On incremental maintenance of 2-hop labeling of graphs

Bramandia Ramadhana

2009

Bramandia, R. (2009). On incremental maintenance of 2-hop labeling of graphs. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/42234>

<https://doi.org/10.32657/10356/42234>

ON INCREMENTAL MAINTENANCE OF 2-hop LABELING OF GRAPHS

BRAMANDIA RAMADHANA

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2009

Acknowledgements

My ultimate gratitude goes to my supervisor, Prof. Byron Choi. I am thankful to him for providing me the opportunity to do a fully-funded research in a truly interesting field. His passion and strong interest in research have influenced me to become a self-motivated student with a strong interest in my research topic. His guidance, encouragement and patience have helped me to go through the hard period of research life. He has devoted endless hours of discussing ideas and solutions, as well as writing and editing manuscripts. His tireless effort contributes vastly to the quality of this thesis as well as our joint publications. Finally, during my research period, he has taught me much of invaluable things about what research is.

I am grateful to my wife for her love and care which have helped me to endure the hard time of the thesis writing process. Her patience and sincerity were significant contributions to the completion of this thesis.

I would like to express my gratitude to Prof. Lim Ee Peng and Prof. Lee Bu Sung who have granted me a chance to pursue the Master program (by Research) in School of Computer Engineering, NTU.

This is a great opportunity to express my respect to Prof. Sourav and Prof. Vivek for their numerous advice and encouragements which keep my spirit and courage at the highest level to go through the research life.

I am pleased to thank my colleagues in Centre of Advanced Information Systems (CAIS): Ardian, Erwin, Maggy, Fajar, Maureen, Linhong as well as many others for the discussions, helps as well as some leisure talks and jokes. They have been good friends of mine during my work in CAIS. Special credit goes to Ardian, with whom I have some through discussions and exchange of ideas.

I thank Chee Keong and Chiew Song for their helps and supports in making

CAIS a comfortable and effective place to spend my 2 years of research life.

Special thanks to my family. I could not have come this far without their endless love and support. I am also thankful for their sincere prayers for me that have helped me to achieve what I have achieved currently.

I am grateful to Prof. Jeffrey Xu Yu and Dr. Jiefeng Cheng for the implementation of *2-hop* labels of [STW05] and [CYL⁺06] that have been used thoroughly for the experimental evaluation.

I would like to thank other people whose name is not mentioned here for their direct or indirect supports that contribute to the completion of this thesis.

Above all, I would like to present my deepest gratitude to Allah - praise be unto Him, Lord of the Worlds. To date, He has gifted me uncountable amount of favors in this world including but not limited to faith, life, health, intelligence, and passions. Only through His permission I could complete this thesis. I pray for Allah to make the work purely for His sake and to reward me for it out of His generosity.

Contents

Acknowledgements	ii
Abstract	xi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Contributions	6
1.3 Thesis Outline	8
2 Background	9
2.1 Background on XML	9
2.2 Definition of 2-hop labels	13
2.3 Construction of 2-hop cover	15
3 Literature Review	18
3.1 Indexes for reachability query	18
3.2 Construction of 2-hop labels	25
3.2.1 Works by Ralf Schenkel et al.	26
3.2.2 Work by Jiefeng Cheng et al. (2006)	27
3.2.3 Work by Jiefeng Cheng et al. (2008)	29
3.3 Maintenance of 2-hop labels	30
3.3.1 Work by Ralf Schenkel et al.	30
3.3.2 Work by Bramandia et al.	32
3.4 Summary	36
4 Properties for Updatable 2-hop (u2-hop) Labeling	38
4.1 Analysis of Updates of 2-hop Labeling	38

4.2	Properties for Updatable 2-hop (u2-hop) Labeling	41
4.2.1	Node separation property	42
4.2.2	Merging Property	43
5	Incremental Maintenance Algorithms for u2-hop Labeling	45
5.1	Deletion of a node	45
5.2	Deletion of an edge	48
5.3	Insertion of an edge	49
5.4	Insertion of a subgraph	52
6	Updatable 2-hop (u2-hop) Labeling Construction	55
6.1	Family of Updatable 2-hop (u2-hop) Labeling	55
6.1.1	2-hop Cover with Cut Vertex	55
6.1.2	2-hop Cover with Minimum Bisection	61
6.1.3	2-hop Cover with Node Deletion	64
6.1.4	Discussions	66
6.2	Hybrid Updatable 2-hop Cover	67
7	Experimental Results	69
7.1	2-hop labeling construction	70
7.2	Deletion performance	72
7.3	Experiments with real world graphs	73
7.4	Insertion performance	74
7.5	Supplementary experiments	76
7.6	Summary	77
8	Conclusions and Future Work	78
8.1	Summary of work	78
8.2	Comparison with related works	78
8.3	Contribution of this work	79
8.4	Limitation and future works	80
	Bibliography	83
A	Tarjan algorithm for collapsing strongly connected component	90

B	Approximation algorithm for finding largest biclique	93
C	Approximation algorithm for finding minimum bisection	96
D	List of Publication	99

List of Tables

3.1	The interval labelings of G and $G_{\uparrow}$	28
3.2	Summary of the graph indexing techniques	36
7.1	Real graphs used in the experiment	73

List of Figures

1.1	Example of the Semantic Web	2
1.2	Example of a RDF graph	2
1.3	Example of RDF syntax to represent a RDF graph	3
1.4	Example of SPARQL query language	3
1.5	Gene Ontology for cellular component	4
2.1	An example of XML document	10
2.2	An example of directed graph representation of an XML document	11
2.3	a) A directed graph and b) its transitive closure	12
2.4	An example of small graph $G_0(V_0, E_0)$ and two possible 2-hop covers	14
2.5	A greedy heuristic algorithm for 2-hop construction	16
3.1	Illustration of interval labeling scheme for tree	19
3.2	Illustration of prefix labeling scheme for tree	20
3.3	Illustration of prime labeling scheme for tree	21
3.4	Illustration of prime labeling scheme for directed acyclic graph . .	22
3.5	Illustration of interval labeling scheme for directed acyclic graph .	23
3.6	Illustration of B+-tree structure	24
3.7	An enhanced algorithm for labeled reachability query	25
3.8	A directed acyclic graph $G (G_1)$ and its $G_{\uparrow}$	27
3.9	Geometrical approach for 2-hop labeling construction	28
3.10	A hierarchical construction algorithm of 2-hop cover	29
3.11	A deletion algorithm for 2-hop	31
3.12	A procedure for determining a subgraph whose reachability information is affected by a given update	33

3.13	An example for the 2-hop labels update	34
4.1	Illustration of deletion of x	39
4.2	Procedure <code>check_All_Lin</code> and <code>check_Lin</code>	40
5.1	Deletion of a node in $u2$ -hop labeling	46
5.2	Deletion of an edge in $u2$ -hop labeling	48
5.3	Insertion of a new edge in $u2$ -hop labeling	50
5.4	Illustration of Case 3a: insertion of new edge (x, y) . y will be added to $L_{in}(d_2)$ and $L_{out}(a_2)$, but not to $L_{in}(d_1)$ nor $L_{out}(a_1)$. . .	51
5.5	Insertion of a new subgraph in $u2$ -hop labeling	52
6.1	Bipartite graph G_x for each node $x \in G_0$	56
6.2	(a) The input graph for MEBP (b) An instance of MSEBP generated from v_2 and v_6 (c) An instance of MSEBP generated from v_1 and v_4 (d) An instance of MSEBP generated from v_2 and v_5 . . .	58
6.3	A greedy algorithm for $u2$ -hop-A construction	60
6.4	Updated weights of bipartite graph G_x , at the end of the first iteration, for each node $x \in G_0$. Dotted edges and no-fill nodes are with weight 0. Solid edges and filled nodes are with weight 1. . .	61
6.5	A greedy algorithm for $u2$ -hop-B construction	63
6.6	A greedy algorithm for $u2$ -hop-C construction	65
6.7	$u2$ -hop-C of graph G_0	66
7.1	(a) Index size by varying the graph size; (b) Construction time by varying the graph size;	70
7.2	(a) Synthetic graphs used in the update experiment; (b) Construction time of synthetic graphs	71
7.3	(a) Deletion performance on synthetic graphs; (b) Change in index sizes after deletions (synthetic graphs)	72
7.4	(a) Real graphs used in the update experiment; (b) Deletion experiment on real graphs	74
7.5	Change in index sizes after deletions (real graphs)	75
7.6	Insertion performance on synthetic graphs	75

7.7 (a) Determining the threshold for H-A-SC; (b) Query performance
of the 2-hop of synthetic graphs 76

A.1 Tarjan algorithm for collapsing strongly connected component . . 92

B.1 Sub-procedure SubtractPair 93

B.2 An approximation algorithm for MSEBP 94

C.1 A heuristic to obtain minimum bisection 97

Summary

Recent interests on XML, Semantic Web, and Web ontology, among other topics, have sparked a renewed interest on graph-structured databases. A fundamental query on graphs is the reachability test of nodes. This thesis includes a survey on various indexes to optimize reachability tests. The focus of this thesis is *2-hop* labeling indexing method which was recently proposed to index large collections of XML and/or graphs for efficient reachability tests. In particular, we note that there has been few works on *2-hop* labeling in dynamic setting where the graph is frequently updated. This is compounded by the fact that data may change over time. In response to these, this thesis studies the incremental maintenance of *2-hop* labeling. We analyze the main reason for the inefficiency of updates of existing *2-hop* labels. Based on our analysis, we propose *node-separation property* that leads to efficient incremental maintenance. Subsequently, we present novel incremental maintenance algorithms for *2-hop* labels. We propose three updatable *2-hop* labelings, hybrids of *2-hop* labeling that satisfies node-separation property. The proposed *2-hop* labeling is derived from graph connectivity, as opposed to SET COVER which is used by *all* previous work. Our experimental evaluation illustrates the space efficiency and update performance of various kinds of *2-hop* labeling. Our results show that the incremental maintenance algorithm can be two orders of magnitude faster than previous methods and the size of our *2-hop* labeling can be comparable to the existing *2-hop* labeling.

Chapter 1

Introduction

1.1 Motivation

Recent interests on XML, Semantic Web, and Web ontology, among other topics, have sparked a renewed interest on graph-structured databases. There has been some works on large XML repositories [Mik], ontology data on the Web [Eur], graph networks [BM06], databases for biology [KCVGC⁺05, ABB⁺00] and classical graph databases with recursive query language support [Sus97, BCF⁺02, BCF04, CKS⁺00, FKS⁺02].

A fundamental query on graphs is the reachability test. Specifically, given two nodes u and v of a graph, the test returns true if and only if v is reachable from u . This query evidently cannot be implemented by first order languages, e.g., SQL. For all the reasons that reachability tests are important in classical graph databases, it is also useful to XML. For instance, [STW05, WLH04, CKM02] have proposed indexing techniques for reachability test on XML documents. In addition, the descendant axis, “//”, in XPath (details are available in Section 2.1) can be considered a special application of reachability tests. The descendant axis in XPATH determines a set of the nodes that are reachable from a set of input nodes (the context nodes). This can be implemented by simply extending the reachability test to support sets of nodes. For example, [CCFV07] used reachability tests to optimize recursive XML view updates.

Another use of reachability tests on graphs is the Semantic Web [W3Cc]. We

Figure 1.1: Example of the Semantic Web

Figure 1.2: Example of a RDF graph

sketch an example of resources on the Web in Figure 1.1. One may want to ask: What resources/services are related/reachable to Resource *A*? This can be easily implemented by reachability tests.

Information on resources on the Web can be naturally represented as a graph with Resource Description Framework (RDF) [W3Cb]. Figure 1.2 shows an example of a RDF graph of a set of vCard description of people. The corresponding RDF syntax to represent the graph is given in Figure 1.3. A successor language, namely Web Ontology Language (OWL) [W3Ca], is proposed to represent information in a more machine-readable way compared to that represented by RDF.

In addition, query languages for the Semantic Web, *e.g.*, SPARQL[W3Cd] a W3C recommendation for the Semantic Web, involve subgraph matching where the reachability test is intrinsically a part of the matching. For instance, Figure 1.4 shows an example of SPARQL query on the RDF graph presented in Figure 1.2. The query retrieves the set of contacts with family name equals to "Smith".

In the area of biology, reachability test can be useful to query Gene Ontol-

```
<?xml version='1.0' encoding='UTF-8'?>
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vCard='http://www.w3.org/2001/vcard-rdf/3.0#'
  >
  <rdf:Description rdf:about="http://somewhere/JohnSmith/">
    <vCard:FN>John Smith</vCard:FN>
    <vCard:N rdf:parseType="Resource">
      <vCard:Family>Smith</vCard:Family>
      <vCard:Given>John</vCard:Given>
    </vCard:N>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/MattJones/">
    <vCard:FN>Matt Jones</vCard:FN>
    <vCard:N
      vCard:Family="Jones"
      vCard:Given="Matthew"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 1.3: Example of RDF syntax to represent a RDF graph

```
SELECT ?givenName
WHERE
{
  ?y <http://www.w3.org/2001/vcard-rdf/3.0#Family> "Smith" .
  ?y <http://www.w3.org/2001/vcard-rdf/3.0#Given> ?givenName .
}
```

Figure 1.4: Example of SPARQL query language

ogy (GO) data. Gene Ontology is an information about gene and gene products in any organism. Each GO term consists of a unique identifier, a term name and classification. Each term is classified into one of the three ontologies: molecular function, cellular component or biological process. GO cellular component describes the locations where a gene product may act. For instance, Figure 1.5 shows a Directed acyclic graph (DAG) of the cichlid specific Gene ontology (GO) slim for cellular component obtained from [SRS⁺08]. Each vertex represents a GO term and the circle size represents relative number of genes annotated to the GO term. The edges represents the ontology of cellular component. Reachability query can be used to answer the following question: what are the GO terms related to "nucleus"?

Various techniques have been proposed to implement reachability tests efficiently. On the one hand, a reachability test on a graph can be evaluated using graph traversal method. However, in the worst case, the entire graph might be traversed, hence this method does not scale. On the other hand, one may precom-

Figure 1.5: Gene Ontology for cellular component

pute and materialize the transitive closure¹ of the graph. Then, the reachability test becomes a simple selection on the transitive closure. However, the size of the transitive closure may be large, $O(|G|^2)$ in the worst case. Some previous work on indexes for reachability tests has focused on optimizing query performance and the size of the transitive closure, *e.g.*, [ABJ89].

Recently, a number of indexes for reachability tests have been proposed for optimizing the query performance and/or index size on trees (*e.g.*, [ZND⁺01]), DAGs (*e.g.*, [WZLL06]) or arbitrary graphs (*e.g.*, [STW05]). In practice, data can be cyclic, *e.g.*, Web data. Thus, we focus on methods that support arbitrary graphs.

This thesis studies a popular indexing technique for reachability tests on arbitrary graphs called *2-hop labeling*. In short, *2-hop labeling* of a graph can be considered as a compact representation of the transitive closure of the graph. *2-hop labeling* was originally proposed by [CHKZ03] and later studied in [STW04, STW05, CYL⁺06], among others. In *2-hop labeling*, each node v is associated with two lists, L_{in} and L_{out} which represents a set of nodes that can reach v and a

¹transitive closure of a graph $G(V, E)$ is a set of ordered pair of vertices (u, v) such that there is a path from u to v in G

set of nodes that are reachable from v , respectively. The transitive closure is implicitly encoded by the following rule: (a,b) is in the transitive closure if and only if $L_{out}(a) \cap L_{in}(b)$ is not empty. The *2-hop* labeling construction algorithm carefully chooses, for each v , the nodes belonging to $Lin(v)$ and $Lout(v)$ such that the total size of the lists is minimized and the lists correctly encode the transitive closure of the graph.

When data evolves, there is a need for efficient maintenance of *2-hop* labeling. This thesis studies the *incremental maintenance* of such labeling of graphs, which have received little attention. For ease of presentation, we may use *2-hop* labeling and *2-hop* interchangeably.

Previous works on *2-hop* labeling has mainly focused on time-efficient index construction and optimization of the index size. Determining the *2-hop* labeling with the minimum size is an NP-hard optimization problem [CHKZ03]. To minimize the index size, *all* previous works used SET COVER as a heuristics for computing a minimal *2-hop* labeling of an input graph [CHKZ03, STW04, STW05, CYL⁺06]. Unfortunately, it is also known that the heuristic construction of *2-hop* labeling is computationally intensive. For example, [STW05] reported that the original algorithm [CHKZ03] spends almost two days to construct the *2-hop* labels for a subset of the DBLP XML document – a bibliography repository for Computer Science publications. Subsequent works on *2-hop* labeling mainly focused on efficient *2-hop* construction. For instance, a divide-and-conquer approach [STW05] and a geometric approach [CYL⁺06] have been proposed to improve the *2-hop* construction performance with some tradeoff in the index size.

Since the construction of *2-hop* labeling is costly, it is not feasible to rebuild the labels in response to each single update of the graph. What is desirable is an efficient *2-hop* label *incremental maintenance* algorithm. To the best of our knowledge, the only work that studied incremental maintenance of *2-hop* labeling is [STW05] and [BCCY]. [STW05] and [BCCY] determine the elements in the transitive closure that are affected by an update (a deletion or an insertion). Then, the construction algorithm is applied to the affected elements. Since a single deletion (or insertion) of a graph may affect many elements² in the transitive closure, many parts of the index need to be rebuilt and hence inefficient.

²In the worst case, the entire graph is affected.

In this thesis, we propose novel incremental maintenance techniques of *2-hop* labeling that do not invoke the construction algorithm, and hence more time and space efficient compared to existing approaches. Our experimental evaluation shows that the new techniques can be two orders of magnitude faster than the previous methods. In addition, the size of the *2-hop* labeling constructed by the new techniques can be comparable to that of the existing approaches.

1.2 Objectives and Contributions

We propose a novel approach to maintain the *2-hop* label in response of updates of the underlying graph database.

In this thesis, we propose a few heuristic functions, derived from cut vertex, minimum graph bisection and node deletion, that produce *2-hop* labeling that exhibits special properties which simplify its maintenance.

The drawback of such heuristics is that it has a relatively remote relationship with the index size, when compared to the SET COVER³-based heuristics. As a consequence, the size of our *2-hop* labelings are often larger than that of previously proposed heuristics. We derive some properties of our heuristics that facilitate many hybrids *2-hop* labeling of our and previously proposed *2-hop* labeling. We yield a family of updatable *2-hop* labeling, called *u2-hop* labeling, and the hybrids of *2-hop* labeling from these heuristics. The resulting hybrid of *2-hop* is benefited from the advantages of individual *2-hop* labeling.

The objectives of this thesis are the followings:

- To study thoroughly the problem of incremental maintenance of *2-hop* labeling
- To propose novel incremental maintenance algorithms of *2-hop* labeling
- To conduct experimental evaluation to compare the proposed algorithms with the existing ones

The main contributions of this thesis are the followings:

³SET COVER is an optimization problem: given k sets, find a minimum number of these sets such that the chosen sets contain all elements appearing in any set

- A survey on works related to *2-hop* labeling

We present a survey on the various works on *2-hop* labeling including the original algorithm proposed by Cohen[CHKZ03]. The survey is available on Chapter 3.

- A thorough analysis on updates of *2-hop* labeling in response to update in graph database

We perform a case analysis of the affected elements of the deletion of a node and determine the bottleneck of deletions in *2-hop* labeling. (We skip the analysis on insertions since it is simple.) Since the heuristics for the construction does not take update into considerations, incremental maintenance of *2-hop* labeling can be complicated.

- Node-separation property that leads to efficient *2-hop* maintenance

Based on a case analysis on updates, we define a *node-separation* property of *2-hop* labels. When *2-hop* labeling satisfies the node-separation property, the update of *2-hop* labeling can be simplified, in particular, the inefficient cases in deletion are no longer necessary. (In any case, insertions are simple.)

- Novel incremental maintenance algorithm of *2-hop* labeling

We propose a novel edge/node deletion algorithm to maintain *2-hop* labelings that satisfy node-separation property. Our experiment study shows that our deletion algorithm can be two orders of magnitude faster than previous methods. In addition, the algorithm is extended to work on arbitrary *2-hop* labels with some increase in cost. We also present an insertion algorithm that works on our updatable *2-hop* labels. Both deletion and insertion algorithms do not invoke *2-hop* construction algorithm;

- Heuristic algorithms to construct *2-hop* labeling that satisfies node-separation property

We present three heuristic algorithms derived from graph connectivity for *2-hop* labeling construction and analyze their complexities. The size of our *2-hop* labels, however, is often larger than the previously proposed *2-hop* labels [STW05, CYL⁺06];

- Hybrids of *2-hop* labels

We present methods to construct *2-hop* labeling with a hybrid characteristic of the three heuristics proposed. In addition, we propose a hybrid between *2-hop* with and without node-separation property.

- Extensive experimental study

We conducted extensive experiments on updates of various versions of our *2-hop* labeling to verify the effectiveness of our heuristics and illustrate their performance characteristics.

1.3 Thesis Outline

The structure of the thesis is as follows. Chapter 2 briefly reviews *2-hop* labeling and other preliminaries of this work. Chapter 3 presents a survey of works related to *2-hop* labels. Chapter 4 analyzes updates of *2-hop* labeling and discusses properties which simplifies the maintenance of *2-hop* labeling. The incremental maintenance algorithms of the updatable *2-hop* labeling are presented in Chapter 5. In Chapter 6, we present the definition and the construction algorithms of three updatable *2-hop* labeling based on graph connectivity and the hybrid of *2-hop* labeling. In Chapter 7, we present an experimental evaluation of the proposed updatable *2-hop* labeling to illustrate the characteristics of the *2-hop* labeling. We conclude this work in Chapter 8.

Chapter 2

Background

2.1 Background on XML

While, this work is not confined to XML, but some of the preliminary works are designed to optimize query on XML data. Hence, in this section, we briefly describe XML and graph terminologies which are used throughout the thesis.

XML is a semi-structured, hierarchical data format. An XML document consists of nested elements structure starting from a root element. Each element can contain data (*e.g.*, plain text or binary data) or arbitrary number of elements. An example of XML document is shown in Figure 2.1. Element 'bib' is the root element of the XML document. 'bib' element has three child elements: 'article', 'article' and 'book'. The first 'article' element has three child elements: 'title', 'author' and 'journal'. The first occurrence of 'title' element contains a plain text data: 'Depth-First and Linear Graph Algorithms'.

XML attributes are omitted in our discussion for simplicity. However, our method can be easily extended to support XML attributes.

XML document can be represented as a tree. The solid lines in Figure 2.2 corresponds to tree representation of the XML document in Figure 2.1. An element of an XML document is represented as a node and the parent-child link is represented as an edge. Note that in XML document, the order of the elements matters, hence the order of the edges in the tree also matters. The plain text data is not represented in the XML tree.

