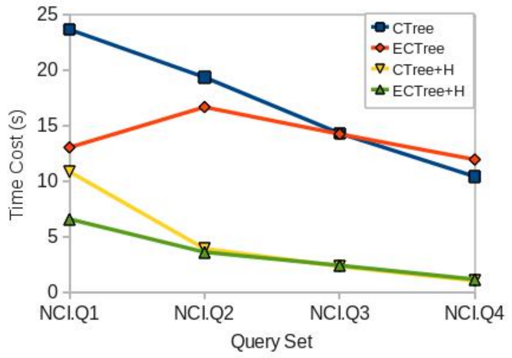
We first compare ECTree to CTree in terms of the results returned from the index to show the selectivity of our index. We take a look at the time to build the index using the NCI Release 2 subset for both CTree and ECTree in Table 5.

	CTree	ECTree
Number of Graphs	20k	20k
Input File Size(MB)	2.6	4.6
Index Size(MB)	3.4	5.5
Time Cost (s)	40.10	77.93

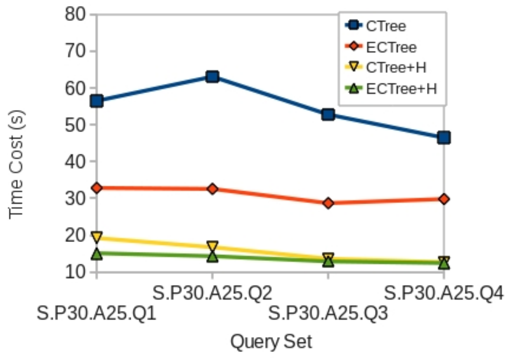
Table 5: Time cost to build index

With the same number of graphs, the size of the dataset that we use for ECTree is larger than the one for CTree because of the additional attributes for each atom. Consequently the index of ECTree is larger as well. The time to build the index for ECTree is almost two times the time to build the index for CTree. One reason is that according to the feature of ECTree, in order to maintain the structure that all the descendents of a graph-closure are subgraphs of this node, more calculations are included in building ECTree than CTree in the vertex/vertex-closure merging.

Figure 29(a) shows the comparison of time to compute the candidate set on the NCI dataset. When the query graph size gets larger, ECTree gradually costs more time than CTree because more graphs are likely to get pruned by the Pseudo-Subgraph



(a) NCI subset



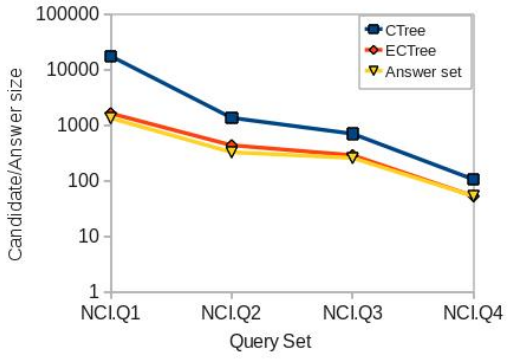
(b) Synthetic.P30.A25

Figure 29: Filtering time cost

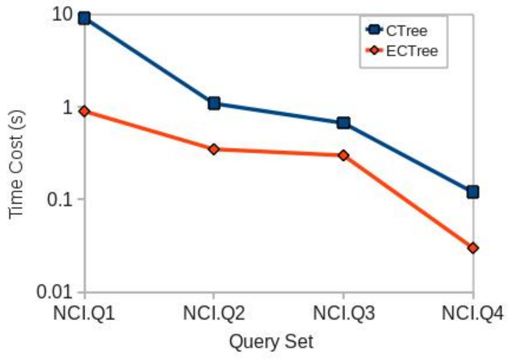
Isomorphism Test and then the rest of the graphs are likely to survive in the attribute test because for NCI dataset, the attributes of a vertex is usually one of a few values. For example, the attribute vector of the label ‘N’ in NCI dataset is usually either $\langle 0, 5, 0 \rangle$ or $\langle 0, 3, 0 \rangle$. Figure 29(b) shows the comparison for the synthetic dataset. Our method outperforms CTree significantly when querying the new format. When querying fixed-label queries, our method outperforms CTree slightly.

Figure 30(a) shows the candidate/answer set size for the NCI subset with respect to the querysets. As the subgraph-isomorphism problem is NP-complete, the less candidates we obtain after filtering, the faster query time we can achieve. The candidate set size of ECTree is always smaller than that of CTree, and the true-positive rate is nearly 100% for ECTree. When we enable the Histogram Pruning, the candidate results are the same because the Pseudo Subgraph Isomorphism (PSI) test is a more accurate pruning, which means some false-positive results that are not pruned by the Histogram pruning will be pruned by PSI test. The results show that our index helps prune more unwanted branches at higher level of the index. Figure 30(b) shows the verification time for both indexes.

We omit the diagram for synthetic datasets because the true-positive rate for both CTree and ECTree are close to 100% due to the randomness of the attributes – the answer set contains only one graph in most cases. Thus the verification time is very



(a) Candidate/Answer set size



(b) Verification time

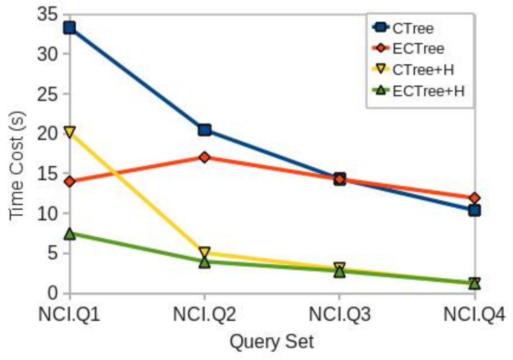
Figure 30: NCI subset verification statistics using CTree and ECTree

small and does not influence the whole querying process significantly.