```

<bib>
  <article ID="A001">
    <title>Depth-First and Linear Graph Algorithms</title>
    <author>Robert Endre Tarjan</author>
    <journal>SIAM Journal of Computing</journal>
  </article>
  <book ID="B001">
    <title>Foundation of Databases</title>
    <author>Abiteboul</author>
    <year>2004</year>
    <cite idref="A001"/>
    <cite idref="A002"/>
  </book>
  <article ID="A002">
    <title>Hybrid transitive closure algorithms</title>
    <author>Rakesh Agrawal</author>
    <author>H. V. Jagadish</author>
    <journal>VLDB</journal>
  </article>
</bib>

```

Figure 2.1: An example of XML document

In addition, XML allows us to arbitrarily insert a link between two elements. This is achieved by ID and IDREF construct. The dashed lines in Figure 2.2 are examples of arbitrary links. In this work, we don't distinguish between the two type of links. Hence, in general, XML document can be modeled as a directed graph. This model is less restrictive compared to the traditional tree model and can be used in variety of applications dealing with graphs, where the traditional tree model does not naturally fit. Many real-world data can be naturally represented as graph, instead of tree. The following are some examples: DBLP citation graph[Ley], proximity DBLP data [oMA], RDF graph[W3Cb], gene ontology data[SRS⁺08] as well as XXL [STW03, TW02]. In addition, such model is also used in [SWK⁺02, STW04, STW05].

Consider Figure 2.1. In graph model, we can formulate the following queries naturally: find the manuscripts that are directly cited by a given manuscript, find the manuscripts that are directly or indirectly cited by a given manuscript. The first query can be translated as finding the children of a given node, whereas the second query can be translated as finding the descendants of a given node. If tree model

Figure 2.2: An example of directed graph representation of an XML document

is used instead, such query is difficult (or much more complex) to formulate.

Since this work focuses on reachability query, in subsequent discussions, we omit the label of the nodes/edges as it is not relevant to our discussion and our method can be easily modified to support such labelings.

Application of reachability test in XML. XPath is the standard language to specify a subset of XML documents. An XPath expression is executed from a context node and returns a set of nodes that satisfy the expression. An XPath expression consists of a set of location steps. Each location step consists of an axis, a node-label and possibly a predicate (filter). Examples of commonly used XPath axis are '/' and '/' which traverse the child and descendants of the context node, respectively. For example, '/bib//book/title' retrieves nodes labeled 'title' which is a child of a node labeled 'book' which is a descendant of a node labeled 'bib' which is a child of the current context node. In Figure 2.2, the XPath query returns the middle 'title' node.

In order to fit into graph representation of XML document, the notion of children and descendants of a node is extended to also include the links between elements (e.g. IDREF). Hence, the '/' axis would treat a link as a normal tree edge that is to be traversed. This definition is also used in [TW02, STW05, STW04]. The '/' axis is extended similarly. Due to this extension, XPath query can not

Figure 2.3: a) A directed graph and b) its transitive closure

be efficiently evaluated using the previous approaches (e.g. tree interval labeling scheme[ABJ89]). A more sophisticated approach is required.

The descendant axis of XPath can be evaluated efficiently by translating it into a sequence of reachability tests on the XML graph. Since, XPath is an integral part of XQuery, a query language for XML which allows more complex queries rather than node selection, reachability query can also be used as a primitive to optimize XQuery evaluation.

Notation and terminologies. In this work, we represent XML as a directed graph. A directed graph is represented as an ordered pair $G(V, E)$, where V is a set of nodes and E is a set of ordered pairs (u, v) of edges, where $u \in V$ and $v \in V$. In this thesis, we use the term node and vertex interchangeable.

A path in $G(V, E)$ is a sequence of nodes $v_1, v_2, \dots, v_n$ where there exists edges $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ in E . A node u *can reach* v , if and only if there is a path that starts from u and ends at v in $G(V, E)$. A cycle in a graph is a path that starts and ends at the same node. A directed acyclic graph (DAG) is a directed graph with no cycle.

Ancestors of a node v , denoted as $anc(v)$, is a set of nodes that can reach v . Descendants of a node v , denoted as $desc(v)$, is a set of nodes that is reachable from v .

Transitive closure of a graph G , $T(G)$, is a set of ordered elements: $(u, v) \in T(G)$ if and only if u can reach v in G . Figure 2.3-a depicts an example of a directed graph. Its transitive closure is shown in Figure 2.3-b. In Figure 2.3-b, each element of the transitive closure is represented as an edge. That is, there is an edge from u to v , if and only if u can reach v in G .

A strongly connected component of a directed graph $G(V, E)$ is a set of nodes W in V where for each pair of nodes a and b in W , a can reach b and b can reach a .

An induced graph of a set of nodes V' in a graph $G(V, E)$ is a subgraph of G , denoted as $G'(V', E')$, where $E' = \{ (u, v) \mid u \in V' \text{ and } v \in V' \}$.

2.2 Definition of 2-hop labels

In this section, we provide some background on *2-hop* labeling and show how reachability test is efficiently supported by such labeling.

Since the reachability information of the nodes in a strongly connected component in a graph is trivial, we assume that each strongly connected component in the graph is reduced to a node. This can be efficiently done by Tarjan algorithm[Tar72] in one scan of the graph (details can be seen in Appendix A). The reduced graph is a directed acyclic graph (DAG). Our subsequent discussions always assume the reduced graph.

We denote a directed graph as $G(V, E)$. Each node v in V is associated with a *label* L , which consists of two lists of nodes $L_{in}(v)$ and $L_{out}(v)$. The two lists are called *2-hop labels*. The nodes that are stored in $L_{in}(v)$ are some nodes that can reach v . Similarly, the nodes that are stored in $L_{out}(v)$ are some nodes that are reachable from v . We often refer to the nodes in either $L_{in}(v)$ or $L_{out}(v)$ as *center nodes*.

The *2-hop* labels must represent the reachability information of G . Specifically, the *2-hop* labels must cover all elements in the transitive closure $T(G)$ of G , and must not cover any element outside the transitive closure. An element (u, v) is covered by the *2-hop* labels if and only if there is at least one common node in $L_{out}(u)$ and $L_{in}(v)$. We call the *2-hop* labels that cover exactly all elements in $T(G)$ *2-hop cover* $H(G)$ of G . Obviously, there are many possible *2-hop* covers

Figure 2.4: An example of small graph $G_0(V_0, E_0)$ and two possible 2-hop covers of a graph.

Given two nodes u and v , v is reachable from u , denoted as $u \rightsquigarrow v$, if and only if $L_{in}(v) \cap L_{out}(u) \neq \phi$. This can be easily derived from the definition of 2-hop cover.

We present the formal definition of 2-hop cover as follows.

Definition 2.1: 2-hop cover of a directed graph $G(V, E)$ assigns to each vertex $v \in V$, a label $L(v) = (L_{out}(v), L_{in}(v))$ where $L_{out}(u) \subseteq V$ and $L_{in}(v) \subseteq V$, such that for every $u, v \in V$, $L_{out}(u) \cap L_{in}(v) \neq \phi$ if and only if there is a path from u to v in G . ■

To ease the presentation, the reflexive closure is implicitly encoded by $L_{in}(v)$ and $L_{out}(v)$. Specifically, v is implicitly included in $L_{in}(v)$ and $L_{out}(v)$.

We may omit v from $L_{in}(v)$ and $L_{out}(v)$, and G from $T(G)$ and $H(G)$ when they are clear from the context.

Next, we illustrate how 2-hop labeling works with an example. Consider the graph G_0 in Figure 2.4 and the nodes v_1 and v_9 . We show one possible 2-hop cover of v_1 and v_9 in Figure 2.4-a: $L_{out}(v_1) = \{v_2, v_3, v_4, v_6\}$ and $L_{in}(v_9) = \{v_6, v_8\}$. The labels can be interpreted as follows: v_2, v_3, v_4 and v_6 are reachable from v_1 ; and v_9 is reachable from v_6 and v_8 . To answer query whether v_1 can reach v_9 , we compute the following equation: $L_{out}(v_1) \cap L_{in}(v_9) = \{v_6\}$. This means that there is a path from v_1 to v_9 via the center node v_6 . To answer query whether v_2

can reach v_7 , we compute $L_{out}(v_2) \cap L_{in}(v_7) = \text{empty}$. Hence, v_2 could not reach v_7 . Figure 2.4-b shows another possible 2-hop cover of graph G_0 .

The size of a 2-hop cover is defined as $\sum_{v \in V} |L_{in}(v)| + |L_{out}(v)|$, e.g., the total number of elements the table contains. In the example above, the size of the 2-hop labels in Figure 2.4-a are 13, whereas the size of transitive closure is 24.

We have shown that 2-hop labels can succinctly represent the transitive closure with a slight increase in query time.

2.3 Construction of 2-hop cover

In this section, we would discuss how 2-hop cover of a graph can be constructed.

As we mentioned in previous section, there could be many possible 2-hop cover of a given graph. One naive method to construct 2-hop cover is by a trivial reduction from transitive closure of the graph, as follow: for each element (u, v) in $T(G)$, add v into $L_{out}(u)$. While, this would produce a 2-hop cover, the size of the 2-hop cover is the same as the size of the transitive closure.

Constructing 2-hop cover with minimum size is not trivial. In the original proposal of 2-hop labeling, Cohen *et al.* [CHKZ03] proved that finding the 2-hop cover with the minimum size is an NP-hard problem. Hence, various heuristics have been proposed to determine space-efficient 2-hop covers iteratively. In particular, we briefly describe the original algorithm proposed by [CHKZ03] in Figure 2.5.

The algorithm proceeds in several steps:

- Compute the transitive closure of the graph and assign it to T' , which stores the uncovered elements of T . (Line 01-02)
- iteratively choose some remaining elements from T' to be processed (Line 05-08)
- process the chosen elements (Line 09-10)
- remove them from T' (Line 11-12). The algorithm terminates when T' is empty.

Procedure 2-hop-construction**Input:** a directed graph $G: (V, E)$ **Output:** 2-hop cover of $G: (L_{in}, L_{out})$

```

01 for each  $v \in V$ 
    construct the bipartite graph  $G_v$ 
02  $T' := TC(G_0)$ 
03  $B_{max}(V_{max_1}, V_{max_2}, E_{B_{max}}) :=$  empty bipartite graph;  $max := \text{null}$ 
04 while  $T'$  is not empty
05   for each  $v \in V$ 
06     find densest subgraph  $B_v(V_1, V_2, E_{B_v})$  of  $G_v$ 
07     if  $\text{density}(B_v) > \text{density}(B_{max})$ 
08        $B_{max} := B_v$ ;  $max := v$ 
09    $L_{out}(a) := \{max\} \cup L_{out}(a)$ ,
       where  $a \in V_{max_1}$ 
10    $L_{in}(d) := \{max\} \cup L_{in}(d)$ ,
       where  $d \in V_{max_2}$ 
11   update  $T'$  according to  $B_{max}$ 
12   update  $G_v$  for  $v \in V$  according to  $T'$ 

```

Figure 2.5: A greedy heuristic algorithm for 2-hop construction

We describe the details of choosing some elements from T' to be processed as follows. An undirected bipartite graph $G_v(A_v, D_v, E_v)$ is constructed for each node v (Line 03). A_v is the set of ancestors of v . Similarly, D_v is the set of descendants of v . Edge $(u, w) \in E_v$ if and only if (u, w) is in T' and w is reachable from u via v . Then, the SET COVER heuristics finds an induced subgraph $B_v(V_1, V_2, E_{B_v})$ of G_v with the ratio $r = \frac{|E_{B_v} \cap T'|}{|V_1 \cup V_2|}$ maximized among all possible subgraphs of G_v (Line 06). This is exactly the problem of finding the *densest* subgraph of G_v . At each iteration of the algorithm, a node max having the largest r is picked as a center node and the densest subgraph B_{max} is chosen as the elements to be processed (Line 05-08). Specifically, the chosen elements of T' to be processed, denoted as C , consist of union of the set (a, max) , where $a \in V_{max_1}$ and the set (max, d) , where $d \in V_{max_2}$.

Then, C are processed by adding max to L_{out} of nodes in V_{max_1} and L_{in} of nodes in V_{max_2} . This ensures that all elements of C have been covered in the 2-hop

labels constructed so far.

Finally, we remove all elements of C from T' (Line 11) and remove the corresponding edges from G_v , $v \in V$ (Line 12).

Example 2.1: Consider graph G_0 in Figure 2.3-a. Firstly, the transitive closure of the graph is computed (Line 02). This is shown in Figure 2.3-b. The next step is to find densest subgraph of G_0 (Line 06). In this iteration, the densest subgraph of v_6 , a complete bipartite graph $B_6(\{v_1, v_3, v_4, v_6\}, \{v_6, v_7, v_8, v_9\})$ with density of $r = \frac{4 \times 4}{4+4} = 2$ is chosen. Next, v_6 is added to L_{out} of $\{v_1, v_3, v_4\}$ as well as L_{in} of $\{v_7, v_8, v_9\}$ (Line 09-10). Next, the edges of B_6 are removed from T' (Line 11). In the next iteration, a complete bipartite graph $B_4(\{v_1, v_3, v_4\}, \{v_4, v_5\})$ is chosen. v_4 is then added to $L_{out}(v_1)$ and $L_{out}(v_3)$ as well as $L_{in}(v_5)$. On the subsequent iteration, $B_2(\{v_1, v_2\}, \{v_2, v_5\})$ is chosen. v_2 is added to $L_{out}(v_1)$ and $L_{in}(v_5)$. In the final iteration, v_3 is added to $L_{out}(v_1)$. The resulting 2-hop cover is presented in Figure 2.4-a. ■

We remark that the problem of finding the densest subgraph of G_v is solvable in polynomial time using network flow technique (see [GGT89],[Law01] and [GT88]). However, the time complexity is high. Hence, the original proposal from Cohen is to use an approximation algorithm based on [KP94] for this problem which takes linear time.

The heuristic algorithm presented aims to minimize the index size. At each iteration, the algorithm heuristically, choose the densest elements from T to be processed. At each iteration, the ratio of number of edges over number of nodes is maximized. The number of edges represents the number of elements in T that are processed (the gain) and the number of nodes represents the increase in index size (the cost) in processing the elements in T . Hence, choosing the densest elements contribute to smaller index size.

Existing approaches to update the 2-hop cover in response to an update in the underlying graph are discussed in Chapter 3.

Chapter 3

Literature Review

3.1 Indexes for reachability query

There has been a large body of works on graph indexing for reachability queries. In this section, we list a few (non-exhaustive) examples and give some brief description of the main idea of each work.

We start our discussion on tree-structured data. [ABJ89] assigns lower number and upper number (denoted as interval) to each node in a tree. The upper number is the post order number of the node (*e.g.*, its position in post order traversal of the tree). The lower number is the lowest post order number of the node's descendants. The interval assigned to a node encodes the set of descendants of the node. A node u can reach node v if and only if the post order number of v lies within the interval of u . Figure 3.1 shows an example of the interval approach. The interval associated with v_2 is $[1, 6]$. The post order number of v_2 is 6. The lowest post order number of descendants of v_2 is 1 (*e.g.*, v_5). Hence, v_2 can reach all nodes with post order number 1 to 6, ($v_2, v_5, v_6, v_7, v_{11}, v_{12}$). For instance, it can be concluded that v_2 can reach v_6 as the post order of v_6 is 4 which lies within 1 to 6.

The interval labeling approach requires index size of $2 \times N$ numbers (linear to the number of nodes). Query can be answered in constant time as it requires only a simple range comparison.

The interval approach requires storage space of $2 \times \log(N)$ bits per node. A

Figure 3.1: Illustration of interval labeling scheme for tree

prefix based approach which aims to further minimize the index size was discussed in [KMS02]. [KMS02] labels each node with a binary string such that u is ancestor of v if and only if the label of u is a prefix of the label of v . This is done by assigning a binary string for each edge such that the collection of strings assigned with the outgoing edges of a node is *prefix-free*. The binary string associated to a node is a concatenation of the binary strings of each edge in the path from the root to the node. An illustration of this approach is shown in Figure 3.2. As required, it is shown that the label of each edge which starts from the same node must be prefix-free. For instance, the labels of outgoing edges of v_1 are '1','01','00'. Label of v_7 is '101' which is a concatenation of '1' and '01', the labels of (v_1, v_2) and (v_2, v_7) respectively. v_2 is ancestor of v_{12} as the label of v_2 , '1' is a prefix of the label of v_{12} , '111'.

Finding the minimum total label size depends on the physical representation of the labels. There are two ways to physically store the label of each node: fixed-length and variable-length representations. The earlier reserves a fixed number of bits for each label where the latter allows each node to have different label length (in bits). [KMS02] proposed an optimal algorithm to find minimum label size, both for fixed-length and variable-length representation which utilizes Huffman encoding algorithm[Huf52]. Finally, the experimental results show that in practice the total label size can be reduced by more than 20%.

Figure 3.2: Illustration of prefix labeling scheme for tree

There has been some extension to the interval labeling scheme as well as prefix labeling scheme in dynamic context, where the tree can be updated [CKM02]. Deletion of nodes is easy as the labels of the nodes do not need to change. [CKM02] addresses two scenarios for insertions: with and without clues. Insertion without clues refers to the case where the insertion operation only specifies the new node to be inserted and its parent node, with no additional information. Insertion with clues refers to the case that an additional clue is given. The paper considers two types of clue: 1) the subtree clue which is an estimate of number of future descendants of the new node; 2) the sibling clue which consists of the subtree clue and an estimate of the number of descendants of the future siblings of the new node. The main idea of the paper is not to use all the available prefixes (*e.g.*, reserve some prefixes to be used for subsequent insertions). Hence, for each node, it is always possible to assign a new prefix to a candidate outgoing edge of the node.

A theoretical work [AKM01] investigated the minimum possible size of such labeling. The main result of the paper is that it proposes a labeling scheme with label size of $1.5 \log n + O(\log \log n)$ bits. The details are somewhat involved and not relevant for our discussion. Interested readers can refer to [AKM01].

A different approach was proposed in [WLH04] which utilizes prime numbers to encode reachability information of a tree. Each non-leaf node is assigned a distinct prime number. The label of each node is a product of two numbers, the

Figure 3.3: Illustration of prime labeling scheme for tree

prime number assigned to the node and the label of its parent. The root has a label of 1. Hence, a node is reachable from another node *iff* its label is divisible by the label of the other node. An example of this scheme is presented in Figure 3.3. In this example, a sequence of prime number starting from 2 is assigned to v_2 - v_{12} respectively. The label of v_7 is 26 which is obtained from a product of 13 (the prime number assigned to the node) and 2 (the label of its parent node, v_2). From the labels, we can deduce that node v_2 is an ancestor of v_{12} . This is due to 638 (the label of v_{12}) is divisible by 2 (the label of v_2). Note that the size of the label of a node (in bits) increases drastically as the depth of the node increases. That is, the label of a node is a multiple of the label of its parent. Consider a tree consisting of a straight line of n nodes. The label of the n -th nodes is as large as $n!$. Hence, while this method is simple and efficient, it only works well on shallow trees where the depth of the tree is small.

All the mentioned works consider tree-structured data instead of the more general graph-structured data.

[WZLL06] extends previous work on [WLH04] to support DAGs. As in [WLH04],

Figure 3.4: Illustration of prime labeling scheme for directed acyclic graph

a unique prime number is assigned to each node v , denoted as $p[v]$. The label of a node v consists of three numbers $(p[v], c_a[v], c_p[v])$, where $c_a[v] = p[v] \times \prod_{v' \in \text{parents}(v)} c_a[v']$ and $c_p[v] = \prod_{v' \in \text{parents}(v)} p[v']$.

We denote $c_a[v]$ as the ancestor number and $c_p[v]$ as the parent number of node v . By using the ancestor number, we can efficiently determine whether one node is an ancestor of another node. Specifically, u is an ancestor of v , if and only if $c_a[u] \mid c_a[v]$. This is easy to see from the definition of c_a .

In addition, using the parent number of a node, we can derive the set of parents of the node. Note that since $c_p[v]$ is a product of prime numbers, it has unique factorization. Each factor refers to one parent of node v . Hence, the set of parents of a node v are nodes whose prime number is a factor of $c_p[v]$.

Figure 3.4 shows an illustration of prime labeling scheme in DAG. Consider v_7 and its label $(11, 1133605968, 1311)$. The unique prime number $p[7]$ associated with v_7 is 11. The ancestor number $c_a[7]$ is 1133605968. This is obtained from a product of $p[7]$ and the ancestor numbers of the parents of v_7 : $c_a[v_2]=204$, $c_a[v_5]=646$ and $c_a[v_6]=782$; Hence, $c_a[7] = 204 \times 646 \times 782 = 1133605968$. The parent number $c_p[7]$ is 1311 which is a product of the prime numbers of its parents. The values of $c_p[v_2]$, $c_p[v_5]$ and $c_p[v_6]$ are 3, 19 and 23 respectively. Hence, $c_p[7] = 3 \times 19 \times 23 = 1311$. v_3 is an ancestor of v_9 because $c_a[9]=14736877584$ is divisible by $c_a[3] = 34$. The set of parents of v_8 can be determine by looking at the set of

Figure 3.5: Illustration of interval labeling scheme for directed acyclic graph

factors of $c_p[8] = 55 = 11 \times 5$, which refers to v_4 and v_7 .

Many techniques to index DAG are based on an extension of interval labeling approach. Since single interval can only capture reachability information of a tree, a set of intervals are assigned to each node in the graph.

For instance, [ABJ89] proposed an extension to interval based method to compactly represent transitive closure of a directed acyclic graph. The main idea is the following:

1. Find a spanning tree of the graph
2. With respect to the spanning tree, assign interval label (u, v) to each node, where v is the post order number of the node and u is the lowest post order number of the node's descendants. This is similar to the interval labeling scheme for tree.
3. Process the nodes in reverse topological order. When visiting node u , for each non-tree edge (u, v) , append intervals of v to intervals of u (e.g., each node maintains a set of intervals). As an optimization, we remove an interval that is completely inside another interval.

Illustration for this approach is described in Figure 3.5. The solid lines represent the spanning tree whereas the dotted lines represent the non-tree edges. The intervals of v_3 are 1) $[7, 9]$, obtained from the spanning tree; 2) $[2, 4]$, obtained

Figure 3.6: Illustration of B+-tree structure

from intervals of v_6 ; 3) $[10, 11]$, obtained from intervals of v_4 . Note that the interval $[10, 10]$ that v_3 inherits from v_6 are completely inside $[10, 11]$, hence it is removed.

The first step of this algorithm is to find a spanning tree of the graph. Any spanning tree would work. However, not all spanning trees are equally good in producing smallest index size. Hence, [ABJ89] also proposes an optimal algorithm to compute the optimal spanning tree of the graph which would minimize the index size.

Cohen et al[CHKZ03] proposed *2-hop* labeling approach to compactly represent the transitive closure of the graph with a slight increase in query time. This work has been described in Section 2.2.

Given an XML document, [CYT06] proposed a join index using B+-TREE to efficiently answer reachability query involving the node labels, e.g., given node labels A and D , find all pair of nodes (u, v) where u can reach v , the label of u is A and the label of v is D . The proposed approach is based on *2-hop labeling*.

Specifically, it computes $H^+(w)$ and $H^-(w)$ where $H^+(w) = \{u | w \in L_{in}(u)\}$ and $H^-(w) = \{u | w \in L_{out}(u)\}$. We define $H^+(L, w)$ as the set of nodes in $H^+(w)$ whose label is L . $H^-(L, w)$ is defined in similar manner.

$H^+(L, w)$ and $H^-(L, w)$ are stored in a B+-tree to be used as an index to answer the query. An illustration of the index is shown in Figure 3.6. The key of the B+-tree is a pair (node-label, center-node). Given the *2-hop* cover of the XML document, the B+-tree can be efficiently constructed.

Procedure Answer-reachability-query**Input:** the ancestor label A , the descendant label D , the inverted join index B+-tree**Output:** the set of pair of nodes (u,v) where $u \rightsquigarrow v$, $\text{label}(u)=A$ and $\text{label}(v)=D$