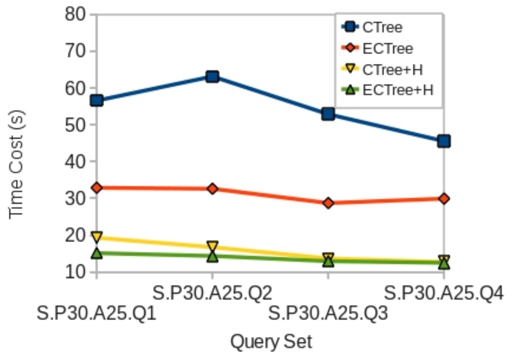
Figure 31 demonstrates the overall time for both datasets. Results show that our index outperforms the CTree in the original query format. For the new query format, our index gets slower than the CTree as the query graph size gets larger.

5.3 The New Query Label

Next we show the efficiency of the new format of query label. We firstly use the queries Q_1, Q_2, Q_3 and Q_4 described in Section 4.1.1 and 4.1.2 to compare the overall time. The DA-pruning is enabled and the Histogram-Pruning is disabled when regular expressions are used. Both prunings are enabled when running queries separately. The dataset NCI subset is used. Results are shown in Table 6. There is a small performance difference when we use the regular expressions on the labels because Q_2 returns only 1 result thus there is not much benefit from combining queries. We show the time enhancement with another example where more queries are combined. As is shown in Figure 32(a), the four queries are real chemical substructures extracted from the NCI subset. They can be combined into the query shown in Figure 32(b). The testing method is the same as the previous test. Results are shown in Table 7.



(a) NCI subset



(b) Synthetic.P30.A25

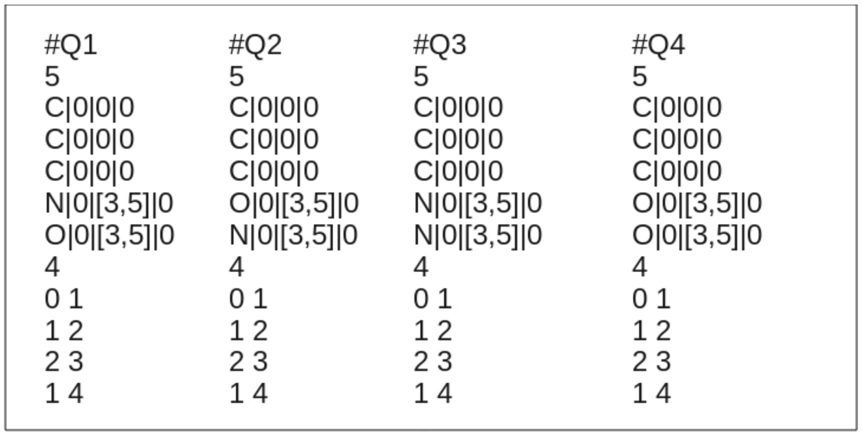
Figure 31: Overall time cost

Method	**Candidate set size**	**Answer set size**	**Overall run time (s)**
Run both queries separately (Q_1 and Q_2)	892	875	1.87
Using regular expressions (Q_3)	896	875	1.80
Using label groups (Q_4)	896	875	5.65

Table 6: Comparison of querying methods 1

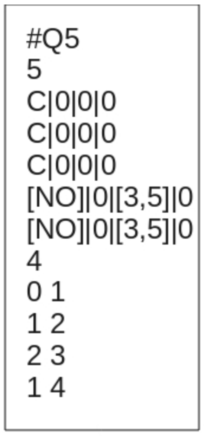
As we can see, though when separate queries are used, we can enable the histogram-based pruning as well as our DA-pruning, it is still slower in time compared to using a combined query in the condition. The enhancement for this test is 80% when our method is used.

When the label groups are used, time instead rises, which shows the cost for enumerating the compositions for the labels is very high.



```
#Q1           #Q2           #Q3           #Q4
5             5             5             5
C|0|0|0       C|0|0|0       C|0|0|0       C|0|0|0
C|0|0|0       C|0|0|0       C|0|0|0       C|0|0|0
C|0|0|0       C|0|0|0       C|0|0|0       C|0|0|0
N|0|[3,5]|0   O|0|[3,5]|0   N|0|[3,5]|0   O|0|[3,5]|0
O|0|[3,5]|0   N|0|[3,5]|0   N|0|[3,5]|0   O|0|[3,5]|0
4             4             4             4
0 1           0 1           0 1           0 1
1 2           1 2           1 2           1 2
2 3           2 3           2 3           2 3
1 4           1 4           1 4           1 4
```

(a)



```
#Q5
5
C|0|0|0
C|0|0|0
C|0|0|0
[NO]|0|[3,5]|0
[NO]|0|[3,5]|0
4
0 1
1 2
2 3
1 4
```

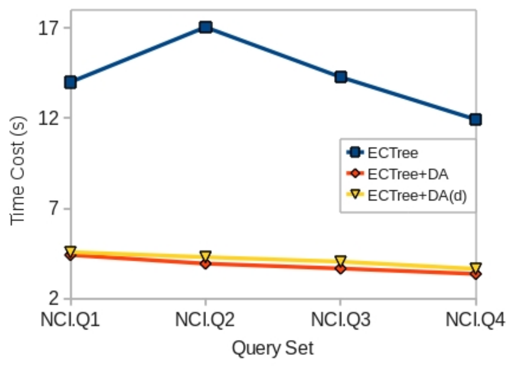
(b)

Figure 32: Four queries and a combined query

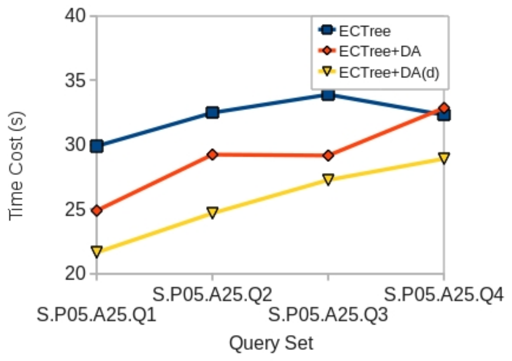
5.4 The Pruning Power

This section focuses on the DA-Pruning method for the new query format. We vary the value of the attribute density P for the synthetic dataset at 0.05, 0.15 and 0.3 to test the efficiency of our pruning. For the synthetic dataset $P = 0.3$ and the NCI subset, we use the same query sets used in the previous section. For synthetic dataset P=0.05 and P=0.15, we generate the query sets using the generating algorithm in Appendix B.3.

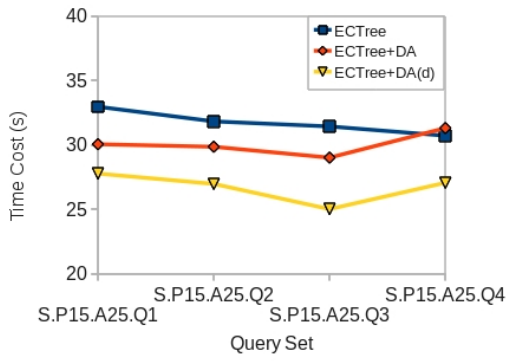
Our purpose is to test the pruning power when using the new query format. Histogram Pruning is always disabled and Pseudo Subgraph Isomorphism is always enabled. We first run the queries in ECTree without DA-Pruning, and then we run the queries again with DA-Pruning, at last we run the queries with DA-Pruning only at the data level of the index which means the pruning is not enabled at the non-leaf index level (to distinguish it from DA-Pruning, this method is named 'DA(d)'). Results show that for all datasets, enabling DA(d)-Pruning enhances the performance in terms of overall time. Enabling DA-Pruning gives better time than disabling it in most cases but fails to give enhancement in some cases. This is because as the attribute density increases, an index node in the ECTree is likely to have high attribute density, large attribute values and a high degree. Therefore it is less likely to get pruned by DA-Pruning at a non-leaf index level. For the NCI subset, DA-Pruning and DA(d)-Pruning do not have significant difference because the attribute density of the NCI subset is very low and attribute values are very limited.



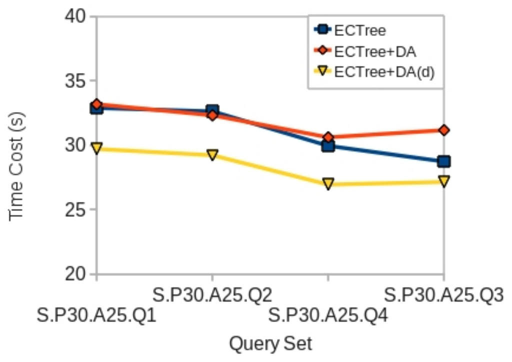
(a) NCI subset



(b) Synthetic.P05.A25



(c) Synthetic.P15.A25



(d) Synthetic.P30.A25

Figure 33: DA-Pruning on different attribute densities

Let t be the time cost without DA-Pruning, and t_{DA} be the time cost with DA-Pruning, we define the enhancement rate $r = \frac{t - t_{DA}}{t}$. Figure 34 shows the enhancement rate with respect to different query sets. We use the data collected from DA(d)-Pruning since it gives better results for all query sets. Results show that as P increases, the enhancement rate decreases. DA-Pruning does enhance the performance of the index in terms of the time with the synthetic datasets and the NCI subset but as the query graph size and P gets bigger the pruning power of the method gradually decreases.

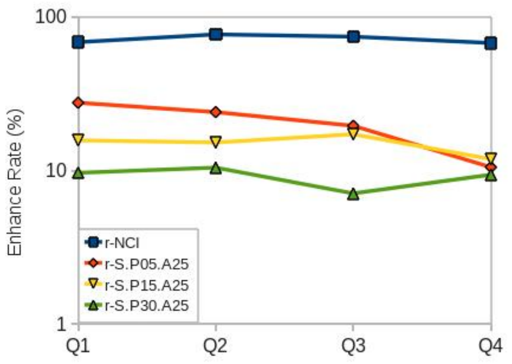


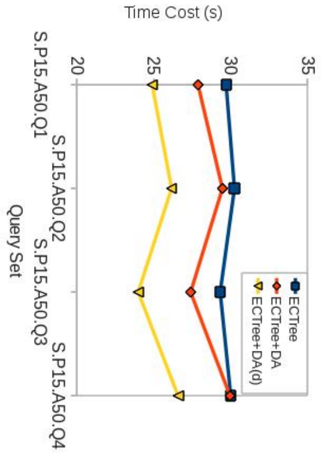
Figure 34: Overall time cost enhance rate using DA(d)-Pruning

We also vary the maximum absolute value of the attributes in order to see the change of enhancement rate of the DA-Pruning. Including the dataset 'Synthetic.P15.A25', seven more synthetic datasets are used. The query sets for those new datasets are generated using the same methods mentioned in Section 5.1.

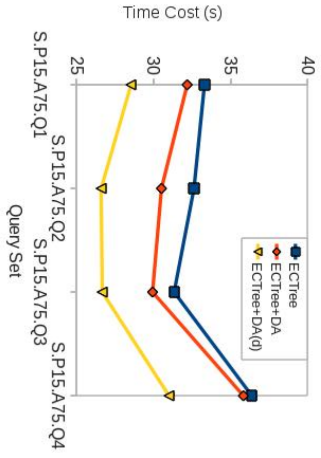
The results show that DA-Pruning fails to give enhancement for some query sets of Q4, however, DA(d)-Pruning always gives better run time for all query sets, as is shown in Figure 35. Table 8 shows the enhancement rates calculated using DA(d)-Pruning, which vary from 10.10% to 21.50% due to the randomness of selected queries, because when a query graph has relatively small values in its DA feature vector, it is less likely to benefit from the pruning, which is a weak point of our method. However, the average enhancement rates show no significant difference as the maximum value of the attribute grows, which can be concluded as our method is effective regardless of the maximum attribute value.