```

01  $W$  = Get the set of nodes  $w$  such that  $H^+(A, w)$  and  $H^-(A, w)$  are not empty
02 for each  $w$  in  $W$ 
03    $B^+$  = search B+-tree for  $H^+(A, w)$ 
04    $B^-$  = search B+-tree for  $H^-(A, w)$ 
05   for each pair of  $a \in B^+$ 
06     for each pair of  $d \in B^-$ 
07       add  $(a,d)$  to the output

```

Figure 3.7: An enhanced algorithm for labeled reachability query

By using this index, the Algorithm 3.7 was proposed to process the reachability query. First, the algorithm finds the set of center nodes W between nodes labeled A and nodes labeled D . Next, from the definition of $H^+(A, w)$ and $H^-(A, w)$, each pair of nodes in $H^+(A, w)$ and $H^-(A, w)$ are definitely reachable. Hence, they are added to the solution. The correctness of this algorithm relies on the correctness of the 2-hop labeling which ensures that each path in the graph is covered in the 2-hop label (and hence in $H^+(A, w)$ and $H^-(A, w)$).

Many of the aforementioned works, the indexes are designed to work in static context, where updates of the underlying graph are not considered. Hence, on every update, the index needs to be rebuilt to reflect the change in the underlying graph. This might be computationally expensive.

3.2 Construction of 2-hop labels

2-hop labeling was originally proposed in [CHKZ03] to succinctly represent all distances in a graph. Specifically, 2-hop labeling can be used to compactly represent all shortest paths as well as the distance between each pair of nodes in the graph. This thesis only addresses reachability query, a specific type of distance. The term 2-hop in this thesis refers to 2-hop reachability labeling in [CHKZ03]. Details of [CHKZ03] has been discussed in Section 2.3. Hence, in this section, we mention some of the follow up works of [CHKZ03].

The objectives of works mentioned in this section are to improve the time and memory requirement of the construction process and/or to reduce the index size. In Section 4.1, we illustrate that such constructions generate *2-hop* labels with small sizes with no special property which leads to computationally expensive update.

3.2.1 Works by Ralf Schenkel et al.

[STW04] investigated an improvement of *2-hop* labeling scheme to efficiently index a large collection of XML documents. It proposed a three step approach on index construction as follows

- Partition step
Partition the XML graph into a number of partitions such that the *2-hop* cover computation for each partition can be computed with memory-based data structures.
- Computation step
Compute the transitive closure and *2-hop* cover for each partition and store the *2-hop* cover on disk.
- Merging step
Merge the *2-hop* cover for partitions by processing the inter partition edges.

While this method tends to produce a *2-hop* cover with larger size, but the memory requirement as well as construction time are much smaller compared to [CHKZ03]. Relational implementation of *2-hop* cover representation and query processing are also discussed.

A follow up work from the same author, [STW05], proposed some improvements on the construction algorithm. As before, the graph is partitioned into a number of partitions and the *2-hop* cover of each partition, denoted as H_i , is computed. It introduces a partition skeleton graph, denoted as *PSG*, of the partition *P*. The nodes of the *PSG* are nodes that have inter-partition edges in *P*. If the *PSG* does not fit into memory, it is further partitioned. Next, the *2-hop* cover of the

Figure 3.8: A directed acyclic graph G ($G_↓$) and its $G_↑$

PSG, denoted as H , is computed. The 2-hop cover of the whole graph is a concatenation of the H , H_i and $\hat{H}$, where $\hat{H}$ is an auxiliary cover which is computed as follows.

1. for each source node v and for each u , ancestor of v in the same partition,
 $\hat{H}_{out}(u) = \hat{H}_{out}(u) \cup H_{out}(v)$
2. for each target node v and for each u , descendant of v in the same partition,
 $\hat{H}_{in}(u) = \hat{H}_{in}(u) \cup H_{in}(v)$

3.2.2 Work by Jiefeng Cheng et al. (2006)

Original proposal in [CHKZ03] iteratively finds the densest subgraph, e.g., $\frac{|E \cap T'|}{|V|}$ is maximized. While, this approach returns space-efficient 2-hop cover, the time and memory requirements for computing G_w are prohibitive. One of the results in [CYL⁺06] shows that the division in this heuristics has a minor impact on the size of 2-hop cover. Hence, [CYL⁺06] has proposed a simpler heuristics where $|E \cap T'|$ is maximized, which leads to more efficient 2-hop cover construction.

We illustrate the approach of [CYL⁺06] with an example. Consider a directed graph G . First, each strongly connected component in G is reduced to a node. The reduced graph $G_↓$ is shown in Figure 3.8 (a). Since our discussion always focuses on the reduced graph, we may use G and $G_↓$ interchangeably. 2-hop labels are then built on the reduced graph. First, the interval labeling [ABJ89] is built on G . The interval labeling of a node v consists of (i) a unique post-order number, $poID(v)$, of the optimum tree cover of G [ABJ89] and (ii) a set of intervals $\{[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]\}$, denoted by $I(v)$. Then, u can reach v iff

Table 3.1: The interval labelings of G and $G_{\uparrow}$

v	G		$G_{\uparrow}$	
	$poID(v)$	$I(v)$	$poID_{\uparrow}(v)$	$I_{\uparrow}(v)$
0	9	[1,9]	4	[4,4]
1	1	[1,1],[3,3]	3	[1,5]
3	6	[1,6]	5	[4,5]
4	2	[2,2]	9	[4,5],[9,9]
5	5	[3,5]	6	[4,6]
8	7	[1,1],[3,3],[7,7]	1	[1,1],[4,4]
9	4	[3,4]	7	[4,7]
11	3	[3,3]	8	[1,8]
12	8	[1,1],[3,3],[8,8]	2	[2,2],[4,4]

$poID(v)$ is contained in an interval, $[s_j, e_j]$, where $[s_j, e_j] \in I(u)$. The size of $I(v)$ can be defined as $\sum_{i=1..n}(e_i - s_i + 1)$, which is the number of nodes reachable from v in G , since each number in $I(v)$ corresponds to a post-order number of a node. For example, the interval labeling of Figure 3.8 (a) is shown in the G column of Table 3.1.

Figure 3.9: Geometrical approach for 2-hop labeling construction

Next, another auxiliary graph, $G_{\uparrow}$, is constructed. $G_{\uparrow}$ is obtained from G by reversing the direction of each edge in G (e.g., Figure 3.8 (b)). The interval labeling of a node v in $G_{\uparrow}$, denoted as $I_{\uparrow}(v)$ and $poID_{\uparrow}(v)$ are constructed, shown in the $G_{\uparrow}$ column in Table 3.1.

Note that all reachability information of G has been encoded with the two interval labelings. A 2-dimensional map is used to represent these interval labelings. The x-axis and y-axis of the reachability map are $I(v)$ and $I_{\uparrow}(v)$, respectively. A cell (x, y) is green iff $u \leadsto v$, where $x = poID(v)$ and $y = poID_{\uparrow}(u)$. For example, since $0 \leadsto 1$, $poID(1) = 1$ and $poID_{\uparrow}(0) = 4$, $(1, 4)$ is green. Hence, the green area

Procedure Topdown**Input:** a directed graph $G: (V, E)$, 2-hop cover H of G **Output:** 2-hop cover H of G

```

01 if  $G$  is smaller than a pre-defined threshold  $\tau$ 
02   Cover( $G, H$ )
03   return
04 bisect  $G$  into  $G_A$  and  $G_D$ 
05 let  $E_c$  be the edges between  $G_A$  and  $G_D$ 
06 CrossCover( $E_c, G, H$ )
07  $G_\top = \text{Extract}(G_A, H)$ 
08  $G_\perp = \text{Extract}(G_D, H)$ 
09 Topdown( $G_\top, H$ )
10 Topdown( $G_\perp, H$ )

```

Figure 3.10: A hierarchical construction algorithm of 2-hop cover

in Figure 3.9 (a) represents all reachability information of G . When Node 3 is chosen as a center of its ancestor and descendant nodes, Node 3 is added into L_{out} of its ancestors and L_{in} of its descendants. The reachability information covered by Node 3 is visualized with the shaded blue area – the rectangle defined by $I(3)$ and $I_\top(3)$.

The 2-hop labels are constructed by picking center nodes repeatedly until the green area is fully covered. For example, the new reachability information covered by centers Node 1, 9 and 0 (in sequence) are illustrated with the shaded blue rectangles in the maps in Figure 3.9 (b), (c) and (d), respectively.

3.2.3 Work by Jiefeng Cheng et al. (2008)

The same author in [CYL⁺08] proposed a hierarchical construction of 2-hop labeling. The idea is to recursively bisect a graph and process the crossing edges. The algorithm is presented in Figure 3.10.

The algorithm starts by bisecting the graph into G_A and G_D (Line 04). According to a mathematical formula, the paper shows that to achieve high compression rate, the size of G_A should be similar to that of G_D . Next, it computes the crossing

edges between G_A and G_D (Line 05) and compute *2-hop* cover with respect to the crossing edges (Line 06). Subsequently, $G_{\top}$ and $G_{\perp}$ are obtained by removing the nodes used as center nodes in previous step from G_A and G_D respectively. This is done through procedure `Extract`. Finally, $G_{\top}$ and $G_{\perp}$ are processed recursively (Line 09-10). At any recursive call, if the size of input graph G is less than threshold τ (Line 01), then the *2-hop* cover of the graph is computed directly in `Cover` procedure, *e.g.*, using the approach in [CYL⁺06] (Line 02).

[CYL⁺08] proposed two approach to implement `CrossCover` in Line 06: node-oriented approach and edge-oriented approach. Let V_c be the set of nodes that are incident of some node in E_c . In node-oriented approach, the idea is to find a minimum subset V_w of V_c such that for each edge e_c in E_c , e_c is incident with some node in V_w . While this can be done exactly using network flow technique, the author proposed to use a greedy algorithm by iteratively picking node with maximum number of incident edges. In edge-oriented approach, for each edge in E_c , a virtual node is created. Then, the *2-hop* cover can be built by considering the virtual node separators.

3.3 Maintenance of 2-hop labels

To the best of our knowledge, incremental maintenance of *2-hop* labeling has only been discussed in [STW05] and [BCCY]. We describe their approaches briefly in this section.

3.3.1 Work by Ralf Schenkel et al.

[STW05] proposed the following maintenance algorithm:

1. Insertion of node/edge

Insertion of a new node can be performed by simply adding the new node to the graph. No change to the *2-hop* cover required. Insertion of a new edge (u,v) between existing nodes can be performed as follow. Add v to $L_{out}(a)$ of all ancestors a of u (including u) and to $L_{in}(d)$ of all descendants d of v .

2. Deletion of node/edge

Procedure delete-node**Input:** a directed graph $G: (V, E)$, a set of nodes X to be removed2-hop cover of G : L_{in} and L_{out} **Output:** updated 2-hop cover of G : (L'_{in}, L'_{out})

```

01 Let  $anc(X) := \{a \mid a \in anc(x), x \in X\}$ 
02 Let  $desc(X) := \{d \mid d \in desc(x), x \in X\}$ 
03 if  $X$  separates  $G$ 
04   for each  $a \in anc(X)$ 
05      $L'_{out}(a) = L_{out}(a) - X - desc(X)$ 
06   for each  $d \in desc(X)$ 
07      $L'_{in}(d) = L_{in}(d) - X - anc(X)$ 
08   for each  $v \notin anc(X)$  and  $v \notin desc(X)$ 
09      $L'_{out}(v) = L_{out}(v)$ 
10      $L'_{in}(v) = L_{in}(v)$ 
11 else
12    $C := desc(anc(X))$ 
13    $TC(C) :=$  Transitive closure of  $C$ 
14    $\hat{L}_{out}$  and  $\hat{L}_{in} =$  2-hop cover of  $C$ 
15   for each  $a \in anc(X)$ 
16      $L'_{out}(a) = \hat{L}_{out}(a)$ 
17   for each  $d \in desc(X)$ 
18      $L'_{in}(d) = (L_{in}(d) - anc(X)) \cup \hat{L}_{in}(d)$ 
19   for each  $v \notin anc(X)$  and  $v \notin desc(X)$ 
20      $L'_{out}(v) = L_{out}(v) \cup \hat{L}_{out}(v)$ 
21      $L'_{in}(v) = L_{in}(v) \cup \hat{L}_{in}(v)$ 

```

Figure 3.11: A deletion algorithm for 2-hop

The algorithm to delete a set of nodes is presented in Figure 3.11. The first two lines contain a formal definition of ancestors and descendants of a set of nodes, respectively. Next, the algorithm checks if the set of nodes to be deleted separate the graph. A set of nodes X in G *separates* G if and only if for every $a \in \text{anc}(X)$, and every $d \in \text{desc}(X)$, deletion of X would disconnect a from d . If this condition is true, then the first approach (Line 04-10) is applied.

Specifically, it disconnects $\text{anc}(X)$ from $\text{desc}(X)$ by removing $\text{desc}(X)$ and X from L_{out} of $\text{anc}(X)$ (Line 04-05) and removing $\text{anc}(X)$ and X from L_{in} of $\text{desc}(X)$ (Line 06-07). The other labels remain the same (Line 08-10).

Although the first approach is efficient, but it only applies to a restricted condition which does not occur frequently in practice. If the condition is not satisfied, the second approach (Line 12-21) is applied as follows.

The first step is to compute the nodes whose labels might be affected by the deletion (Line 12). Next, the transitive closure as well as $\hat{L}$, the 2-hop cover, of those nodes are computed (Line 13-14). The updated 2-hop cover of ancestors of X and descendants of X are derived from the $\hat{L}$ (Line 15-18). The other labels remain the same (Line 19-21).

3.3.2 Work by Bramandia et al.

[BCCY] improves the algorithm proposed by [STW05]. The improved algorithm utilizes the geometrical data structure used during the construction process of [CYL⁺06] described in Section 3.2.2.

Given an update δ on G , we want to find L' , the updated 2-hop labels L of G . The approach of [BCCY] consists of three steps:

1. find a small subgraph $\Delta_G(\Delta_V, \Delta_E)$ that is relevant to all modifications of reachability information of G due to δ (all nodes that are not in Δ_G are not affected by δ);
2. compute the 2-hop labels $\Delta_L, (\Delta_{L_{\text{in}}}, \Delta_{L_{\text{out}}})$, of Δ_G ; and

Procedure get_small_relabeling_subgraph**Input:** a DAG G andan update operation δ **Output:** a subgraph of G , Δ_G 01 compute A_δ and D_δ of δ 02 $I_A^\uparrow :=$ union of the intervals $I_\uparrow(v)$ for all $v \in A_\delta$ 03 $I_D :=$ union of the intervals $I(u)$ for all $u \in D_\delta$ 04 $A := \{v \mid poID_\uparrow(v) \text{ in } I_A^\uparrow, v \in G.V\}$ 05 $D := \{u \mid poID(u) \text{ in } I_D, u \in G.V\}$ 06 $I_{A_\Delta} :=$ union of the intervals $I(v)$ for all $v \in A$ 07 $I_{D_\Delta}^\uparrow :=$ union of the intervals $I_\uparrow(u)$ for all $u \in D$ 08 $A_\Delta := \{v \mid poID(v) \text{ in } I_{A_\Delta}, v \in G.V\}$ 09 $D_\Delta := \{u \mid poID_\uparrow(u) \text{ in } I_{D_\Delta}^\uparrow, u \in G.V\}$ 10 **if** the size of $I_{A_\Delta} <$ the size of I_{D_Δ} 11 **return** the induced subgraph of A_Δ 12 **else**13 **return** the induced subgraph of D_Δ

Figure 3.12: A procedure for determining a subgraph whose reachability information is affected by a given update

3. derive L' from Δ_L and L .

The algorithm to find a small subgraph Δ_G efficiently is presented in Figure 3.12. Given a set of (edge/node) updates δ on G , nodes of G is divided into two sets. (i) A_δ : A node belongs to A_δ *iff* there is an edge *from* the node that is being inserted or deleted. (ii) D_δ : A node belongs to D_δ *iff* there is an edge *to* the node that is being inserted or deleted. Note that A_δ and D_δ cannot be empty at the same time. Since the input graph is a DAG, it is obvious that a node cannot be in both A_δ and D_δ of a given update operation.

Δ_G is obtained from the smaller induced subgraph of A_Δ and D_Δ where A_Δ consists of all descendants of all ancestors of A_δ and D_Δ consists of all ancestors of all descendants of D_δ . It should be remarked that if A_Δ is used, *i.e.*, Line 11 is executed, then our method is the same as the existing approach [STW05] for the case of a node deletion.

Figure 3.13: An example for the 2-hop labels update

Next, we describe how A_Δ and D_Δ can be computed efficiently using the auxiliary structures described previously. To compute A_Δ , we first obtain A , the ancestors of A_δ . This is done in two steps: (i) combine all intervals of nodes in A_δ in $G_\uparrow$ (Line 2) and (ii) identify all nodes belonging to these intervals (Line 4). Next, we compute A_Δ , the descendants of all nodes in A by the following steps: (i) combine all intervals of nodes in A in G (Line 6) and (ii) identify all nodes belonging to these intervals (Line 8). D_Δ can be computed in a similar manner (Lines 3, 5, 7 and 9).

Finally, the induced subgraph Δ_G of the smaller interval of A_Δ and D_Δ is returned (Line 11 and 13).

Example 3.1: We illustrate the algorithm with an example. Consider the (partial) graph and the 2-hop labels shown in Figure 3.13 (a). The centers are marked with dotted circles. Suppose Node 6 is deleted. A_δ and D_δ are $\{\text{Node 5}\}$ and $\{\text{Node 9}\}$, respectively. $I_A^\uparrow$ contains the $poID_\uparrow$ s of Node 1, 2, and 5. Hence A contains Node 1, 2 and 5. Then, A_Δ is computed and contains all nodes. In contrast, by similar processing, D_Δ can be computed, which contains only 7 nodes. Therefore, Δ_G is obtained from induced subgraph of D_Δ , as shown in Figure 3.13 (b). ■

The geometrical approach [CYL⁺06] is used to obtain Δ_L from Δ_G . A partial example of Δ_L is shown in Figure 3.13 (b).

Next, we discuss the remaining steps for deriving L' from L and Δ_L .

Deletions. For a deletion of a node or an edge, the current 2-hop labels L are updated with two requirements: (1) the paths in Δ_G and the paths between nodes in $V \setminus \Delta_V$ and Δ_V are all covered; and (2) any 2-hop labels in L which covers a

disconnected path between $\text{anc}(A_\delta)$ and $\text{desc}(D_\delta)$ must be removed. Based on these, [BCCY] derives equations for updating L . On the one hand, suppose Δ_V is obtained from A_Δ , *i.e.*, Line 11 in Figure 3.12. For each $v \in \Delta_V$, we update L with the following two equations:

$$\begin{aligned} L'_{in}(v) &= (L_{in}(v)) \setminus \text{anc}(A_\delta) \cup \Delta_{L_{in}}(v) \\ L'_{out}(v) &= \Delta_{L_{out}}(v) \end{aligned}$$

The above processing is similar to the existing approach [STW05]. On the other hand, suppose Δ_V is obtained from the induced subgraph of D_Δ , *i.e.*, Line 13 in Figure 3.12. For each $v \in \Delta_V$, L is updated with the followings:

$$\begin{aligned} L'_{in}(v) &= \Delta_{L_{in}}(v) \\ L'_{out}(v) &= (L_{out}(v)) \setminus \text{desc}(D_\delta) \cup \Delta_{L_{out}}(v) \end{aligned}$$

Figure 3.13 (c) demonstrates the above processing for three nodes, Node 1, 11 and 12. The proof of correctness of L' obtained by above processing follows the above two requirements.

Insertions. For insertion of a node or an edge, requirement (2) needs not be taken into considerations. Suppose Δ_V is obtained from induced subgraph of A_Δ , *i.e.*, Line 11 in Figure 3.12. For each $v \in \Delta_V$, L is updated by the following equations.

$$\begin{aligned} L'_{out}(v) &= \Delta_{L_{out}}(v) \\ L'_{in}(v) &= L_{in}(v) \cup \Delta_{L_{in}}(v) \end{aligned}$$

We omit the other case since it is similar.

Analysis. The time to obtain the small subgraph is linear to the size of A_Δ and D_Δ . The time for the update of L is bounded by the reconstruction of the 2-hop labels of the small subgraph, using the geometrical approach, as well as the time for applying the equations. In the worst case, the small subgraph could be as large as the graph itself. However, it will never be larger than the subgraph obtained by

No	Method	Query time	Index time	Index size	Update time
1	Interval (tree)[ABJ89]	$O(1)$	$O(N)$	$O(N)$	$O(N)$
2	Prefix (tree)[KMS02]	$O(1)$	$O(N)$	$O(N)$	N.A.
3	Prime (tree)[WLH04]	$O(N)$	$O(N)$	$O(N^2)$	$O(N)$
4	Dual labeling[WHY ⁺ 06]	$O(1)$	$O(N + M + K^3)$	$O(N + K^2)$	N.A.
5	Interval (DAG)[ABJ89]	$O(N)$	$O(N^3)$	$O(N^2)$	$O(N + E)$
6	Prime (DAG)[WZLL06]	$O(N)$	$O(N + E)$	$O(N^2)$	$O(N + E)$
7	2-hop[CHKZ03]	$O(m^{1/2})$	$O(N^4)$	$O(NM^{1/2})$	N.A.

Table 3.2: Summary of the graph indexing techniques

[STW05].

Despite the efforts on incremental maintenance of 2-hop labels by [STW05, BCCY], to date, there has not been heuristics that considers updates of the 2-hop labels *when they are constructed*. In the next chapter, we shall analyze updates of 2-hop labeling and illustrate how maintenance of 2-hop labels becomes *inherently* complicated if no special effort is spent on the construction of 2-hop labels.

3.4 Summary

Table 3.2 summarizes the characteristics of the graph indexing techniques presented in this chapter under worst case situations.

The proposed method in this thesis also works on graph data, whereas methods 1-3 only works for tree data. Dual Labeling[WHY⁺06] is optimized for cases where $M - N$ is small (much less than N) which limits its application to very sparse graphs or tree-like graph, whereas our proposed method is designed for general directed graph. The extended interval labeling scheme[ABJ89], in the worst case, requires $O(N^2)$ storage which is in the same order as the transitive closure itself. Similar characteristic also applies to extended Prime labeling scheme[WZLL06]. On the contrary, 2-hop labeling[CHKZ03] only requires $O(NM^{1/2})$ storage space. However, the original proposal of 2-hop labeling[CHKZ03] has some limitations: slow construction time and no discussion on update support.

[CYL⁺08, CYL⁺06, STW04] have proposed algorithms to optimize the con-

struction time of *2-hop* labeling. The work proposed in this thesis complements such works by proposing incremental maintenance algorithms to maintain *2-hop* labeling upon update.

Incremental maintenance of *2-hop* labeling has been discussed in [STW05] and [BCCY]. Both methods require a reconstruction of *2-hop* labels as well as transitive closure of the subgraphs affected by the update. In the worst case, the affected subgraph could be as large as the entire graph. Hence, upon node/edge deletion, in the worst case, the entire *2-hop* cover needs to be recomputed.

On the contrary, this work proposes novel incremental maintenance algorithm which does not require reconstruction of transitive closure. In addition, only the labels of the ancestors and descendants of the deleted node is affected.

Chapter 4

Properties for Updatable 2-hop (u2-hop) Labeling

In this chapter, we analyze the inefficient steps in maintenance of *2-hop* labeling. Subsequently, based on our analysis, we propose properties which simplifies the maintenance of *2-hop* labeling.

4.1 Analysis of Updates of 2-hop Labeling

In this section, we perform a case analysis on the steps required to update *2-hop* labels after the deletion of a node of a graph. The aim is to highlight the inefficient steps among them. For *2-hop* labeling, insertions are simpler than deletions (see Chapter 5). Therefore, in this section, we focus on deletions.

Consider a deletion of a node $x \in V$. Let T be the transitive closure of G . The nodes in G can be partitioned into three disjoint sets with respect to x (see Figure 4.1):

- $A(x) = \{a \mid (a, x) \in T\}$, e.g. the ancestor nodes of x
- $D(x) = \{d \mid (x, d) \in T\}$, e.g. the descendant nodes of x
- $R(x) = V - A(x) - D(x)$, e.g. any other nodes that do not belong to $A(x)$ nor $D(x)$

Figure 4.1: Illustration of deletion of x

We omit x from $A(x)$, $D(x)$ and $R(x)$ when it is clear from the context. The updates of 2-hop labels of D are symmetric to those of A . We first discuss the updates of 2-hop labels of nodes in A . An element (a, d) in T , where $d \in L_{out}(a)$, belongs to one of these four disjoint cases:

1. $E_1 = \{(a, d) \mid a, d \in A\}$
2. $E_2 = \{(a, d) \mid a \in A, d \in R\}$
3. $E_3 = \{(a, d) \mid a \in A, d \in D\}$
4. $E_4 = \{(a, x) \mid a \in A\}$

Cases 1 and 2. When x is deleted, E_1 and E_2 are not affected. Specifically, for each (a, d) in E_1 and E_2 , after deletion of x , d would remain in $L_{out}(a)$.

Case 3. To illustrate the updates on E_3 , we describe a procedure for processing the deletion of x . Consider $(a, d) \in E_3$ and $d \in L_{out}(a)$. We need to check if d should still be in $L_{out}(a)$ after the deletion of x . The idea is that if a can no longer reach d after the deletion of x , then d would be removed from $L_{out}(a)$. Checking whether a can still reach d can be done in following way. We check whether or not some of the children of a can reach d via some path(s) that do not pass through x . This is done with the help of 2-hop labels. Here, we assume that we process

```

Procedure check_All_Lin
Input: a directed graph  $G: (V, E)$ , nodes  $a, d \in V$ ,
         $x \in V$  the node to be deleted, a 2-hop cover  $H(G)$ 
01 for each  $d' \in \text{desc}(d)$ 
02   if  $d \in L_{in}(d')$ 
03     check_Lin( $G, a, d', x, H(G)$ )

Procedure check_Lin
Input: a directed graph  $G: (V, E)$ , nodes  $a, d' \in V$ ,
         $x \in V$  the node to be deleted, a 2-hop cover  $H(G)$ 
04 if exist_path( $a, x, d'$ ) is true
05    $L_{out}(a) := L_{out}(a) \cup \{d'\}$ 

```

Figure 4.2: Procedure check_All_Lin and check_Lin

E_3 in topological order of nodes in A , starting from x . Hence, the 2-hop labels of the children have been updated in previous steps.

If a can no longer reach d after the deletion of x , then d would be removed from $L_{out}(a)$. But this might cause some existing reachability information to be removed from the index. For instance, (a, d') may have been removed from the 2-hop cover due to the removal of d from $L_{out}(a)$.

Hence, we need to perform some *additional* checks to the descendants of d . The additional check is done by Procedure check_All_Lin depicted in Figure 4.2: Consider a descendant d' of d where $d \in L_{in}(d')$, i.e., d' uses d as a center node. For each such d' , we perform check_Lin(a, d') as follows: We need to check if there are some paths from a to d' that do not pass through x (Line 04). If there is such a path, then d' should be added into $L_{out}(a)$ to maintain the connectivity (Line 05). In the worst case, we need to consider $|A| \times |D|$ check_Lin cases. Note that finding such path can be done efficiently as described in Chapter 5.

Case 4. We consider (a, x) for all $a \in A$, $d \in L_{out}(a)$, and (a, x) for all $d \in D$, $a \in L_{in}(d)$ together. We define two sets: $P: \{p \mid x \in L_{out}(p), p \in A\}$ and $Q: \{q \mid x \in L_{in}(q), q \in D\}$, where x is the node to be deleted. It is obvious that x would be removed from L_{out} of nodes in P and L_{in} of nodes in Q . However, as mentioned in Case 3, this might cause some existing reachability information to be removed

from the index, for instance (a, d''') . Hence, for each $p \in P$ and $q \in Q$, we execute $\text{check_Lin}(p, q)$. Case 4 requires at most $|P| \times |Q|$ check_Lins .

We can define Case 1-4 for the labels of nodes in D in similar manner. The difference is that we consider an element (a, d) in T , where $a \in L_{in}(d)$. As before, Case 1 and 2 are not affected. Elements of E_3 is defined as (a, d) , $a \in L_{in}(d)$, a in A and d in D . Case 3 can be processed with the same approach. There are some minor differences: 1) To perform reachability test, we use the 2-hop labels of the parents (instead of the children); 2) The processing is done in reverse topological order. Note that the processing of E_3 of nodes in A and E_3 of nodes in D must be done in parallel in the correct order (see Chapter 5 for details); 3) check_Lin checks each ancestor a' of a and might need to add a' to $L_{in}(d)$. Finally, Case 4 is processed together with Case 4 of nodes in A .

The bottleneck of deletion of 2-hop labels is check_Lin computation which is used heavily in Case 3 and Case 4. Hence, in order to efficiently process the node deletion, check_Lin needs to be executed efficiently or if possible eliminated.

In this section, we have shown that node deletion can be processed without invoking the construction algorithm which is often inefficient, as proposed by the existing approaches[STW05, BCCY].

Deletion of an edge can be processed by a similar procedure. This is described in more detail in Section 5.

4.2 Properties for Updatable 2-hop (u2-hop) Labeling

Based on the analysis in Section 4.1, we present the node-separation property and merging property that lead to simplified maintenance, especially on edge/node deletions.

4.2.1 Node separation property

The heuristics of *u2-hops* are derived from the *node-separation* property: We say that a set of nodes X *separate* u and v if and only if the removal of all nodes in X disconnects u and v . In other words, every path from u to v must contain at least one node in X . We consider that if X contains u or v then X separates u and v . This definition extends to the case where u and v are a set of nodes respectively. That is, a set of nodes X *separate* a set of nodes U and a set of nodes V if and only if the removal of all nodes in X disconnects all pairs of (a,d) , where $a \in U$ and $d \in V$.

Definition 4.1: A *2-hop* cover satisfies the node-separation property if and only if for each element (u, v) in T , the center nodes of u and v *separate* u and v . We define the *center nodes of u and v* to be $\{x \mid x \in L_{out}(u) \cap L_{in}(v)\}$. ■

It is interesting to note that transitive closure of a graph G in adjacency list format satisfies *node-separation* property. However, as discussed before, its size could be huge.

When a *2-hop* cover satisfies the node-separation property, on deletion of a node, the processing of E_3 and E_4 can be simplified as follows: (1) `check_All_Lin` is no longer required for E_3 ; (2) x can simply be removed from $L_{out}(a)$ and $L_{in}(d)$ for E_4 . Unlike the deletion described in Chapter 4.1 (Case 3), we do not add new nodes into the *2-hop* cover.

The correctness can be argued as follows: (a) After deletion of x , if a is not connected to d but a is still connected to d' then there must be an alternative path from a to d' not through x . According to the node-separation property, center nodes of a and d' must contain at least one representative node on each alternative path from a to d' , i.e. in both $L_{out}(a)$ and $L_{in}(d')$. Hence, the information that a can still reach d' is not accidentally removed from the index. (b) If a is no longer connected to d' . This implies that the set of paths from a to d' always include d . Hence, any other center nodes of a and d' must be either an ancestor or a descendant of d . Hence, all the center nodes of a and d' belong to E_3 and will be checked and removed in some other step of the algorithm. The arguments for the correctness of Case 4 is similar.

Example 4.1: Consider the example graph G_0 shown in Figure 2.4 (a). The *2-hop*

cover, as shown, does not exhibit the node-separation property because the center nodes of v_1 and v_8 is $\{v_6\}$, which does not separate v_1 and v_8 . After the removal of v_6 , v_1 can still reach v_8 through v_3 . Similarly, the center nodes of v_1 and v_9 , and the center nodes of v_3 and v_8 do not satisfy the node-separation property.

After the deletion of v_6 , the 2-hop labels need to be updated by deleting v_6 from L_{in} and L_{out} of all nodes. In addition, it is necessary to insert v_3 into $L_{in}(v_8)$ and $L_{in}(v_9)$ to cover the paths from v_1 and v_3 to v_8 and v_9 .

In comparison, suppose that v_8 is added to $L_{out}(v_1)$ and $L_{out}(v_3)$ and v_3 is added to $L_{in}(v_8)$. The resulting 2-hop cover satisfies the node-separation property. For example, the center nodes of v_1 and v_8 , $\{v_3, v_6\}$, separates v_1 and v_8 . In this case, the deletion of v_6 could be processed by simply removing v_6 from L_{in} and L_{out} of all nodes.

■

We found that determining the 2-hop cover that satisfies the node-separation property with the minimum size is NP-hard. This can be easily established by a reduction from an instance of the minimum SET COVER problem. We sketch the main idea of the proof here: Given a collection of elements $S = \{p_1, \dots, p_n\}$ and a collection of subsets $S_1, \dots, S_m$, where $S_i \subseteq S$. We construct a graph G with vertices for (1) $p_1, \dots, p_n$, (2) S_i 's, (3) a source node a and (4) a sink node d . For each p_i , we construct a path from a to d . Suppose p_i appears in $S_1, S_2, \dots$, and S_j , then there will be a path, representing p_i , in G going through exactly $a, S_1, S_2, \dots, S_j$ and d . Hence, selecting an S_x , where x in $[1 \dots j]$ will cover the path of p_i . We could find the minimum S_i 's to cover all alternative paths, i.e., the 2-hop cover with minimum size, iff we could find a solution for the minimum SET COVER problem.

4.2.2 Merging Property

Next, we discuss the *merging* property that is used in the construction of *u2-hops* that satisfy the node-separation property. Consider possibly overlapping subsets of $T(G)$: $T_1, T_2, \dots, T_m$, where each T_i represents partial connectivity of a graph G and $\bigcup_{i=1..m} T_i = T$. Suppose each T_i is covered by the 2-hop labels H_i . Reachability query can be done by independently querying H_i 's.

Definition 4.2: The *merging* property states that if T is covered by $H_1, H_2, \dots, H_m$ and each H_i satisfies the node-separation property, then we can merge H_i for $i = 1..m$ into a single 2-hop H_{all} and H_{all} also satisfies the node-separation property. The merging is defined as follows:

- $L_{out}(a) = \bigcup_{i=1..m} L_{out}(a)$ of H_i , for all i .
- $L_{in}(a) = \bigcup_{i=1..m} L_{in}(a)$ of H_i , for all i .

■

The correctness of this property is immediate: Consider an element (a, d) of T . According to the definition, there exist H_i where the center nodes of a and d in H_i separate (a, d) . Adding more nodes into the 2-hop labels does not violate the node-separation property.

Chapter 5

Incremental Maintenance

Algorithms for $u2$ -hop Labeling

In this chapter, we describe our incremental maintenance algorithms for 2 -hop labeling that satisfy node-separation property (denoted as $u2$ -hop).

5.1 Deletion of a node

The deletion algorithm of $u2$ -hops is presented in Algorithm `delete` in Figure 5.1. The inputs of Algorithm `delete` are a directed graph G , its $u2$ -hop cover H and a node to-be-deleted x . The output of the algorithm is the updated 2 -hop cover that still satisfies the node-separation property. We remove x from $L_{in}(v)$ and $L_{out}(v)$ for all $v \in V$ (Line 01). This deals with Case 4 in Section 4.1. We obtain the ancestors and descendants of x with the help of the input graph G (Line 02). We sort the ancestors and descendants based on the topological ordering (Line 03-04). Next, we gather the edges that belong to Case 3 in E_3 (Lines 05-06). We perform deletions of edges in Case 3 in topological order. We sort the edges of E_3 by their indexes in A' and D' (Line 07). Hence, when we process an edge (a, d) in E_3 , the relevant 2 -hop labels have been correctly updated. Then, we scan through the edges in E'_3 , the sorted E_3 (Line 08). For each edge (a, d) in E'_3 , there are only two possible cases: (i) d belongs to $L_{out}(a)$ or (ii) a belongs to $L_{in}(d)$ (Lines 09 and 12). (i) For the first case, we check if a can still reach d

Procedure delete

Input: a directed graph G , an $u2$ -hop cover of G and
 x a vertex to-be-deleted

Output: the updated $u2$ -hop cover