Method	**Candidate set size**	**Answer set size**	**Overall run time (s)**
Run four queries separately	13	13	3.04
Using regular expressions	13	13	0.62

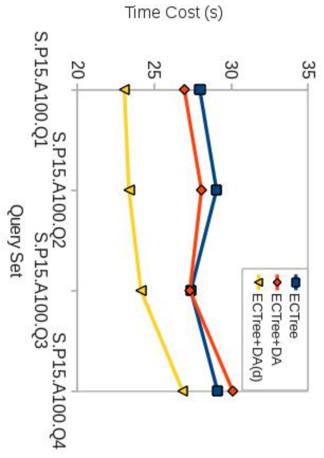
Table 7: Comparison of querying methods 2



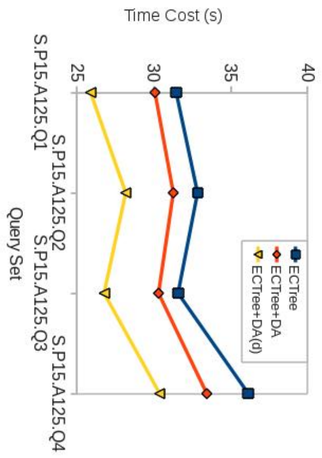
(a) Synthetic.P15.A50



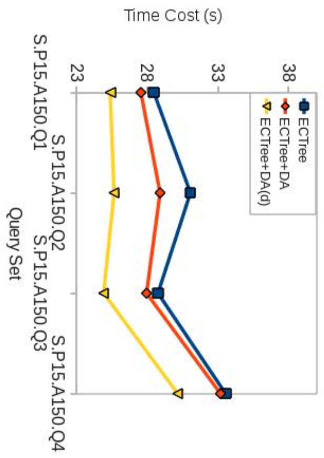
(b) Synthetic.P15.A75



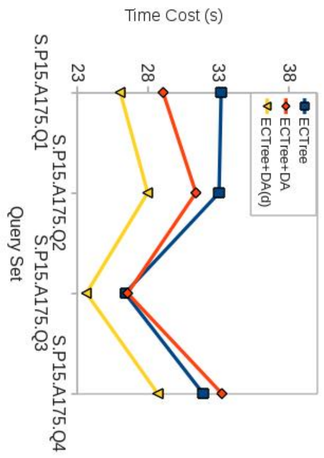
(c) Synthetic.P15.A100



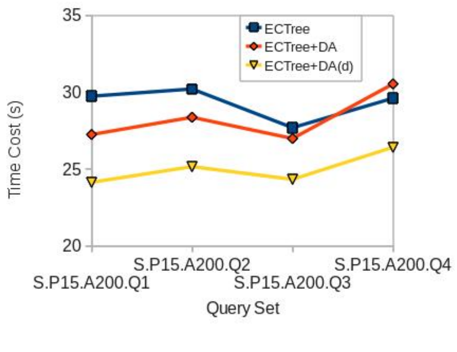
(d) Synthetic.P15.A125



(e) Synthetic.P15.A150



(f) Synthetic.P15.A175



(g) Synthetic.P15.A200

Figure 35: DA-Pruning on different maximum attribute values

Data Set Name	**Enhancement Rate on Q1**	**Enhancement Rate on Q2**	**Enhancement Rate on Q3**	**Enhancement Rate on Q4**	**Average Enhancement Rate**
Synthetic.P15.A25	15.74%	15.23%	20.42 %	11.88 %	15.81 %
Synthetic.P15.A50	15.89%	13.53%	18.18 %	11.43 %	14.76 %
Synthetic.P15.A75	14.32%	19.03%	15.56 %	15.90 %	16.20 %
Synthetic.P15.A100	17.56%	19.63%	11.88 %	9.89 %	14.21 %
Synthetic.P15.A125	17.71%	16.04%	15.23 %	15.93 %	16.22 %
Synthetic.P15.A150	10.89%	17.40%	13.38 %	10.10 %	12.94 %
Synthetic.P15.A175	21.50%	15.31%	10.40 %	10.10 %	14.32 %
Synthetic.P15.A200	18.82%	16.78%	12.23 %	10.77 %	14.63 %

Table 8: Enhancement rates using DA(d)-Pruning on synthetic datasets

Chapter 6

Related Work

The graph model has been widely used and has attracted much attention of research. There have been numerous graph indexes presented in recent research, but there are no studies of graph indexes that allow the data/query graphs to have numbers/strings as attributes. To the best of our knowledge no studies of multiple numerical attributed graph indexing strategies have been done prior to this thesis work. In this chapter, we review the recent and/or major works that are related to graph database indexing.

The *gIndex* [YYH04] and *FG-Index* [CKNL07] are both frequent subgraph based approaches. *gIndex* firstly generates the frequent subgraphs of size up to $maxL$; the resulting frequent graphs are sequentialized into a unique sequence and inserted into a prefix tree. In the query phase, *gIndex* enumerates all its fragments up to a maximum size and locates them in the index. The ID lists associated to the fragments are intersected as the candidate set for the latter subgraph-isomorphism test. *FG-Index* generates frequent subgraphs regardless of the size as well as an distinct edge index. If a query is a frequent graph, *FG-Index* returns the answer set without verification. Otherwise, *FG-Index* returns a candidate set according to the subgraphs of the query graph and its frequent/infrequent edges, if any. The frequent subgraph based approaches perform very well when the queries are frequent subgraphs in the data set.

The *TreePi* [ZHY07] and *Tree*+Δ [ZYY07] are tree structure feature based approaches. They both use frequent sub-trees as indexing structures rather than subgraphs because trees are easier to manipulate. *TreePi* also adopts a method called

Center Distance Constraints to prune the search space. $Tree{+}\Delta$ is an index structure with tree-features plus a small number of discriminative graphs. It is proved that the most frequent features are non-linear trees and generates all frequent trees in the offline process by [ZYY07]. In the query phase, $Tree{+}\Delta$ generates the graph features on demand in order to improve its pruning power. The graph feature generation starts from choosing each simple cycle from a query graph and extends it by one vertex at a time and checks whether the extended graph is discriminative to the previous one with respect to the supergraphs that are associated in the dataset. If the extended graph qualifies, it is added to Δ. The process is repeated until the size of the vertex set reaches a predefined maximum value.