```

01 remove  $x$  from  $L_{in}$  and  $L_{out}$  of all nodes
02 compute  $A$ :  $A(x)$  and  $D$ :  $D(x)$ 
03  $A' := \text{sort } A$  in reverse topological order (“bottom up”)
04  $D' := \text{sort } D$  in topological order (“top down”)
05  $E_3 := (a, d_a), d_a \in L_{out}(a), d_a \in D'$  and  $a \in A'$ 
06  $E_3 := E_3 \cup (a_d, d), a_d \in L_{in}(d), a_d \in A'$  and  $d \in D'$ 
07  $E'_3 := \text{sort } E_3(a, d)$  by  $i$  then by  $j$  in ascending order
   where  $a = A'[i]$  and  $d = D'[j]$ 
08 for each  $(a, d)$  in  $E'_3$  in order
09   if  $d \in L_{out}(a)$ 
10     if  $c_a \not\rightsquigarrow d$  for all children  $c_a$  of  $a$  (after deletion)
11       remove  $d$  from  $L_{out}(a)$ 
12   if  $a \in L_{in}(d)$ 
13     if  $a \not\rightsquigarrow p_d$  for all parent  $p_d$  of  $d$  (after deletion)
14       remove  $a$  from  $L_{in}(d)$ 

```

Figure 5.1: Deletion of a node in $u2$ -hop labeling

after the deletion of x . The checking is done by using the 2 -hop cover to test if any child c_a of a can reach d . Specifically, $L_{out}(c_a) \cap L_{in}(d) \neq \{\}$. If no children of a can reach d , we remove d from $L_{out}(a)$ (Lines 10-11). (ii) For the second case, we remove a from $L_{in}(d)$ if a cannot reach d after the deletion of x (Lines 13-14). This checking is done by checking the parents of d in a similar manner. We emphasize that the 2 -hop cover can be used for reachability tests in Lines 10 and 13 because the relevant 2 -hop labels have been correctly updated in previous steps.

Example 5.1: Consider a deletion of v_6 in the graph G_0 presented in Figure 2.4. We use the 2 -hop cover depicted in Figure 2.4 (b). Note that the 2 -hop cover satisfies the node-separation property. The deletion algorithm removes v_6 from L_{out} of v_1, v_3 and v_4 as well as $L_{in}(v_7)$ (Line 01). The set of ancestors and descendants are obtained: $A = \{v_1, v_3, v_4\}$ and $D = \{v_7, v_8, v_9\}$. The topologically

sorted representation is as follows: $A'=\{v_4, v_3, v_1\}$ and $D'=\{v_7, v_8, v_9\}$. The set of edges in Case 3, E_3 is $\{(v_1, v_8), (v_3, v_8), (v_4, v_8)\}$. The sorted E_3, E'_3 , is $\{(v_4, v_8), (v_3, v_8), (v_1, v_8)\}$. Note that all of these edges will be processed by Lines 10-11. Next, we process E'_3 in sequence: 1) process (v_4, v_8) : we check if v_4 can still reach v_8 after deleting v_6 . The only remaining child of v_4 is v_5 . Since $L_{out}(v_5) \cap L_{in}(v_8)$ is empty, they are not connected, thus we remove v_8 from $L_{out}(v_4)$. 2) process (v_3, v_8) : since v_8 is a child of v_3 , then we keep v_8 in $L_{out}(v_3)$. 3) process (v_1, v_8) : v_1 has three children, v_2, v_3 and v_4 . v_2 cannot reach v_8 . But, as we have processed in Step 2), v_3 can reach v_8 . Thus, we keep v_8 in $L_{out}(v_1)$. ■

We also remark that Algorithm `delete` can be extended to handle deletions of arbitrary 2-hop covers, including those proposed by [STW05, CYL⁺06]. This can be implemented with `check_All_Lin`, as described in Section 4.1. Specifically, if d is to be removed from $L_{out}(a)$ (Line 11), then we check all $d' \in D$ having $d \in L_{in}(d')$. Thus, if a can still reach d' after the deletion of x (again, this checking is done through 2-hop reachability test), then we need to add d' to $L_{out}(a)$ to restore back the reachability information. These *additional* operations must be performed consistently to the edge order in E'_3 . As an optimization, if (a, d') is already covered by the current 2-hop labels after the removal of d from $L_{out}(a)$, the previous steps can be skipped. Similarly, if a is to be removed from $L_{in}(d)$ (Line 14), a symmetric processing is needed: for each a' having a in $L_{out}(a')$, we might need to add a' to $L_{in}(d)$ if a' can still reach d after deletion of x .

From the discussion above, it is clear that the deletion algorithm can also be extended to work on the hybrid of *u2-hop* and non-*u2-hop* (although the details are somewhat involved), *e.g.*, the hybrids of 2-hops described in Section 6.2. This extension has been used in our experiment. To extend the algorithm to work on hybrids of 2-hop that do not satisfy the node-separation property, the two 2-hop covers are to be updated in parallel.

Complexity. Sorting the nodes or edges in some topological order can be implemented efficiently. The dominating steps in Algorithm `delete` are Lines 8-14. For each edge in E_3 , we performed at most $|V|$ 2-hop lookups, Lines 10 and 13. Hence, the complexity of the deletion is $O(|E_3| \times |V|)$ 2-hop lookups. However, in practice, the number of lookups required is much fewer than this. This is because

Procedure delete-edge**Input:** a directed graph G , an $u2$ -hop cover of G and (x, y) an existing edge to-be-deleted**Output:** the updated $u2$ -hop cover01 If x can still reach y after the deletion

02 return

03 remove y from $L_{out}(x)$ and x from $L_{in}(y)$ 04 compute A : $A(x)$ and D : $D(y)$

05 execute Steps 03 - 14 in Algorithm delete

Figure 5.2: Deletion of an edge in $u2$ -hop labeling

in most cases, the number of children of a node is much less than the total number of nodes. Hence, $O(|V|)$ 2-hop lookups bound is a very loose bound.

5.2 Deletion of an edge

Deletion of an edge can be processed by Algorithm delete-edge, a modification of Algorithm delete. Consider a deletion of an existing edge (x, y) . Note that since (x, y) already exists, either $x \in L_{in}(y)$ or $y \in L_{out}(x)$. We first check if x can still reach y after deletion. The checking is done by using the 2-hop reachability test of x 's children excluding y . If x can still reach y , there is no other connection removed due to the removal of (x, y) . Hence, no other update is necessary (Lines 01-02). Otherwise, we remove y from $L_{out}(x)$ and x from $L_{in}(y)$ to disconnect y from x (Line 03). In addition, we need to check if all connections from $A(x)$ to $D(y)$ are still valid after the edge deletion. This is done as in the deletion of node (Line 04-05). It is straightforward that Algorithm delete-edge preserves the node-separation property of the 2-hop labels.

Example 5.2: Consider a deletion of edge (v_6, v_8) in the graph G_0 presented in Figure 2.4. As before, we use the 2-hop cover that satisfies node-separation property and depicted in Figure 2.4 (b). The only child of v_6 (excluding v_8) is v_7 and since $L_{out}(v_7) \cap L_{in}(v_6) = \emptyset$, v_7 can not reach v_8 . Since, v_6 can no longer reach v_8 after the deletion of the edge, the algorithm proceed (Line 01). Next, in Line 03,

v_6 is removed from $L_{in}(v_8)$. The set of ancestors and descendants are computed: $A = \{v_6, v_4, v_3, v_1\}$ and $D = \{v_8, v_9\}$. Next, Line 03-14 of Algorithm `delete` is executed, in which v_6 is removed from $L_{in}(v_9)$. ■

5.3 Insertion of an edge

Algorithm `insert`, as shown in Figure 5.3, handles insertions of $u2$ -hops. We aim at an insertion algorithm that preserves the node-separation property. For simplicity, we assume that the insertion would not introduce cycles to the graph. Consider a single-edge insertion (x, y) . Suppose x already exists in the graph and y is new (Line 01). All nodes that can reach x can also reach y . Hence, we insert $L_{in}(x)$ together with x in $L_{in}(y)$ (Line 02). Case 2 (Lines 03-04) is symmetric to Case 1.

Case 3 deals with insertions of an edge between two existing nodes. Case 3 can be processed by a simple algorithm as follows: add y to L_{out} of $A(x)$ and L_{in} of $D(y)$. This approach is efficient and preserves the node-separation property. The latter is due to the fact that adding more nodes in the 2 -hop labels will never violate the property and all new paths introduced have been covered by the insertion of y to the 2 -hop labels of nodes. The shortcoming of this approach is that the index size could increase by $|A(x)| + |D(y)|$.

We propose an insertion algorithm that would reduce this increase of the index size. The idea is to exploit the node-separation property. More specifically, under certain condition, it is not necessary to add y to the labels of some nodes.

We divide Case 3 into 2 sub-cases: Case 3a: x already reaches y before the insertion; and Case 3b: otherwise.

Case 3a is illustrated in Figure 5.4. M is the set of nodes between x and y , i.e., nodes that are ancestors of y and descendants of x . A and D are ancestors of x and descendants of y , respectively. Due to the node-separation property, the center nodes $C_{a,d} = L_{out}(a) \cap L_{in}(d)$, where $a \in A$ and $d \in D$, separate a and d . This implies that the path(s) from a to d through x or through y has already been included in $C_{a,d}$. Note that the insertion of an edge (x, y) introduces a new path from a to d through (x, y) . Then, $C_{a,d}$ does not separate a and d only if $C_{a,d}$

Procedure insert

Input: a directed graph G , an $u2$ -hop cover,
an edge to-be-inserted (x, y)

Output: the updated $u2$ -hop cover

```

01 case 1:  $x \in V$  and  $y \notin V$ 
02    $L_{in}(y) := L_{in}(x) \cup \{x\}$ 
03 case 2:  $x \notin V$  and  $y \in V$ 
04    $L_{out}(x) := L_{out}(y) \cup \{y\}$ 
05 case 3a:  $x, y \in V$  and  $(x, y) \notin E$ ,  $x$  can reach  $y$ 
06   compute  $A$ :  $A(x)$  and  $D$ :  $D(y)$ 
07    $M := D(x) \cap A(y)$ 
08    $\hat{M} := \{p \mid p \in M, p \in L_{out}(a), \text{ for some } a \in A\}$ 
09   for each  $d \in D$ 
10     if  $L_{in}(d) \cap \hat{M}$  is not empty
11        $L_{in}(d) := L_{in}(d) \cup \{y\}$ 
12    $\hat{M} := \{p \mid p \in M, p \in L_{in}(d), \text{ for some } d \in D\}$ 
13   for each  $a \in A$ 
14     if  $L_{out}(a) \cap \hat{M}$  is not empty
15        $L_{out}(a) := L_{out}(a) \cup \{y\}$ 
16 case 3b:  $x, y \in V$  and  $(x, y) \notin E$ ,  $x$  cannot reach  $y$ 
17   compute  $A$ :  $A(x)$  and  $D$ :  $D(y)$ 
18   for each  $a \in A$ 
19      $L_{out}(a) := L_{out}(a) \cup \{y\}$ 
20   for each  $d \in D$ 
21      $L_{in}(d) := L_{in}(d) \cup \{y\}$ 

```

Figure 5.3: Insertion of a new edge in $u2$ -hop labeling

Figure 5.4: Illustration of Case 3a: insertion of new edge (x, y) . y will be added to $L_{in}(d_2)$ and $L_{out}(a_2)$, but not to $L_{in}(d_1)$ nor $L_{out}(a_1)$.

contains some nodes in M and hence violates the node-separation property. Thus, it is necessary to add y to $L_{in}(d)$, if there exists a such that $C_{a,d}$ contains some nodes in M . In another case, we can skip adding y to $L_{in}(d)$. Similar arguments apply to $L_{out}(a)$.

For example, consider the insertion of (x, y) illustrated with Figure 5.4. y needs to be added to $L_{out}(a_2)$ and $L_{in}(d_2)$. This is because C_{a_2,d_2} contains m . This is not the case for C_{a_1,d_1} . Hence, y is added to neither $L_{out}(a_1)$ nor $L_{in}(d_1)$.

The algorithm for Case 3a starts by computing A and D (Line 06). Next, the algorithm computes M and a subset of M that is included in L_{out} of some nodes in A , denoted by $\hat{M}$ (Lines 07-08). The next step is to identify nodes $d \in D$ where y needs to be added to $L_{in}(d)$. As discussed before, the algorithm adds y to $L_{in}(d)$ if $L_{in}(d)$ contains nodes in $\hat{M}$ (Lines 9-11). Lines 12-15 perform similar processing for L_{out} of nodes in A .

The analysis above does not handle Case 3b where x and y are not connected before the insertion (Case 3b). In response to this, Lines 17-21 adds y to the labels of A and D accordingly.

Complexity. The dominating steps are Lines 08-15. There are at most $O(|V|)$ 2-hop label intersections (Lines 08, 10, 12 and 14) and $O(|V|)$ updates of 2-hop labels (Lines 11 and 15).

Example 5.3: In this example, we illustrate Case 3a which is the most complex