A recent work [KKM11] proposed *CT-Index* which uses exhaustive enumeration of cycles and trees as features. All trees and cycles of a graph not exceeding a specified maximum size are enumerated and transformed into unique *canonical forms*, then stored using a hash-key fingerprint system [Day]. The size of the index and the size of the fingerprint can be adjusted to control collisions of elements in the fingerprint system. In the graph mining phase, *CT-Index* first enumerates features of the query graph and the database graph separately, and then runs an inexpensive bitwise AND-operation using fingerprints extracted from the fingerprint system via a hash-function using the enumerated features for filtering. For verification, a new backtracking algorithm similar to VF2[CFSV04] is presented.

[SZLY08] uses *QI-Sequence* for bounding the search space in the subgraph isomorphism test for a given query graph. Then a novel index called $Swift$*-Index* where the mined frequent tree features are represented as *QI-Sequences* and are organized as a prefix tree. In the filtering phase, the cost of subgraph isomorphism test can be largely reduced due to the sharing of structure of the prefix tree index. A new subgraph-isomorphism testing algorithm called *QuickSI* is also introduced.

[GS02] proposed *GraphGrep*, which builds a database to represent the graphs as sets of paths in the offline index construction. Each query graph is parsed into several paths and those paths are sent to the index for filtering graphs that clearly do not contain any occurrences of the query. The remaining graphs are candidates for the final answer set and are sent to the subgraph-isomorphism test. Since a path does not contain structural information of a graph, many false positive answers could be returned in the candidate set. When the graphs and queries in use are relatively complex, the path-based approach is not suitable.

The *Closure-Tree* [HS06] uses a graph bounding box method to organize the tree index structure called *CTree*. A *CTree* has graphs as its nodes, and it is a hierarchical tree that each node is a bounding graph of its descent nodes. In the query process, if a node is disqualified for a query, all graphs recursively contained by this node are pruned. A pruning method called *Pseudo Subgraph Isomorphism* is also introduced. In *ECTree*, we have extended the concept of bounding graph to the numerical attributes of the vertices in the graphs.

The *gCode* [ZCYL08] is a two-step filtering at both the index level and the object level. *gCode* computes a signature from a combination of neighborhood information for each vertex of every graph in the data set. The vertex signatures are later made into a graph signature as a feature for the graph and indexed in a tree called *gCode-Tree*. *gCode* also maintains a list of binary vectors <*signatureID, count*> for each distinguished vertex signature and how many times this signature appears in this graph. For each query graph, *gCode* extracts the graph signature of the query and finds all qualified signatures from *gCode-Tree* as the index filtering step. The qualified graphs are then sent to the object pruning that is done according to the list of pairs of vertex signatures to perform a "vertex-to-vertex" comparison filtering. The subgraph-isomorphism test is performed at the final step.

Summarization Graph [ZCZ+08] uses a novel subgraph searching algorithm based on *Summarization Graph Model*. Frequent subgraphs are first selected using a feature selecting algorithm, then each occurrence of frequent subgraphs in a dataset graph is summarized as a vertex. Each vertex in a summarization graph is a set of pairs denoted as <*Label, Length*>, where *Label* is the ID of the feature subgraph, and *Length* is an integer value that captures the *distance between occurrences* which is calculated according to the topology of the dataset graph. Each summarization graph is a complete graph with unlabeled edges. In the query process, a method of retrieving objects with set-valued attributes is used to obtain the candidate set.

Other interesting related work includes *GString* [JWYZ07], *GDI* [WHW07], *iGraph* [HLPY10] and *GiS* [PR11]. [JWYZ07] proposes *GString* for chemical compound databases. However, extending *GString* to other graph database applications is not straightfoward. [WHW07] proposes a method to enumerate all connected induced subgraphs in the graph database, then organizes them into a *Graph Decomposition Index* (*GDI*). A *GDI* contains a graph database Directed Acyclic Graph (DAG) which is merged from the graph decomposition DAGs of all the database graphs, and a hash table that cross-references nodes in the database DAG. This method does not work well with relatively large graphs because of the explosion of enumerations of subgraphs. [HLPY10] gives a framework to compare disk-based graph indexes, namely *iGraph*. A number of recent graph database indexes are implemented and a number of indicators such as the number of disk I/Os, elapsed time, etc. are compared against each other on both real and synthetic data sets. [HPL+11] further presents the visual tools for *iGraph* using several real datasets and their workloads. Similar to *iGraph*, the *GiS* [PR11] is a tool for indexing and querying a large database of labeled, undirected graphs. *GiS* supports various recent indexing techniques and provides both exact and approximate graph queries.

Chapter 7

Conclusion

This chapter gives conclusions of the thesis work, states the limitations and outlines potential future directions.

7.1 Summary of Contributions

This section describes the contributions that have been made in this thesis work. It describes our approaches to build up a novel graph index and differences from other indexes.

We have a graph database indexing system *ECTree* which is based on the design of Closure-Tree [HS06] that defines an extended data type of graphs which includes both labels and attributes on the vertices. The numerical attributes of a vertex is usually ignored when considering the graph indexing problems, but our approach tries to organize the labels and attributes at the same time. We adopt the concept of bounding box which is used in Closure-Tree to obtain a graph-closure into our method of indexing the attributes. We also provide a query format to query our new index with integer intervals as vertex attributes. We show the efficiency of our index when querying a graph database with the new graph format against the Closure-Tree. Our approach reveals that it is possible to index both labels and attributes at the same time and to improvement the speed.

A more flexible query format for our index which allows the labels of the query vertices to be non-fixed is applied to our index. The use of *Regular Expressions* allows flexible and complex query labels. The definition for Label Groups is provided. The new query format helps in reducing the query time when a group of queries share some common structures and information such as graph structures and labels. Running each of the queries may require many more subgraph-isomorphism tests than just running the combined query.

We also develop a pruning method named DA-Pruning which is used for the new query format that supports non-fixed query labels when querying attributed graph databases or when certain query labels are unknown. It is a pruning method based on the joint information from both the degree information and the attribute information of vertices. We describe in detail the methods to calculate the DA-feature vector for a data graph/graph-closure and for a query graph and also give the algorithms for the method. Comparisons are presented between DA-Pruning on the entire index and implementing it only on the data level which is named DA(d)-Pruning. We demonstrate that our pruning is effective for random queries extracted from the datasets by experiments on both real and synthetic datasets.

7.2 Limitations

There are some weak points in our work. Summarized as follows.

Our graph database index ECTree yet does not support large and complex graphs well (e.g. graphs that have more than 15 vertices and/or have a high density), due to the characteristic of a bounding box. When a graph database contains mostly large and complex graphs, the dead space in a graph-closure becomes very large and unacceptable and therefore reduces the performance of the index.

The sizes of the query sets and data sets are small. The NCI Release 2 contains 250K structures and we only used 20K as the real dataset. The testing factors are simple and more factors can be added and more tests can be made using various datasets and data types to further evaluate the index. For example, social networks, co-authorship networks, and so on.

The performance study is between ECTree and Closure-Tree only. More graph database indexes can be extended to adopt the new data and query formats so that they can be compared to our index.

7.3 Future Work

There are a few areas that we think are worthy looking into for further improvement and investigation.

Research towards extending other graph indexes to support both label indexing and attribute indexing can be an interesting direction. Our approach shows that indexing both labels and attributes is feasible and efficient. There are a lot of other graph indexes that are mentioned in Chapter 6 which can be considered for extending and further compared to this work.

We did not study incremental maintenance and updates for the index when a large number of graphs is involved. Closure-Tree has methods to maintain the index structurally only. If maintenance needs to be performed for ECTree on inserting, modifying or removing, finding a fast way for those operations can be an improvement.

ECTree only supports the integer type of attributes. Still, some other types may be considered such as boolean, float, or even string type. However, more complex data types involved means a more complex index. Whether our index can be extended to include such data types remains an open question.

There might also be interest on similarity queries with attributed graphs. The Closure-Tree index supports similarity queries but in our work we did not extend it for similarity queries with attributed graphs. The graph index for attributed similarity graph queries can be an interesting direction of research.

Reducing the structural dead space and the attribute dead space can benefit EC-Tree for performance enhancement. Though we adopt the Naive Biased Mapping for the data graph mapping phase, the structural dead space is still a big problem that prevents CTree and ECTree to be competitive in the large graph indexing realm. Moreover, ECTree further has the attribute dead space which is produced when merging the attribute vectors. The index could be more efficient and effective if a proper way of reducing dead space can be found.

Bibliography

[AEP09] H.M. Afsar, M.-L. Espinouse, and B. Penz. Building flight planning for an airline company under maintenance constraints. *Journal of Quality in Maintenance Engineering*, 15(4):430–443, 2009.

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):39, 2008.

[AW10] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Springer, 1st edition, February 2010.

[BK09] Joonhyun Bae and Sangwook Kim. A global social graph as a hybrid hypergraph. *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 1025–1031, 2009.

[BL] Paul E. Black and Alen Lovrencic. Subgraph. Dictionary of Algorithms and Data Structures, `http://www.nist.gov/dads/HTML/subgraph.html`. Accessed in April, 2011.

[BV99] H. Bunke and M. Vento. Benchmarking of graph matching algorithms. *2nd IAPR-TC15 workshop on graph-based representations*, pages 109–144, 1999.

[BWWZ06] G. Butler, Guang Wang, Yue Wang, and Liqian Zou. Query optimization for a graph database with visual queries. *Database Systems for Advanced Applications. 11th International Conference*, pages 602–616, 2006.

[Cac] Cacycle. Guanosine monophosphate. WIKIPEDIA, `http://en.wikipedia.org/wiki/Guanosine_monophosphate`. Accessed in June, 2011.

[CFSV04] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[CG70] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, Jan 1970.

[Cha85] Gary Chartrand. *Introductory Graph Theory.* Dover, 1985.

[CKN] James Cheng, Yiping Ke, and Wilfred Ng. GraphGen — a synthetic graph data generator. `http://www.cse.ust.hk/graphgen/`. Accessed in May, 2011.

[CKNL07] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. FG-Index: Towards verification-free query processing on graph databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 857–872, 2007.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. *Proc. 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Day] Daylight Chemical Information Systems. Daylight theory manual. `http://www.daylight.com/`. Accessed in August, 2011.

[Deb] Debstar. S.pombe pop2p protein structure rainbow. WIKIMEDIA COMMONS, `http://commons.wikimedia.org/wiki/File:Spombe_Pop2p_protein_structure_rainbow.png`. Accessed in June, 2011.

[DNH+92] A. Dalby, J.G. Nourse, W.D. Hounshell, A.K.I. Gushurst, D.L. Grier, B.A. Leland, and J. Laufer. Description of several chemical structure file formats used by computer programs developed at molecular design limited. *Journal of Chemical Information and Computer Sciences*, 32(3):244–255, 1992.

[eI] eMolecules Inc. eMolecules. `http://www.emolecules.com/`. Accessed in July, 2011.

[Epp95] David Eppstein. Subgraph isomorphism in planar graphs and related problems. *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 632–640, 1995.

[FB08] M. Fiedler and C. Borgelt. Subgraph support in a single large graph. *ICDM Workshops. 2007 7th IEEE International Conference on Data Mining Workshops*, pages 399–404, 2008.

[FGMP09] M. Franceschet, D. Gubiani, A. Montanari, and C. Piazza. From entity relationship to XML schema: a graph-theoretic approach. *Database and XML Technologies. Proceedings 6th International XML Database Symposium*, pages 165–179, 2009.