```

Procedure insert
Input: a directed graph  $G$ , an  $u2$ -hop cover,
        a subgraph  $G'(V', E')$  to-be-inserted
Output: the updated  $u2$ -hop cover
01  $H' := \text{construct } 2\text{-hop cover of } G'$ 
02  $E_{cross} := \text{the crossing edges from } G' \text{ to } G$ 
03  $C_{cross} := \text{compute min vertex cover of } E_{cross}$ 
04 for each  $c \in C_{cross}$ 
05   for  $v$  in  $V$ 
06     if  $v \rightsquigarrow c$  then 07        $L_{out}(v) := L_{out}(v) \cup \{c\}$ 
08     else if  $c \rightsquigarrow v$  then 09        $L_{in}(v) := L_{in}(v) \cup \{c\}$ 

```

Figure 5.5: Insertion of a new subgraph in $u2$ -hop labeling

case. Consider an insertion of edge (v_4, v_8) . Line 06 computes $A = \{v_4, v_3, v_1\}$ and $D = \{v_8, v_9\}$. Line 07 computes the set of nodes between v_4 and v_8 , $M = \{v_6\}$. Next, $\hat{M} = \{v_6\}$ (Line 08), this is due to v_6 is included in $L_{in}(v_8)$ or $L_{in}(v_9)$. Next, we iterate through elements of D (Line 09). We add v_8 to $L_{in}(v_8)$ and $L_{in}(v_9)$ since both $L_{in}(v_8)$ and $L_{in}(v_9)$ includes v_6 (Line 10-11). The algorithm performs similar operations to A (Line 12-15). Again, $\hat{M} = \{v_6\}$ (Line 12). We iterate through elements of A (Line 13) and add v_8 to $L_{out}(v_4)$, $L_{out}(v_3)$, and $L_{out}(v_1)$ (Line 14-15). ■

5.4 Insertion of a subgraph

We end this chapter with a discussion on the insertion of a subgraph G' into an existing graph G . While, this can be done by repetitive insertion of edges using Algorithm 5.3, the updated index size is potentially much larger than the optimal one. Hence, a specialized algorithm to handle subgraph insertion is beneficial.

The subgraph insertion algorithm is provided in Figure 5.5. First, we build the 2 -hop cover of the induced subgraph of the new nodes in G' , $H'(G')$. Second, we handle the insertion of E_{cross} , the crossing edges between G' and G . It is easy to see that the nodes in E_{cross} form a bipartite graph. We cast E_{cross} into an

undirected bipartite graph. We determine the minimum vertex cover C of E_{cross} . The purpose of this is to get the minimum set of nodes that separate G and G' .

While the general minimum vertex cover problem is NP-complete[Kar72], but since we are dealing with bipartite graph, the minimum vertex cover can be computed in PTIME using Konig's theorem[Kn31]: in any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. Hence, we can use network flow technique[RL01]¹ to solve the problem. That is, we use maximum flow² algorithm[RL01] to compute the maximum matching³ of the bipartite graph and based on Konig's theorem[Kn31], we can derive the minimum vertex cover from the solution of the maximum matching problem.

It is easy to see that the nodes in minimum vertex cover separate G and G' . Hence, they will be used as the center nodes.

Consider a node c in C . For each ancestor a of c , we insert c into $L_{out}(a)$. Similarly, for each descendant d of v_2 , we add c into $L_{in}(d)$. It is easy to see that the updated 2-hop cover satisfies node-separation property.

Example 5.4: Consider the graph G_0 shown in Fig. 2.4. Let $V_1 = \{v_1, v_2, v_3, v_4, v_5\}$ and G_1 be a graph induced by V_1 in G_0 . Let $V_2 = \{v_6, v_7, v_8, v_9\}$ and G_2 be a graph induced by V_2 augmented the crossing edges: $\{(v_1, v_6), (v_3, v_8), (v_4, v_6)\}$. Consider that we have constructed the 2-hop cover of G_1 and we are going to insert subgraph G_2 . Procedure `insert` as shown in Figure 5.5 computes 2-hop cover of G_2 (Line 01). Next, it process the crossing edges to obtain the final 2-hop cover. The edges in the bisection E_{cross} are $\{(v_1, v_6), (v_3, v_8), (v_4, v_6)\}$ (Line 02). Next, the minimum vertex cover is computed (Line 03). The nodes in the min-vertex cover C_{cross} are $\{v_6, v_8\}$ (Line 03). We process the nodes in the minimum

¹A flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a source, which has more outgoing flow, or sink, which has more incoming flow. Some examples: traffic in a road system, fluids in pipes, currents in an electrical circuit.

²The maximum flow problem is to find a feasible flow through a single-source, single-sink flow network that is maximum.

³Maximum flow problem in bipartite graph is equivalent to maximum matching problem (e.g. find a set of edges with maximum cardinality such that no two edges share a common vertex). Details are available in [RL01]

vertex cover one by one (Lines 04-08). First, we process v_6 . We add v_6 to L_{in} of $\{v_7, v_8, v_9\}$ as well as L_{out} of $\{v_1, v_3, v_4\}$. Next, we process v_8 . Similarly, we add v_8 to $L_{in}(v_9)$ as well as $L_{out}(v_1)$ and $L_{out}(v_3)$. ■

Chapter 6

Updatable 2-hop (u2-hop) Labeling Construction

6.1 Family of Updatable 2-hop (u2-hop) Labeling

As discussed in earlier Chapter, the hardness of finding the minimum *2-hop* cover that satisfies the node-separation property is intractable. In this chapter, we describe three heuristic functions that guide the construction of *2-hops* that satisfy the node-separation property. First, we consider X as a singleton set – a cut vertex of a subgraph. Second, we consider X as a bisection cut in G . Third, we propose a simplified heuristic function that uses node deletion.

6.1.1 2-hop Cover with Cut Vertex

We first introduce *u2-hop-A*, our first attempt to construct *2-hop* cover with node-separation property. Unlike, previous approaches which use set cover heuristic, *u2-hop-A* is based on cut vertex. It is based on slight modification of arbitrary *2-hop* construction presented in 2.5. The minor modification are as follows:

- An additional constraint on the edges of the bipartite graphs
For each node $x \in V$, we construct a bipartite graph $G_x(A, D, E_x)$, where A and D are $A(x)$ and $D(x)$ respectively and the edges E_x are $\{(a, d) \mid (a, d) \in T, x \text{ separates } a \text{ and } d, a \in A, d \in D\}$.

Figure 6.1: Bipartite graph G_x for each node $x \in G_0$

Example 6.1: Figure 6.1 shows the bipartite graph G_x constructed from each $x \in V_0$. Consider G_{v_4} in Figure 6.1. It shows the bipartite graph G_{v_4} . From the graph G_0 depicted in Figure 2.4, v_4 separates v_3 from v_5 , v_6 and v_7 . Hence, there is an edge from v_3 to v_5 , v_6 and v_7 in G_{v_4} . Note that it is possible that G_x does not have any edge, e.g., G_{v_2} in Figure 6.1. It is also possible that one side of the graph is empty, e.g., G_{v_1} in Figure 6.1. The biclique $B_{v_8}(\{v_1, v_3, v_4, v_6\}, \{v_9\})$ covers $4 + 1 + 4 \times 1 = 9$ elements. This is the maximum among all possible bicliques in G_x for all x . Another biclique $B_{v_1}(\{\}, \{v_2, v_3, v_4, v_5, v_6, v_7, v_8\})$ only covers $7 + 0 + 7 \times 0 = 7$ elements. ■

- Finding largest biclique instead of densest subgraph

Given a bipartite graph G_x , we are interested in finding $A' \subseteq A$ and $D' \subseteq D$ that x separates. From the definition of G_x , this is equivalent to finding A' and D' in which there is an edge (a, d) , for all $a \in A'$ and $d \in D'$. Note that the induced subgraph of A' and D' is a biclique in G_x . Hence, our problem is equivalent to finding a biclique in G_x .

Similar to other heuristic algorithms, to minimize the index size, at each step, we greedily maximize the number of elements of T that are covered. This is analogous of the existing method in [CYL⁺06]. Consider a biclique $B_x(A', D')$. The 2-hop cover is updated to cover B_x as follow. We add x to $L_{out}(a)$ and $L_{in}(d)$, for all $a \in A'$ and $d \in D'$. The number of elements of T that are covered is $|A'| + |D'| + |A'| \times |D'|$. Hence, in each iteration, we find the node x whose biclique $B_x(A', D')$ maximizes $|A'| + |D'| + |A'| \times |D'|$.

$u2\text{-hop-}A$ is constructed by iteratively finding biclique $B_x(A', D')$ and augmenting $u2\text{-hop-}A$ to cover B_x 's until T is fully covered.

We remark that a node x can be chosen as a center node multiple number of times (each at different iteration). This does not cause any problem due to the merging property discussed previously. More importantly, this guarantees that the $u2\text{-hop-}A$ construction terminates and also covers all elements of T .

At the end of each iteration, the algorithm maintains an invariant that the partial 2-hop cover constructed satisfy node-separation property with respect to the elements of T covered so far. This can be seen as follows. Consider an arbitrary iteration of the algorithm. Let $B_x(A', D')$ be the largest biclique chosen. The elements of T covered by the algorithm, denoted as S , are the following set $\{(a, d) | a \in A', d \in D'\}$. Let H be the entries to be added to 2-hop cover. From the definition of $B_x(A', D')$, it is clear that H satisfies node-separation property with respect to S .

Hence, 2-hop cover constructed by this heuristic algorithm satisfies the node-separation property.

Unfortunately, we find that given a bipartite graph G , finding a biclique B in G that covers the maximum number of elements in the transitive closure T is intractable. Specifically, we proved the following theorem. (Note that it is neither the maximal independent set problem of a bipartite graph nor the maximum edge biclique problem.)

Theorem 1.1: [MSEBP] *Given a bipartite graph $G(A, D, E)$, finding a biclique $B(B_X, B_Y)$ that maximizes $f(B) = |B_X| + |B_Y| + |B_X| \times |B_Y|$ is NP-complete, where $B_X \subseteq A$ and $B_Y \subseteq D$.* ■

Proof: (Sketch) Our reduction is established from an NP-complete problem, namely, the *maximum edge bipartite problem* (MEBP) [Pee03]: Given a bipartite graph $G(A, D, E)$, MEBP finds a biclique $B(B_X, B_Y)$ having $g(B) = |B_X| \times |B_Y| \geq k$.

Note that for both problems, we only need to consider maximal biclique B , otherwise the biclique can be extended and produce larger value of $f(B)$ and $g(B)$.

Given an instance of MEBP on an input graph $G(A, D, E)$, we generate $|A| \times |D|$ instances of MSEBP. Specifically, for each pair of nodes $a \in A$ and $d \in D$, we generate an instance of MSEBP $\mathcal{G}_{a,d}$ as follows: We remove (1) a and d ; (2) $d' \in D$ where d' is not adjacent to a , e.g., there is no edge (a, d') ; and (3) $a' \in A$ where a' is not adjacent to d , e.g., there is no edge (a', d) . The graph induced by the remaining nodes is an instance of MSEBP.

Consider any maximal biclique $B(A', D')$ in G having $g(A', D') = |A'| \times |D'| = k$. In every instance $\mathcal{G}_{a,d}$, there exists some nodes in B but not in $\mathcal{G}_{a,d}$. We refer to the subgraph of B in $\mathcal{G}_{a,d}$ as the reduced biclique, denoted as $B_{red}(A'_{red}, D'_{red})$. We show that 1) in all $\mathcal{G}_{a,d}$ generated, $f(B_{red}) \leq k - 1$ and 2) there are some $\mathcal{G}_{a,d}$ having $f(B_{red}) = k - 1$.

Figure 6.2: (a) The input graph for MEBP (b) An instance of MSEBP generated from v_2 and v_6 (c) An instance of MSEBP generated from v_1 and v_4 (d) An instance of MSEBP generated from v_2 and v_5

We first give a proof of 2). Consider an instance $\mathcal{G}_{a,d}$, where $a \in A'$ and $d \in D'$. In this instance, there is *exactly* one node in A' and D' , respectively, that is not in $\mathcal{G}_{a,d}$, e.g., $A'_{red} = A' \setminus \{a\}$ and $D'_{red} = D' \setminus \{d\}$. Hence, $f(B_{red}) = (|A'| - 1) + (|D'| - 1) + (|A'| - 1) \times (|D'| - 1) = |A'| \times |D'| - 1 = k - 1$. This is illustrated in Figure 6.2. Consider the biclique B and the reduced biclique B_{red} in Figures 6.2 (a) and 6.2 (b). The values of $g(B)$ and $f(B_{red})$ are 6 and 5, respectively.

We then prove 1). In any $\mathcal{G}_{a,d}$, there is *at least* one node in A' and D' , respectively, that is not in $\mathcal{G}_{a,d}$, e.g., $|A'_{red}| \leq |A'| - 1$, $|D'_{red}| \leq |D'| - 1$. Hence, $f(B_{red}) \leq (|A'| - 1) + (|D'| - 1) + (|A'| - 1) \times (|D'| - 1) \leq |A'| \times |D'| - 1 \leq k-1$. This is illustrated in Figures 6.2 (c) and 6.2 (d), where the values of $f(B_{red})$ are 3 and 1.

Hence, the answer of MEBP is true if and only if there is at least one instance of MSEBP that contains a biclique B_{red} having $f(B_{red}) \geq k - 1$.

In response to this, we reuse an approximation algorithm [BYR02] for MSEBP as the heuristics for *u2-hop-A* construction.

Construction of u2-hop-A. The key issue in constructing a reasonable *u2-hop-A* with a small size is to find a reasonable approximation for MSEBP for each bipartite graph of each node.

We associate a weight to an element of T . Weight 0 (resp. 1) means that the associated element in T has been covered (resp. has not been covered). Initially, all elements of T have not been covered and thus have a weight of 1. In addition, we also associate a weight to all nodes and edges in the bipartite graph $G_x(A, D, E)$. The weight of a node $a \in A$ in G_x is the weight of (a, x) in T . Similarly, the weight of a node $d \in D$ in $G_x(A, D, E)$ is the weight of (x, d) in T . Whereas, the weight of an edge (a, b) in E is the weight of (a, b) in T . The weights are updated in each iteration of the algorithm as T is updated.

Our heuristic function for each bipartite graph is to find a biclique with the maximum sum of weights. This problem can be solved by a 2-approximation algorithm given in [BYR02] (described in Appendix B), namely (2,2)-deletion problem. Note that this problem is a more general problem than MSEBP.

Putting these together, we present a greedy algorithm for *u2-hop-A* construction in Figure 6.3. The algorithm operates as follows. At Line 01, we construct the bipartite graph G_v for each $v \in G$. T' is used to record the uncovered elements in T . Initially, $T' = T$ (Line 02). max stores the center node of B_{max} that would cover T the most. Initially, B_{max} is set to be an empty biclique with weight 0 (Line 03). As long as T' is not fully covered, the iteration repeats. At each iteration, we compute an approximation of the biclique with the maximum weight of the bipartite graph for each node, as discussed above and in Section 6.1.1. Then,

```

Procedure u2-hop-A-construction
Input: a directed graph  $G: (V, E)$ 
Output:  $u2\text{-hop-}A$  of  $G: (L_{in}, L_{out})$ 
01 for each  $v \in V$ 
    construct the bipartite graph  $G_v$ 
02  $T' := T$ 
03  $B_{max}(V_{max_1}, V_{max_2}) := \text{empty biclique}; max := \text{null}$ 
04 while  $T'$  is not empty
05   for each  $v \in V$ 
06     construct max weight biclique  $B_v(V_1, V_2)$  of  $G_v$ 
07     if  $\text{totalweight}(B_v) > \text{totalweight}(B_{max})$ 
08        $B_{max} := B_v; max := v$ 
09    $L_{out}(a) := \{max\} \cup L_{out}(a),$ 
        where  $a \in V_{max_1}$ 
10    $L_{in}(d) := \{max\} \cup L_{in}(d),$ 
        where  $d \in V_{max_2}$ 
11   update  $T'$  according to  $B_{max}$ 
12   update the weights of  $G_v$  for  $v \in V$  according to  $T'$ 

```

Figure 6.3: A greedy algorithm for $u2\text{-hop-}A$ construction

we determine the biclique B_{max} and the center node max with the largest weight among other bicliques (Lines 05-08). At the end of each iteration, we select max as a center node to cover connections between the nodes in V_{max_1} and V_{max_2} in B_{max} (Lines 09-10). We update T' and the weights of all bipartite graphs accordingly before the next iteration (Lines 11 and 12).

Example 6.2: Consider G_0 , as shown in Figure 2.4. The bipartite graphs G_v constructed in Line 1 is shown in Figure 6.1. All the nodes and edges have weight 1. As described in Example 6.1, in the first iteration, we obtain $max = v_8$, $V_{max_1} = \{v_1, v_3, v_4, v_6\}$ and $V_{max_2} = \{v_9\}$. Then, in Lines 09 and 10, v_8 is added to L_{out} of v_1, v_3, v_4 and v_6 and L_{in} of v_9 . Then, the weights of T' is updated. Subsequently, the weights of the bipartite graphs are updated. Figure 6.4 shows the updated weights after the first iteration. At the next iteration, the biclique having the maximal weight is $(\{v_1, v_3, v_4\}, \{v_7\})$ with center node v_6 . Subsequent iterations would choose the following bicliques and their center nodes in this order:

Figure 6.4: Updated weights of bipartite graph G_x , at the end of the first iteration, for each node $x \in G_0$. Dotted edges and no-fill nodes are with weight 0. Solid edges and filled nodes are with weight 1.

$(\{v_3\}, \{v_5\})$ with center v_4 , $(\{v_1, v_2\}, \{\})$ with center v_5 , $(\{v_1\}, \{\})$ with center v_3 , $(\{v_1\}, \{\})$ with center v_2 and, finally, $(\{\}, \{v_3\})$ with center v_1 . The resulting 2-hop cover is shown in Figure 2.4 (b). ■

Complexity. The initial construction of a bipartite graph is as costly as computing the transitive closure, $O(|V| \times (|V| + |E|))$. To compute all bipartite graphs, it takes $O(|V|^2 \times (|V| + |E|))$. The approximation algorithm that we used takes $O(|V| + |E|)$. We adopted the priority queue implementation to optimize the for-loop (Lines 05-08) in *u2-hop-A* construction [CHKZ03]. Overall, the dominating step in the construction is Line 01.

6.1.2 2-hop Cover with Minimum Bisection

Next, we generalize *u2-hop-A* to *u2-hop-B* in this subsection. Specifically, as opposed to choosing a single-node separation, we allow a node separation that may be a set of nodes. The idea is the following.

1. Partition the graph into two partitions
2. Find the center nodes that separate the two partitions.
3. Update two hop cover to cover the partitions
4. Recursively process each of the partitions

There are many ways to partition a graph. We were tempted to use min-cuts to partition the graph. A min-cut of a directed graph G is defined as the minimum number of edges to be removed from the graph such that the graph becomes disconnected. *e.g.*, partition the graph into two partitions such that the number of crossing edges is minimum. However, the construction algorithm may be guided by numerous small cuts and the resulting *2-hop* cover can be large.

To reduce the number of cuts, we opt to use the minimum graph bisection. A bisection of a graph is a partition of a graph into two parts where both parts have roughly the same size. Minimum graph bisection problem finds minimum set of edges to produce a bisection of a graph.

This approach leads to relatively smaller number of iterations and tends to produce smaller *2-hop* covers. While finding the minimum graph bisection is also a classical NP-hard optimization problem, there has been a number of heuristics for solving this problem [KL70]. In particular, we used [Kar] to determine a small bisection (described in Appendix C).

Suppose the bisection B , $B \subseteq E$, divides the graph G into $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, where $|G_1| \approx |G_2|$. We *combine* G_1 and G_2 using the same approach as in subgraph insertion, presented in Algorithm 5.5.

We cast B into an undirected bipartite graph. We determine the minimum vertex cover C of B . This can be computed in PTIME using the network flow technique. While the complexity of computing the minimum vertex cover could be expensive, but in practice, the size of B is much smaller compared to the whole graph.

It is easy to see that the nodes in minimum vertex cover separate G_1 and G_2 . Consider a node c in C . For each ancestor a of c , we insert c into $L_{out}(a)$. Similarly, for each descendant d of v_2 , we add c into $L_{in}(d)$.

```

Procedure u2-hop-B-construction
Input: a directed graph  $G: (V, E)$ ,
          $T'$ : the uncovered elements of  $T$ 
Output:  $u2\text{-hop-}B$  of  $G: (L_{in}, L_{out})$ 
01  $B := \text{min\_bisection}(G)$ 
02 denote the subgraph separated by  $B$  as  $G_1$  and  $G_2$ 
03  $C := \text{min\_vertex\_cover}(B)$ 
04 for each  $c$  in  $C$ 
05   for  $v$  in  $V$ 
06     if  $v \rightsquigarrow c$  then  $L_{out}(v) := L_{out}(v) \cup \{c\}$ 
07     else if  $c \rightsquigarrow v$  then  $L_{in}(v) := L_{in}(v) \cup \{c\}$ 
08 update  $T'$  according to Lines 06-07
09 if  $T'$  is not empty
10   u2-hop-B-construction( $G_1, T'$ )
11   u2-hop-B-construction( $G_2, T'$ )

```

Figure 6.5: A greedy algorithm for $u2\text{-hop-}B$ construction

Next, we construct $u2\text{-hop-}B$ recursively on G_1 and G_2 , denoted by H_1 and H_2 until the transitive closure is entirely covered. Due to the merging property, the final 2-hop cover is obtained by merging H_1 , H_2 and H_B .

While, in this work, $u2\text{-hop-}B$ is constructed using bisection. We can easily derive variations of it by partitioning the graph into arbitrary number of partitions and combine them using the subgraph insertion algorithm. However, this is beyond the scope of this work.

At a first glance, $u2\text{-hop-}B$ might seem similar to the construction approach proposed in [STW05]. The main difference is that in [STW05], once the graph and its transitive closure are small enough to fit the main memory, the construction algorithm in 2.5 is used. The latter causes the 2-hop cover constructed not to satisfy node-separation property. This is not the case for $u2\text{-hop-}B$. In Section 6.2, we discuss construction of hybrid $u2\text{-hop}$ labeling which uses similar approach; yet the 2-hop labeling satisfies node-separation property.

Construction of $u2\text{-hop-}B$. The construction of $u2\text{-hop-}B$ also uses T' to keep track of the uncovered elements in T . We assume that $T' = T$ initially. We use [Kar] to compute the minimum bisection B of G (Line 01). Then, we use

a classical algorithm to determine the minimum vertex cover C of B (Line 03). Next, we add 2-hop labels related to B (Lines 04-07). We updated T' according to the elements of T covered by C (Line 08). If T is not entirely covered, the construction procedure is called recursively with the subgraphs defined by B , G_1 and G_2 (Lines 09-11).

Complexity. In Lines 01-08, the dominating step is the approximation algorithm for finding the minimum bisection of a graph [Kar]. Denote M to be the time complexity for the algorithm. The construction is called at most $|V|$ times in the worst-case. The overall complexity is $O(M \times |V|)$. However, note that M depends on the size of the input graph, which decreases as the recursion proceeds.

6.1.3 2-hop Cover with Node Deletion

While the previous $u2\text{-hops}$ lead to simple maintenance algorithm, their constructions involve non-trivial problems. This motivates us to investigate the third $u2\text{-hop}$, namely $u2\text{-hop-}C$, whose construction is relatively simple. It is named with “node deletion” because $u2\text{-hop-}C$ is built with a sequence of node deletions.

The idea of $u2\text{-hop-}C$ is based on a direct translation of the definition of the node-separation property. That is, for each alternative path p between nodes a and d , there will be at least one node in p selected as the center of a and d .

On the contrary, the other $u2\text{-hops}$ discussed before are not a direct consequence of the node-separation property. Both of them satisfy the node-separation property by imposing an additional restriction on the construction. For instance, $u2\text{-hop-}A$ only considers a single cut vertex to separate two subgraphs. Similarly, $u2\text{-hop-}B$ only considers bisections of the graph. Although it is immediate that such restrictions would allow the 2-hop covers to satisfy the node-separation property, in many cases they are too restrictive and might produce large index.

$u2\text{-hop-}C$ is built iteratively. At each iteration, one node is chosen to be added as a center node of its ancestors and descendants. The chosen node (and its adjacent edges) is removed from the graph. This is node deletion. This process is repeated until there is no more edges in the graph, which implies that there is no alternative path remaining. To minimize the index size, at each iteration, the node with the densest connectivity are chosen.

<p>Procedure $u2\text{-hop-C-construction}$ Input: a directed graph $G: (V, E)$, Output: $u2\text{-hop-C}$ of $G: (L_{in}, L_{out})$</p> <pre> 01 $G' := G$ 02 while G' has an edge 03 determine v in G' where $(A(v) \times D(v)) / (A(v) + D(v))$ is maximized 04 add v into L_{out} of each node in $A(v)$ 05 add v into L_{in} of each node in $D(v)$ 06 remove v and its edges from G' </pre>
--

Figure 6.6: A greedy algorithm for $u2\text{-hop-C}$ construction

We now show that such construction produces 2-hop cover that satisfies the node-separation property. Consider any two nodes a and d in the input graph. At each iteration, the algorithm guarantees that a and d is connected in the graph if and only if center nodes of a and d (thus far) do not separate a and d . At a particular iteration, when a node w is chosen, w will be added as a center node of its ancestors and descendants. This implies that *all* paths through w have been covered, and hence the deletion of w does not affect the reachability information of the remaining nodes. The construction terminates when all alternative paths have been covered.

Construction of $u2\text{-hop-C}$. The construction algorithm greedily finds a node v having the densest connectivity in the graph G' , i.e., $(|A(v)| \times |D(v)|) / (|A(v)| + |D(v)|)$ is maximized (Line 03). Then, we add v to L_{out} of $A(v)$ and L_{in} of $D(v)$ (Lines 04-05). Subsequently, all paths from $A(v)$ to $D(v)$ via v are no longer relevant to the construction. Next, the algorithm deletes v and its adjacent edges from the graph G' and proceed to next iteration (Lines 06). The algorithm terminates where there is no edge left in G' (Line 02).

Complexity. Counting the number of nodes in $A(v)$ and $D(v)$ for all v in G can be computed in $O(|V|(|V| + |E|))$. Since there are at most $|V|$ iterations, the overall complexity of the construction is $O(|V|^2 \times (|V| + |E|))$.

Example 6.3: Consider graph G_0 as illustrated in Figure 2.4. In the first iteration,

	Lin	Lout
v_1		v_2, v_3, v_4, v_6
v_2		
v_3		v_4
v_4		
v_5	v_4, v_2	
v_6	v_4	
v_7	v_4, v_6	
v_8	v_3, v_4, v_6	
v_9	v_3, v_4, v_6, v_8	

Figure 6.7: $u2\text{-hop-C}$ of graph G_0

the node having the densest connectivity is v_4 , where $A(v_4) = \{v_1, v_3\}$ and $D(v_4) = \{v_5, \dots, v_9\}$. Next, v_4 is added to L_{out} of $A(v_4)$ and L_{in} of $D(v_2)$. Afterwards, v_4 is removed from the graph and will not be considered in subsequent iterations. In the second iteration, v_6 is chosen and added to L_{in} of v_7, v_8, v_9 and L_{out} of v_1 . Then, v_6 is removed from the graph. v_3 is chosen in the next iteration and is added to L_{out} of v_1 and L_{in} of v_8 and v_9 . At the end of the algorithm, the constructed index is as shown in Figure 6.7.

■

6.1.4 Discussions

It is immediately true that the previous three heuristics generate $u2\text{-hop}$ covers that satisfy the node-separation property. It should also be remarked that all center nodes in the 2-hop cover are selected in a special way such that simple deletions for $u2\text{-hop-A}$, $u2\text{-hop-B}$ and $u2\text{-hop-C}$ become possible: $u2\text{-hop-A}$ ensures that a node w that separates many (uncovered) node pairs are selected earlier than the others. In contrast, when there are multiple alternative paths P from u to v , $u2\text{-hop-B}$ and $u2\text{-hop-C}$ enforce (at least) one node on each alternative path in P is included in both $L_{out}(u)$ and $L_{in}(v)$. A deletion becomes simply the maintenance of the node separation between pairs of nodes.

It is worth remarking that the heuristics derived from the node-separation property may not be as space-efficient as the ones derived from SET COVER, which finds the densest graphs in subgraph in T .

6.2 Hybrid Updatable 2-hop Cover

The three *u2-hops* introduced in the previous sections have different properties. *u2-hop-A* requires the construction of a large number of bipartite graphs as in [CHKZ03] that may be memory-bounded and computationally intensive. In contrast, *u2-hop-B* may result in large 2-hop covers as the bisection can often be relatively large when the input subgraphs is relatively small. The reason is that the minimum bisection of a small graph may often be a large subset of the edges of the graph. Therefore, we propose a hybrid approach of *u2-hops* that takes advantages of both *u2-hops*.

Recall the merging property discussed earlier. We can mix *u2-hop-A*, *u2-hop-B* and *u2-hop-C* and the hybrid of these 2-hop covers still satisfies the node-separation property.

There are many simple alternatives for combining the *u2-hops*. For example, we propose to first use *u2-hop-B* to build 2-hop recursively until *u2-hop-B* becomes inefficient in terms of space. Then, we use *u2-hop-A* to cover the remaining elements in T . The inefficiency of *u2-hop-B* can be estimated as follows: In the worst case, the size of T is $|V|^2$. Suppose C is the vertex cover of the bisection cut of G_1 and G_2 . The size of *u2-hop-B* of a given C , denoted as $|u2-hop-B(C)|$, can be estimated as $|V_1| \times |C| + |V_2| \times |C| + |V_1|^2 + |V_2|^2$. Through experimental studies on the size of T and *u2-hop-A* of random graphs, we can obtain the average size of *u2-hop-A* when compared to T , say $|u2-hop-A| \approx X\% \times |T|$. Hence, we use *u2-hop-B* recursively until $|u2-hop-B(C)| \geq X\% \times |T|$.

Another example is that we can use *u2-hop-B* recursively until the size of the graph is small enough that the graph together with its transitive closure and bipartite graphs can be stored in the main memory. Then, *u2-hop-A* is used.

We can also combine *u2-hop-B* and *-hop-C* in similar manner. This allows *u2-hop-C* to be used for very large graphs which does not fit into the main memory.

All of these approaches aim to reduce the construction time as well as the memory requirements of index construction.

This scenario is similar to [STW05] that recursively partition the XML documents. It has some differences such as 1) we use minimum vertex cover to further reduce the index size; 2) [STW05] builds the 2-hop cover on XML graph, whereas

we collapse the input graph into *DAG* before building the *2-hop* cover; 3) the *2-hop* cover generated by [STW05] does not satisfy *node-separation* property and hence the efficient maintenance algorithm does not apply.

Hybrid of updatable and arbitrary 2-hop labeling. Similarly, a hybrid of updatable and *arbitrary 2-hop* labeling, not necessarily updatable, can be easily defined. As verified with our experiment, this hybrid *2-hop* labeling can be used to reduce the size of some large *u2-hops*.

One scenario is to apply the *u2-hop-A* construction algorithm until the estimated compression ratio of the remaining uncovered elements in T is smaller than certain threshold. Then, the remaining elements are covered by any *2-hop* construction technique.

Since this hybrid *2-hop* labeling does not entirely satisfy the node-separation property, the merging property is not applicable here. However, we can still store and maintain the two *2-hop* labels separately. Note that the query needs to be done on both labels independently.

The extended deletion algorithm presented in Section 5.1 can be modified to support the hybrid *2-hop* labeling.

It should be remarked that determining the hybrid of *2-hop* and *u2-hop* with the minimum size does not seem to be tractable. For example, consider the hybrid of *u2-hop-A* and [CHKZ03].

To avoid having numerous “small” center nodes in the resulting *2-hop* cover, we ignore *u2-hop-A* center nodes that connect only two nodes, where the node-separation property does not offer any advantage. Then, we have the following statement.

Theorem 2.1: *Finding the hybrid of $u2\text{-hop-A}$ and [CHKZ03] with the minimum size is NP-hard.* ■

Theorem 2.1 can be obtained by using the reduction from 3-SAT due to Cohen *et al.* [CHKZ03]. The graph obtained from a 3-SAT instance is dense where an empty *u2-hop-A* is obtained. This graph is not modified and we need to find its minimum *2-hop* cover [CHKZ03], which is NP-hard.

Chapter 7

Experimental Results

We now evaluate the performance of our proposed techniques with a set of experiments. Our experimental evaluation focuses on the effects of the graph size and graph density on the index size and update performance of various versions of *2-hop* labels. In addition, we evaluate the effectiveness of the proposed updatable *2-hop* labeling on both synthetic and real graphs.

We used the *2-hop* labeling of [STW05] (denoted as SC-II) and [CYL⁺06] (denoted as SC-I) as well as their respective deletion algorithm implemented by [BCCY]. *u2-hop-A*, *u2-hop-B* and the hybrid of the two are denoted as UH-A, UH-B and UH-B-A, respectively. *u2-hop-C* is denoted as UH-C. We tried UH-B-A on a large number of random graphs and selected a constant X for switching between UH-B and UH-A. The hybrid of *u2-hop-A* and SC-I is H-A-SC. For H-A-SC, we switch from UH-A to SC-I when UH-A does not offer more than 10% compression. All these labelings have been implemented in C++. The experiments were run on a system with a 3.4GHz Pentium processor with 3G bytes of RAM running Windows XP operating system.

We ran our experiment on both synthetic directed acyclic graphs randomly generated by [JK91] as well as real-world graphs obtained from [BM06].

The synthetic data set consists of random directed acyclic graphs generated by [JK91]. The graph generator takes as inputs N , the number of nodes and E , the number of edges. It generates a random topological order of the N nodes in the graph. Next, a random edge is produced with a restriction that the edge does not

Figure 7.1: (a) Index size by varying the graph size; (b) Construction time by varying the graph size;

violate the topological order. This step is repeated until E edges are produced. Throughout the experiments, different values of N and E are used. The graph generator[JK91] is used in the experiments because 1) it generates random graph with uniform distribution 2) it takes number of nodes and number of edges as inputs which are useful for scalability testing.

Real-world graphs are also used in this experiments to provide an insight of the performance of the algorithms on real-world scenarios. While, there are different kinds of real-world graphs, we randomly pick some of them from the available sources with some restrictions: the graph must not be too small (less than 100 nodes) or too large (more than 10000 nodes) such that all of the algorithms can be tested and we can still distinguish their performance.

7.1 2-hop labeling construction

The first experiment studies the characteristics of the 2-hop labeling. We set the density of the graphs to 2 and varied the size of the synthetic graphs. The size of the 2-hop cover of the graphs are reported in Figure 7.1 (a). It shows that UH-B is the largest in size and its hybrid with UH-A, *i.e.*, UH-B-A, consistently

Figure 7.2: (a) Synthetic graphs used in the update experiment; (b) Construction time of synthetic graphs

reduces the size of the *2-hop* covers. The reason for large UH-B and UH-B-A is that the bisections are relatively large as the subgraphs being processed are getting smaller. UH-A, HA-SC and UH-C produce smaller *2-hop* covers when compared to previous works SC-I and SC-II. UH-C is the smallest in size.

The runtime of the *2-hop* label constructions are reported in Figure 7.1 (b). The construction of UH-A and H-A-SC are significantly slower than the others. The reason is that *u2-hop-A* requires to construct and maintain many bipartite graphs. The construction of UH-B is often slower than that of SC-I and SC-II but faster than that of UH-A and H-A-SC. This is due to the computation of cut vertices in a graph. The runtime of our proposed construction algorithms are more sensitive to the number of vertices in a graph than SC-I and SC-II.

Next, we set $|V| = 8000$ and varied the density of the synthetic graphs by varying the number of edges $|E|$. The results are presented in Figure 7.2 (a). We note that the size of our updatable *2-hops* are more sensitive to the graph density than SC-I and SC-II. This is because the cut vertices and bisections separate relatively smaller subgraphs in a dense graph. Subsequently, more center nodes are needed to record all reachability information. H-A-SC remains efficient because, when only few cut vertices are found, it switches to SC-I. We remark that the relative

Figure 7.3: (a) Deletion performance on synthetic graphs; (b) Change in index sizes after deletions (synthetic graphs)

index size of our updatable *2-hops* is roughly the same as the one presented in Figure 7.1 (a). The size of the smallest updatable *2-hop* is comparable to that of SC-I and SC-II. As the graph density increases (≥ 5), the size of UH-C becomes larger than that of SC-I and SC-II.

The construction time of these synthetic graphs are presented in Figure 7.2 (b). As we increase the density of the graphs, SC-I and SC-II increase more significantly than the other methods. This illustrates that SC-I and SC-II use more computation on minimizing the size of the *2-hop* covers. However, note that UH-C also produces small *2-hop* covers with small size and has an efficient construction.

7.2 Deletion performance

The next experiment verifies the efficiency of Algorithm `delete`. The deletion algorithm for H-A-SC is the extended version of Algorithm `delete` as described in Section 5. We used the synthetic graphs and *2-hop* covers presented in Figure 7.2 (a). We generated a deletion sequence consisting of 100 random node deletions and applied this workload to these *2-hop* covers. The deletion times are reported in Figure 7.3 (a). Note that the *y*-axis is given in logarithmic scale.

The result shows that the updatable *2-hops* clearly outperform SC-I and SC-II. For instance, updatable *2-hops* are faster than SC-I and SC-II by two and three orders of magnitude, respectively, when $|E| = 48000$. The main reason is that SC-I and SC-II invoke *2-hop* construction. Among the updatable *2-hops*, the deletion performance is proportional to the size of the *2-hop* cover. This is because all updatable *2-hops* use Algorithm `delete`. For instance, UH-C is significantly smaller than the others and the deletion of UH-C outperforms the others. Consider H-A-SC. Although the extended algorithm is required, which requires more computation than Algorithm `delete`, H-A-SC outperforms UH-A, UH-B and UH-B-A. The reason is that the size of H-A-SC is clearly smaller than that of UH-A, UH-B and UH-B-A.

A counter-intuitive fact about deletions is that previous deletion algorithms do not always reduce the size of the index. We report the change of the index size due to the deletion workload in Figure 7.3 (b). Since the *2-hop* construction is called with the affected subgraph in the deletions of SC-I and SC-II, the index size often increases after deletions. We find the increase in the index size is proportional to the index size. In comparison, deletions of UH-A, UH-B, UH-B-A, H-A-SC and UH-C always return a smaller *2-hop* cover. The decrease in the index size of UH-A, UH-B, UH-B-A, H-A-SC and UH-C are proportional to the index size.

7.3 Experiments with real world graphs

Table 7.1: Real graphs used in the experiment

Graph	$ V $	$ E $	$ TC $
YEASTL	2361	6646	489039
GRIFFITH	1022	4892	73479
SCIENTOMETRICS	3070	10308	297809

We have tested the deletion performance on three real-world graphs. From our experiment, we find that the relative performance of updatable *2-hops* is consistent to the one presented in the previous experiment. Since we find UH-C is both space- and update-efficient, we only present the comparison between UH-C and previous *2-hops*. The size of the real graphs and their index are reported in Table 7.1 and Figure 7.4 (a), respectively.

Figure 7.4: (a) Real graphs used in the update experiment; (b) Deletion experiment on real graphs

As before, we chose 100 random nodes and sequentially deleted these nodes from the graph. The time taken to perform this workload is presented in Figure 7.4 (b). The result shows UH-C outperforms SC-I by 10, 14 and 9 times on YEASTL, GRIFFITH and SCIENTOMETRICS, respectively, while UH-C outperforms SC-II by 160, 24 and 29 times on these graphs. The change of the index size after deletion is presented in Figure 7.5. The index size of SC-I and SC-II increase after deletion. For UH-C, the index size is always smaller after deletion.

7.4 Insertion performance

The next experiment verifies the insertion performance of UH-A, UH-B, UH-B-A, H-A-SC and UH-C. Insertion of SC-II is briefly discussed in [STW05] and that of SC-I is not discussed in [CYL⁺06]. Thus, in this experiment, we skipped SC-I and SC-II. We used a random graph with $|V| = 4000$ and a density 4 for this experiment. The newly inserted graph has 1000 nodes and 4000 edges. Then, we ranged the number of crossing edges, that connect the new graph to the existing graph, from 5 to 100. The insertion times are roughly 4 seconds for all workloads and all 2-hops. This is because these 2-hop labels use the same insertion algorithm. We

Figure 7.5: Change in index sizes after deletions (real graphs)

Figure 7.6: Insertion performance on synthetic graphs

then study how the number of crossing edges in the insertion workload affects the size of *2-hops*. We present the results in Figure 7.6. Since the number of nodes in the new graph is fixed, we breakdown the index size of the new graph and the index size due to the crossing edges. Figure 7.6 shows that the index size increases gradually with the number of crossing edges.

Figure 7.7: (a) Determining the threshold for H-A-SC; (b) Query performance of the 2-hop of synthetic graphs

7.5 Supplementary experiments

Next, we present a supplementary experiment that illustrates (i) the impact of the threshold for switching from UH-A to SC-I in the construction of H-A-SC and (ii) the query performance of the proposed 2-hop labeling. First, we fixed the graph: $|V| = 8000$ and $|E| = 40000$. We tested the size of H-A-SC with various compression threshold. We switch to SC-I when UH-A does not offer $x\%$ compression. The result is reported in Figure 7.7 (a). It shows that the index size decreases gradually as x increases. Second, to verify the query performance of $u2$ -hop, we generate 1,000,000 random reachability tests on the 2-hop covers shown in Figure 7.2 (a). The results are reported in Figure 7.7 (b). As expected, the query performance is roughly proportional to the index size. Note that the time difference between the 2-hops is almost always smaller than 2 seconds for 1,000,000 reachability tests. In practice, we expect the query performance difference between 2-hops is insignificant.

7.6 Summary

We end this chapter by presenting a summary of the findings of our experimental evaluation:

- In general, UH-C has smaller index size as well as faster construction time compared to other updatable 2-hops: UH-A, UH-B, UH-B-A and H-A-SC.
- For sparse graph (density < 5), the size of UH-C is comparable to the existing 2-hops: SC-I and SC-II whereas for denser graph, the size of UH-C is larger than SC-I and SC-II.
- The construction time of UH-C is comparable to that of SC-I and SC-II. As the density increases, the construction time of SC-I and SC-II increase more significantly than other methods.
- The deletion performance of UH-C is faster than SC-I and SC-II by two and three order of magnitudes, respectively.
- After the deletion, the index size of UH-C decreases, whereas the index size of SC-I and SC-II increases.
- The insertion performance of updatable 2-hops as well as existing approaches are comparable.
- The performance on real-world graphs is consistent with the performance on synthetic data set.
- The query performance difference between 2-hops are insignificant.

We conclude that updatable 2-hops proposed in this work 1) is more update efficient compared to existing approach and 2) has comparable index size, construction time as well as query performance.

Chapter 8

Conclusions and Future Work

8.1 Summary of work

In this thesis, the problem of incremental maintenance of *2-hop* labeling graph indexing scheme has been discussed thoroughly. We have presented a survey of the works related to graph indexing and more specifically *2-hop* labeling indexing scheme. Next, we performed analyzed the deletion of *2-hop* labeling and proposed the node-separation property which leads to efficient incremental maintenance algorithms. We have presented novel incremental maintenance algorithms for our *2-hop* labeling in detail. We have proposed three heuristics which are based on cut vertex, minimum bisection and node deletion, for *2-hop* label construction. Extensive experiments have been conducted to show the characteristics of various versions of *2-hop* labeling. The results show that (i) the incremental maintenance algorithm can be two orders of magnitude faster than previous methods and (ii) the size of our updatable *2-hops* can be comparable to previous *2-hops*.

8.2 Comparison with related works

Compared to other graph indexing methods discussed in Chapter 3, *2-hop* labeling indexing scheme has a relatively smaller index size. Other works[CYL⁺06, CYL⁺08, STW04] have proposed variety of optimizations on *2-hop* labeling index construction. However, the incremental maintenance of the index did not

receive much attention. To the best of our knowledge, incremental maintenance of *2-hop* labeling has only been discussed in [STW05] and [BCCY].

The incremental maintenance approach in both works are similar: they require a reconstruction of *2-hop* labels as well as transitive closure of the subgraphs affected by the update. In the worst case, the affected subgraph could be as large as the entire graph. Hence, upon node/edge deletion, in the worst case, the entire *2-hop* cover needs to be recomputed.

On the contrary, this work proposes novel incremental maintenance algorithm which does not require reconstruction of transitive closure. In addition, only the labels of the ancestors and descendants of the deleted node is affected.

As shown, in experimental evaluation, our incremental maintenance algorithms can be two orders of magnitude faster than previous methods and the size of our updatable *2-hops* are comparable.

However, we note that some of our construction algorithms *u2-hop-A* and *u2-hop-B* are not as scalable as the existing approaches.

8.3 Contribution of this work

We summarize the contributions of this thesis:

- We present a survey on the various works on graph indexing and more specifically *2-hop* labeling scheme including the original algorithm proposed by Cohen[CHKZ03].
- We perform a thorough analysis of the affected elements of the deletion of a node and determine the bottleneck of deletions in *2-hop* labeling.
- We define a *node-separation* property of *2-hop* labels. When *2-hop* labeling satisfies the node-separation property, the update of *2-hop* labeling can be simplified.
- We propose novel incremental maintenance algorithms of our *2-hop* labeling which includes edge and node deletion as well as edge and subgraph insertion. We show how to extend the algorithms to work on arbitrary *2-hop* labels with some increase in cost.

- We present three heuristic *2-hop* cover construction algorithms derived from graph connectivity and analyze their complexities. We also present hybrids of the three heuristics.
- We conducted extensive experiments on updates of various versions of our *2-hop* labeling to verify the effectiveness of our heuristics and illustrate their performance characteristics.

This work complements and improves *2-hop* labeling indexing scheme with efficient incremental maintenance algorithms which is not discussed in original proposal of the algorithm[CHKZ03]. This work eliminates a plausible weakness of *2-hop* labeling indexing scheme which did not support efficient update.

This work could be readily used in application where as *2-hop* labeling is used as the graph indexing method. In particular, the proposed incremental maintenance algorithms could be used to replace the existing approaches. This work could also be used in applications that require updatable graph indexing method.

8.4 Limitation and future works

We notice that our approach has some limitations. In the following, we discuss the limitation of our approaches as well as possible future works.

- Improve *2-hop-A* construction. We notice that the *u2-hop-A* construction is computationally intensive both in time and memory requirements. As future work, we shall investigate how scalable construction algorithms[CYL⁺06] may be applied.
- Improve *2-hop-B* construction. We realize that the size of *u2-hop-B* is relatively large. We shall investigate approaches to reduce the index size. One possible approach is to use *k*-way partitioning instead of the current recursive 2-way partition.
- Improve performance on dense graph. We notice that the performance of our proposed technique might degrade as the graph gets denser. While dense graph is not very common in practice, but it is beneficial to investigate this case further.

- We remark that the three heuristics provided have no theoretical guarantee in terms of index size it produces. While finding *2-hop* label satisfying node-separation property is untractable. As future work, we shall explore a construction algorithm with a theoretical guarantee in terms on index size (with respect to the minimum size).
- Throughout this thesis, we assume that the construction process can be performed in main memory. However, in some cases, as the input graph gets larger, this might not be possible. We shall explore a possibility of external memory *2-hop* labeling construction. One obvious approach is to use divide and conquer. That is, the graph is divided into k subgraphs such that each subgraph and its transitive closure fits into main memory. The construction algorithm is applied to each subgraph and they are merged together to form the final *2-hop* cover.

Bibliography

- [ABB⁺00] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nat Genet*, 25(1):25–29, May 2000.
- [ABJ89] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 253–262, New York, NY, USA, 1989. ACM.
- [AKM01] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact labeling schemes for ancestor queries. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 547–556, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.
- [BCCY] Ramadhana Bramandia, Jiefeng Cheng, Byron Choi, and Jeffrey Xu Yu. Updating recursive XML views without transitive closure. (Technical report at <http://www.ntu.edu.sg/home/kkchoi/techreport.pdf>).
- [BCF⁺02] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Rajeev Rastogi, Shihui Zheng, and Aoying Zhou. Dtd-directed publishing with

- p>attribute translation grammars. In
- VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*
- , pages 838–849. VLDB Endowment, 2002.
- [BCF04] Philip Bohannon, Byron Choi, and Wenfei Fan. Incremental evaluation of schema-directed xml publishing. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 503–514, New York, NY, USA, 2004. ACM.
- [BM06] Vladimir Batagelj and Andrej Mrvar. Pajek datasets, 2006. <http://vlado.fmf.uni-lj.si/pub/networks/data/>.
- [BYR02] Reuven Bar-Yehuda and Dror Rawitz. Approximating element-weighted vertex deletion problems for the complete k-partite property. *Journal of Algorithms*, 42(1):20–40, 2002.
- [CCFV07] B. Choi, Gao Cong, Wenfei Fan, and S.D. Viglas. Updating recursive xml views of relations. *ICDE '07: Proceedings of the 21st International Conference on Data Engineering*, pages 766–775, 15-20 April 2007.
- [CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.
- [CKM02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic xml trees. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–281, New York, NY, USA, 2002. ACM.
- [CKS⁺00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 646–648, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

- [CYL⁺06] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In *International Conference on Extending Database Technology*, pages 961–979, 2006.
- [CYL⁺08] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 193–204, New York, NY, USA, 2008. ACM.
- [CYT06] Jiefeng Cheng, Jeffrey Xu Yu, and Nan Tang. Fast reachability query processing. In *Database Systems for Advanced Applications*, pages 674–688, 2006.
- [Eur] European Bioinformatics Institute. QuickGo: GO browser. Web interface. <http://www.ebi.ac.uk/ego/>.
- [FKS⁺02] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. Silkroute: A framework for publishing relational data in xml. *ACM Trans. Database Syst.*, 27(4):438–493, 2002.
- [FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.
- [GGT89] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.
- [GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

- [Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [JK91] Richard Johnsonbaugh and Martin Kalin. A graph generation software package. In *Special Interest Group on Computer Science Education*, pages 151–154, 1991.
- [Kar] Karypis lab. *Family of Multilevel Partitioning Algorithms*. <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>.
- [Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [KCVGC⁺05] I.M. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I.T. Paulsen, M. Peralta-Gil, , and P.D. Karp. Ecocyc: A comprehensive database resource for escherichia coli. *Nucleic Acids Research*, 33(D334-D337), 2005.
- [KL70] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. In *Bell Systems Technical Journal*, pages 291–308, 1970.
- [KMS02] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries, 2002.
- [Kn31] D. Knig. Graphs and matrices. *Mat. Lapok*, 38:116–119, 1931.
- [KP94] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. *J. Algorithms*, 17(2):222–236, 1994.
- [Law01] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Publications, 2001.
- [Ley] Michael Ley. The dblp computer science bibliography. <http://www.informatik.uni-trier.de/~ley/db/>.

- [Mik] Gerome Miklau. UW XML repository.
<http://www.cs.washington.edu/research/xmldatasets/>.
- [oMA] University of Massachusetts Amherst. Proximity dblp database.
<http://kdl.cs.umass.edu/data/dblp/dblp-info.html>.
- [Pee03] René Peeters. The maximum edge biclique problem is np-complete. *Discrete Appl. Math.*, 131(3):651–654, 2003.
- [RL01] Ronald L. Rivest and Charles E. Leiserson. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.
- [SRS⁺08] Walter Salzburger, Susan CP Renn, Dirk Steinke, Ingo Braasch, Hans A Hofmann, and Axel Meyer. Annotation of expressed sequence tags for the east african cichlid fish *astatotilapia burtoni* and evolutionary analyses of cichlid orfs. In *BMC Genomics*, 2008.
- [STW03] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Ontology-enabled xml search. In *Intelligent Search on XML Data*, pages 119–131, 2003.
- [STW04] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *International Conference on Extending Database Technology*, pages 237–255, 2004.
- [STW05] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 360–371, Washington, DC, USA, 2005. IEEE Computer Society.
- [Sus97] Susan Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *Proceedings of the International Conference of Data Engineering*, pages 55–65, 1997.

- [SWK⁺02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *In VLDB*, pages 974–985, 2002.
- [Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [TW02] Anja Theobald and Gerhard Weikum. The index-based xml search engine for querying xml data with relevance ranking. In *In EDBT*, pages 477–495, 2002.
- [W3Ca] W3C. OWL web ontology language overview. <http://www.w3.org/TR/owl-features>.
- [W3Cb] W3C. Resource description framework (RDF). <http://www.w3.org/RDF>.
- [W3Cc] W3C. Semantic web activity. <http://www.w3.org/2001/sw/>.
- [W3Cd] W3C. Sparql query language for rdf. <http://www.w3.org/TR/rdf-sparql-query/>.
- [WHY⁺06] Haixun Wang, Hao He², Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 75, Washington, DC, USA, 2006. IEEE Computer Society.
- [WLH04] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 66, Washington, DC, USA, 2004. IEEE Computer Society.
- [WZLL06] Gang Wu, Kuo Zhang, Can Liu, and Juan-Zi Li. Adapting prime number labeling scheme for directed acyclic graphs. In *Database Systems for Advanced Applications*, pages 787–796, 2006.

- [ZND⁺01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 425–436, New York, NY, USA, 2001. ACM.

Appendix A

Tarjan algorithm for collapsing strongly connected component

This appendix describes Tarjan algorithm[Tar72] to collapse each strongly connected component in a directed graph into a super node. The algorithm would produce the equivalent directed acyclic graph (DAG) of the input graph.

The pseudocode of the algorithm is given in Figure A.1. The algorithm takes a directed graph G as an input and produce an array of integer *sccindex*, such that for each pair of node u and node v within the same connected component, $sccindex[u]=sccindex[v]$, e.g., $sccindex[u]$ is the index of the strongly connected component (SCC) of u . With the help of *sccindex*, it is easy to construct the equivalent DAG where all nodes in a strongly connected component are collapsed into a super node.

The algorithm performs some initialization (Line 01-04) and subsequently calls `doTarjan` on each unprocessed node. The meaning of each variable is described below.

- *index* represents the number of nodes that have been visited so far
- S represents a stack containing the nodes in the current traversal
- $flag[u]$ is true IFF u is currently in the stack S
- $visit[u] = i$ means u is the i -th node discovered by the traversal (e.g., the depth search index)

- $\text{lowest}[u] = j$ means that j is the smallest value of $\text{visit}[v]$ where v is reachable from u up to the current traversal

Consider a traversal of node u in Sub-Procedure `doTarjan`. Initially, $\text{visit}[u]$ and $\text{lowest}[u]$ are initialized (Line 02-03). Next, u is pushed on to the stack and the flag is updated (Line 04-05). Next, it performs traversal of each of the children and if necessary, $\text{lowest}[u]$ is updated (Line 06-11). After finish traversing all the children of u , $\text{lowest}[u]$ would contain the smallest depth search index of the node reachable from u . Hence, u is the root of the strongly connected component, IFF $\text{lowest}[u] = \text{visit}[u]$ (Line 12). If this is true, we set the `sccindex` of all nodes in the strongly connected component such that u is the representative node. The nodes in the strongly connected component are those located in the stack S after node u (Line 13-15).

Complexity. The complexity of the algorithm is $O(n + m)$ as each edge is at most visited once on the entire execution of the algorithm.