[Fri97] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly Media, 1997.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. *Proceedings of the 3rd IAPR-TC15 Workshop on Graph based Representation, Italy*, pages 188–199, 2001.

[GBL95] M. Graves, E.R. Bergeman, and C.B. Lawrence. Graph database systems. *IEEE Engineering in Medicine and Biology Magazine*, 14(6):737–745, 1995.

[Gen] Genome Programs. `http://genomics.energy.gov`. Accessed in April, 2011. The U.S. Department of Energy.

[GNU] GNU. GNU C Library. `http://www.gnu.org/s/libc/`. Accessed in June, 2011.

[Goo05] Nathan Good. *Regular Expression Recipes for Windows Developers: A Problem-Solution Approach.* Apress, 2005.

[GS02] Rosalba Giugno and Dennis Shasha. GraphGrep: A fast and universal method for querying graphs. *Proceedings - International Conference on Pattern Recognition*, 16(2):112–115, 2002.

[He07] Huahai He. *Querying and Mining Graph Databases*. PhD thesis, University of California, Santa Barbara, 2007.

[HK71] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *Switching and Automata Theory, 12th Annual Symposium on*, pages 122–125, 1971.

[HLPY10] Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1):449–459, 2010.

[HPL+11] Wook-Shin Han, Minh-Duc Pham, Jinsoo Lee, Romans Kasperovics, and Jeffrey Xu Yu. iGraph in action: Performance analysis of disk-based graph indexing techniques. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1241–1242, 2011.

[HS06] Huahai He and Ambuj K. Singh. Closure-tree: An index structure for graph queries. *Proceedings of the 22nd International Conference on Data Engineering*, page 38, 2006.

[IEE] IEEE. IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). `http://standards.ieee.org/findstds/standard/1003.1-2008.html`. Accessed in June, 2011.

[JWYZ07] Haoliang Jiang, Haixun Wang, P.S. Yu, and Shuigeng Zhou. GString: a novel approach for efficient search in graph databases. *2007 IEEE 23rd International Conference on Data Enginering*, page 10, 2007.

[KEG] KEGG. Kegg pathway database. `http://www.genome.jp/kegg/pathway.html`. Accessed in April, 2011.

[KKM11] Karsten Klein, Nils Kriege, and Petra Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. *Proceedings 2011 27th IEEE International Conference on Data Engineering*, pages 1115–1126, 2011.

[Kru01] Robert Krulwich. *Cracking the Code of Life*. Public Broadcasting Service, April 2001.

[LWSO04] Zhenjiang Li, Honggui Wan, Yuhu Shi, and Pingkai Ouyang. Personal experience with four kinds of chemical structure drawing software: review on ChemDraw, ChemWindow, ISIS/Draw, and ChemSketch. *Journal of Chemical Information and Computer Sciences*, 44(5):1886–1990, 2004.

[Mel96] J. Melton. SQL language summary. *ACM Computing Surveys*, 28(1):141–143, March 1996.

[NC05] David L. Nelson and Michael M. Cox. *Principles of Biochemistry.* New York: W. H. Freeman, 4th edition, 2005.

[NCI] NCI/CADD Group. CADD group chemoinformatics tools and user services. `http://cactus.nci.nih.gov/download/nci/`. Accessed in April, 2011.

[NK04] S. Nijssen and J.N. Kok. Frequent graph mining and its application to molecular databases. *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, 5:4571–4577, March 2004.

[PK96] J. Pospichal and V. Kvasnicka. Pruning the search tree in the constructive enumeration of molecular graphs. *Discrete Applied Mathematics*, 67(1-3):189–207, 1996.

[PLM08] A.N. Papadopoulos, A. Lyritsis, and Y. Manolopoulos. SkyGraph: an algorithm for important subgraph discovery in relational graphs. *Data Mining and Knowledge Discovery*, 17(1):57–76, Aug 2008.

[PR11] Dipali Pal and Praveen R. Rao. A tool for fast indexing and querying of graphs. *Proceedings of the 20th International Conference Companion on World Wide Web*, pages 241–244, 2011.

[Rav] Ravidreams. Gene. WIKIPEDIA, `http://en.wikipedia.org/wiki/Gene`. Accessed in June, 2011.

[RSoC] Cambridge Royal Society of Chemistry. Chemspider. `http://www.chemspider.com/`. Accessed in February, 2012.

[SWG02] D. Shasha, J.T.-L Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. *In PODS*, pages 39–52, 2002.

[Sym] Symyx Solutions Inc. CTfile Formats. `http://www.symyx.com/downloads/public/ctfile/ctfile.pdf`. Accessed in June, 2011.

[SZLY08] H. Shang, Y. Zhang, X. Lin, and J.X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):364–375, 2008.

[Ull76] J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[Vic] Tim Vickers. NAD metabolism pathway. WIKIPEDIA, `http://en.wikipedia.org/wiki/File:NAD_metabolism.svg`. Accessed in June, 2011.

[Wan10] Yue Wang. *On visual queries and graph databases with application to genomics*. PhD thesis, Concordia University, Montreal, Quebec, 2010.

[WC10] Zhisong Wang and Ran Chai. A new algorithm for mining maximal frequent subgraph. *Journal of Computational Information Systems*, 6(2):469–476, Feb 2010.

[WHW07] D.W. Williams, Jun Huan, and Wei Wang. Graph database indexing using structured graph decomposition. *Data Engineering, ICDE. IEEE 23rd International Conference on*, pages 976–985, June 2007.

[Wig] Adam Wiggins. Graph databases. `http://adam.heroku.com/past/2010/3/15/graph_databases/`. Accessed in April, 2011.

[YYH04] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 335–346, 2004.

[ZCYL08] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. A novel spectral coding in a large graph database. *Proceedings of the 11th International Conference on Extending Database Technology*, pages 181–192, 2008.