```

Procedure tarjan
Input: a directed graph  $G: (V, E)$ 
Output: an array sccindex; sccindex[i] is the SCC node id of i
01 index = 0
02 initialize flag of all nodes to false
03 initialize stack S to empty
04 initialize visit of all nodes to 0
05 for each node u in V
06   if (visit[u] == 0)
07     doTarjan(u)

Sub-Procedure doTarjan
Input: a starting node u in G
01 index = index + 1
02 visit[u] = index
03 lowest[u] = index
04 flag[u] = true
05 push u to S
06 for each edge (u,v)
07   if (visit[v] == 0)
08     doTarjan(v)
09   lowest[u] = min(lowest[u], lowest[v])
10   else if flag[v] is true
11     lowest[u] = min(lowest[u], visit[v])
12 if (visit[u] == lowest[u])
13   v = null
13   while S is not empty and v != u
14     v = S.pop
15     sccindex[v] = visit[u]

```

Figure A.1: Tarjan algorithm for collapsing strongly connected component

Appendix B

Approximation algorithm for finding largest biclique

We outline the approximation algorithm used to solve an instance of MSEBP (see Section 6.1.1). The algorithm was originally proposed in [BYR02].

The algorithm is described in Figure B.2. The input of the algorithm is the biclique $B(A, D, E)$ and the weight function w . $w(v)$ denotes the weight of node v and $w(e)$ denotes the weight of edge e . $\bar{w}$ denotes the sum of weight of node v and its incident edges, e.g., $\bar{w}(v) = w(v) + \sum_{e \in E(v)} w(e)$, where $E(v)$ denotes the edges incident to v . A positive node v is a node with $\bar{w}(v) > 0$.

A pair of conflicting nodes (u, v) are a pair of nodes, where u in A and d in D such that there is no edge (u, v) in E . Obviously, in a solution of MSEBP, either u , v or both must be removed from B . A pair of positively conflicting nodes are a pair of conflicting nodes (u, v) where u and v are positive.

Procedure SubstractPair

Input: a pair of nodes u and v , weight function w

Output: weight function w is updated

01 $\Delta = \min(\bar{w}(u), \bar{w}(v))$

02 Substract(u, Δ)

03 Substract(v, Δ)

Figure B.1: Sub-procedure SubstractPair

```

Procedure msebp-approximation
Input: a bipartite graph  $B(A, D, E)$ 
Output: a set of nodes  $X$  to be removed
01 initialize the doubly linked list with positive nodes
02 while Head  $\neq$  NULL
03    $u = \mathbf{Head}$ 
04   foreach  $v$  in  $E(u)$ 
05     MTF ( $v$ )
06   while  $\overline{w}(u) > 0$  AND Next( $u$ )  $\neq$  NULL
07      $v = \mathbf{Next}(u)$ 
08     SubtractPair( $u, v$ )
09     if ( $\overline{w}(v)=0$ ) then Delete( $v$ )
10   Delete( $u$ )
11 Return  $u | \overline{w}(u) = 0$ 

```

Figure B.2: An approximation algorithm for MSEBP

The idea of the algorithm is to repetitively invoke **SubtractPair** on a pair of positively conflicting nodes from the graph. The algorithm terminates when there is no positively conflicting pair of nodes.

Procedure **SubtractPair** is shown in Figure B.1. The procedure takes a conflicting pair of nodes u and v as an input. It reduces $\overline{w}(u)$ and $\overline{w}(v)$ by the minimum between the two values (Line 01-03).

Procedure **Subtract**(u, Δ) which is called by procedure **SubtractPair** performs the following. It reduces the weight of u and its incident edges up to Δ (negative weight is not allowed). The choice of incident edges are arbitrary. Subsequently, the value of $\overline{w}(u)$ is also reduced by up to Δ .

The algorithm utilizes a doubly linked-list to store all the nodes. The linked-list is accessed using the following operations.

- **Head**: retrieve the head of the linked-list
- **Next**(u): retrieve the next element in the linked-list after u
- **Delete**(u): delete u from the linked-list
- **MTF**(u): if u is in the list, then move u to the front of the list

The algorithm starts by placing all the positive nodes into the doubly linked-list (Line 01). Next, it traverse the list starting from the head (Line 02-03). For each node u , it moves all of its neighbors to the front of the list (Line 04-05). Hence, all the non conflicting nodes are in front of u in the list. Next, while $\bar{w}(u) > 0$, we invoke `SubtractPair` with for each of the remaining elements in the list and we remove the remaining elements if the value of $\bar{w}$ drops to zero (Line 06-09). u is subsequently removed from the list (Line 10). At the end of the algorithm, the nodes to be removed are those with $\bar{w}$ equals to zero (Line 11).

In order to produce larger biclique, a minor modification is made to the algorithm. That is, we expand the biclique as much as possible by iteratively adding a node from the bipartite graph to the biclique until there is no more node to be added. The node chosen must be incident to all existing nodes in the biclique. The ordering of nodes to be added are arbitrary. This modification ensures that the biclique generated is maximal.

Complexity. Single invocation of MTF in Line 05 takes constant time as we already maintain a pointer to the node v . The amortized complexity of single call of `SubtractPair` is constant. There are at most $n - 1$ invocations of `SubtractPair` and $O(n + m)$ invocations of MTF. Hence, the total complexity of the algorithm is $O(n + m)$.

Appendix C

Approximation algorithm for finding minimum bisection

The software package [Kar] that we use to obtain small bisection of the input graph in the construction of *u2-hop-B* is based on Fiducia-Mattheyses heuristic [FM82] presented on Figure C.1. The presented algorithm is an improvement on classic algorithm on bisection [KL70].

The algorithm computes an arbitrary initial partition G_A and G_D . This can be obtained by randomized algorithm or by other heuristic. Next, each node v is associated with a score function S_v . The score function indicates the gain achieved when the node is swapped. It is initialized as the difference between the number of adjacent nodes that are in the different partition and the number of adjacent nodes that are in the same partition (Line 03-09).

Next, it performs n iterations of node pair swapping. At each iteration, nodes with highest score from each partition are chosen (Line 11-12). This can be implemented in amortized constant time by maintaining an array indexed with the value of S_a and S_d and maintaining a pointer to the current largest value of S_a and S_d . Next, the gain of the swap is computed and the best partition so far is maintained (Line 13-15). Next, we remove the chosen nodes from candidates of subsequent selections (Line 16-17). Finally, the score of each node is updated (Line 18-27). For each x , neighbor of a , if x is in the same partition as a (before the swap), we increase its score by 2; otherwise, we decrease its score by 2 (Line 18-22). Neighbors of d can be processed in similar manner (Line 23-27).

After the n iterations have been performed, the best solution so far is returned

Procedure Fiducia-Mattheyses algorithm

Input: an undirected graph $G(V, E)$

Output: a bisection of G into G_A and G_D

- 01 Compute initial partition G_A and G_D with cost C
- 02 Initialize S_v to 0 for all $v \in V$
- 03 for each edge (u, v) in E
- 04 if u and v are in the same partition
- 05 $S_u = S_u - 1$
- 06 $S_v = S_v - 1$
- 07 else
- 08 $S_u = S_u + 1$
- 09 $S_v = S_v + 1$
- 10 for i from 1 to n
- 11 Compute $a \in G_A$ such that S_a is maximized
- 12 Compute $d \in G_D$ such that S_d is maximized
- 13 $\delta = S_a + S_d - 2$
- 14 $C = C - \delta$
- 15 if C is the best C so far, save G_A and G_D accordingly
- 16 $G_A = G_A - a$
- 17 $G_D = G_D - d$
- 18 for each edge (a, x) in E
- 19 if $x \in G_A$
- 20 $S_x = S_x + 2$
- 21 else
- 22 $S_x = S_x - 2$
- 23 for each edge (d, x) in E
- 24 if $x \in G_D$
- 25 $S_x = S_x + 2$
- 26 else
- 27 $S_x = S_x - 2$
- 28 return the best solution so far

Figure C.1: A heuristic to obtain minimum bisection

(Line 28).

Complexity. The overall complexity of the algorithm is $O(n + m)$. This is based on following observation. It is clear that the overall contribution of Line 02-09 and Line 18-27 are $O(n + m)$. Based on previous discussion, Line 11-12 can be implemented in constant time. It is obvious that the remaining lines can be implemented in constant time.

Appendix D

List of Publication

During the two years of master degree, the student has published the following manuscripts:

- *On incremental maintenance of 2-hop labeling of graphs*
Ramadhana Bramandia, Byron Choi, Wee Keong Ng
Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008
- *The incremental maintenance of 2-hop labeling of large graphs*
Ramadhana Bramandia, Byron Choi, Wee Keong Ng
IEEE Transactions on Knowledge and Data Engineering (submitted for publication)