[ZCZ+08] Lei Zou, Lei Chen, Huaming Zhang, Yansheng Lu, and Qiang Lou. Summarization graph indexing: beyond frequent structure-based approach. *Database Systems for Advanced Applications. Proceedings 13th International Conference*, pages 141–155, 2008.

[ZHY07] Shijie Zhang, Meng Hu, and Jiong Yang. TreePi: a novel graph indexing method. *IEEE 23rd International Conference on Data Engineering*, pages 966–975, 2007.

[ZYT] ZYTRAX. Regular Expressions - User Guide. ZYTRAX Communications, `http://www.zytrax.com/tech/web/regex.htm`. Accessed in June, 2011.

[ZYY07] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: tree + delta >= graph. *VLDB '07 Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 938–949, 2007.

Appendices

Appendix A

Meaning of Values in SDF Atom Block

This appendix presents the detailed SDF atom block meaning introduced in [Sym]. An atom block is made up of atom lines, one line per atom with the following format:

```
xxxxx.xxxx yyyyy.yyyy zzzzz.zzzz aaaddcccssshhhbbbvvvHHHrrriiimmmnnneee
```

We show part of the meaning of values in SDF atom block in Table 9. In our experiments, we only use mass difference, charge and atom stereo parity as attributes. And in fact most of the non-zero values appear in those attributes and that is why we keep three attributes for each vertex in our experiments. The manual CTfile Formats [Sym] can be referred to for full explanations of the SDF file format.

Field	**Meaning**	**Values**	**Notes**
x y z	atom coordinates		
aaa	atom symbol	Entry in periodic table or L for atom list, A, Q, * for unspecified atom, and LP for lone pair, or R# for Rgroup label	
dd	mass difference	-3, -2, -1, 0, 1, 2, 3, 4 (0 if value beyond these limites)	Difference from mass in periodic table. Wider range of values allowed by **M ISO** line, below. Retained for compatibility with older CTabs, **M ISO** takes precedence.
ccc	charge	0 = uncharged or value other than these, 1 = +3, 2 = +2, 3 = +1, 4 = doublet radical, 5 = -1, 6 = -2, 7 = -3	Wider range of values in **M CHG** and **M RAD** lines below. Retained for compatibility with older CTabs, **M CHG** and **M RAD** lines take precedence.
sss	atom stereo parity	0 = not stereo, 1 = odd, 2 = even, 3 = either or unmarked stereo center	Ignored when read.

Table 9: Meaning of values in the atom block [Sym]

Appendix B

Related Algorithms

B.1 Counting Different Labels and Finding New Bindings

Algorithm 8 shows the function *CountVertex*(V) and Algorithm 9 shows the function *FindNewBinding*(B, c), which are used in Subsection 4.1.3.

Algorithm 8: CountVertex(V)

```
input  : A vertex vector $V$
output: A vertex vector $V'$
$V' \leftarrow \emptyset$ ;
for $i \leftarrow 1$ to $|V|$ do
    if $V[i] \notin V'$ then
        Add $V[i]$ to $V'$ ;
return $V'$;
```

```
Algorithm 9: FindNewBinding($B$, $c$)
input  : An integer array indicating the current binding $B$, number of
         different labels to bind $c$
output: A new binding array $B'$

$B' \leftarrow \varnothing$ ;
$i \leftarrow |B|$;
while true do
    if $B[i] + 1 \leqslant c$ then
        $B[i] \leftarrow B[i] + 1$;
        foreach $j \in [1, c]$ do
            if j appears twice or more in B then
                continue while ;
        $B' \leftarrow B$;
        break while;
    else
        $B[i] \leftarrow 1$;
        $i \leftarrow i - 1$ ;
        if i equals 0 then
            break while ;
return $B'$;
```

B.2 Adding Attributes to Non-Attributed Graphs

Algorithm 10 adds integers as attributes to vertices. The system time is used as a seed for random integers. In our programming *srand*(*unsigned*(*time*(0))) is used for seeding and *rand*() is used to generate random integers.

```
Algorithm 10: AddAttribute($S$, $n$, $P$, $M$)
input  : A set of non-attributed graphs $S$, number of attributes to be
         added $n$, attribute density $P$, the maximum absolute value of the
         attributes $M$
output: A set of attributed graphs $S'$

Seed the random number generator to system time;
foreach graph $G(V, E) \in S$ do
    $V' \leftarrow \emptyset$;
    foreach vertex $v \in V$ do
        for $i \leftarrow 1$ to $n$ do
            $x \leftarrow$ a random integer;
            $x \leftarrow x \bmod 100$;
            $y \leftarrow 0$;
            if $x < P$ then
                $y \leftarrow$ a random integer, $y \leftarrow y \bmod M$;
                $z \leftarrow$ a random integer;
                if $z \bmod 2$ equals 0 then
                    $y \leftarrow -y$;
            Add $y$ to the attribute list of $v$;
        Add $v$ to vertex set $V'$;
    Add $G'(V', E)$ to $S'$;
return $S'$;
```

B.3 Generating Query Sets for Synthetic Datasets

Although our index supports disconnected graph queries, we generate only connected graphs as queries to avoid possible problems in indexing and querying, as is shown in Algorithm 11.

Algorithm 11: GenerateSyntheticQueryset(S, n, m)

```
input  : A set of attributed graphs $S$, number of queries to be generated $n$,
         size of vertex set of each query $m$,
output: A set of queries $Q$

Seed the random number generator to system time;
$Q \leftarrow \varnothing$;
while $|Q| < n$ do
    $x \leftarrow$ a random positive integer;
    $x \leftarrow x\ mod\ |S|$;
    $G(V, E) \leftarrow S[x]$;
    if $|V| < m$ then
        continue;
    else
        while $|V| > m$ do
            Delete a random vertex $v$ from $V$;
            Delete all edges related to $v$;
            if $G$ is a disconnected graph then
                Roll back the deletion of vertex and edges;
                continue;
        if $\forall v \in V$, $v$ has only zero attributes then
            continue;
    Add $G'(V, E)$ to $Q$;
return $Q$;
```