Scalable Mining of Large Graphs and Its Applications

BY

YAN XIE
B.S., Hangzhou Institute of Electronics, China, 2003
M.S., Zhejiang University, China, 2006
M.S., University of Illinois at Chicago, 2008

THESIS

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:
Philip S. Yu, Chair and Advisor
Clement Yu
Bing Liu
John Lillis
Jing Wang, Mathematics, Statistics, and Computer Science
ACKNOWLEDGMENTS

My first thanks go to my Ph.D. advisor, Professor Philip S. Yu, for his continuous guidance, mentor and encouragement during my graduate studies in the University of Illinois at Chicago. Were it not for him, this dissertation would not have been possible. He led me explore in this interesting field, and his navigation taught me how to sense important research problems, how to approach them by creative conceptual brainstorming and practical mathematic solution. Also, it was my honor to collaborate with Dr. Charu Aggarwal from IBM Research. It was always inspiring and exciting to work with him.

I would like to thank Professor Clement Yu, Professor Bing Liu, Professor John Lillis and Professor Jing Wang for taking their valuable time to serve on my dissertation committee and for their constructive suggestions and feedbacks. Special thanks to Professor Clement Yu who continued to help me even when he was sick.

I also want to thank many colleagues and friends I met and worked with in the lab and in UIC. We had thorough technical discussions and enjoyable moments in life throughout the years. I will always remember the time I spent here, which were all fantastic memories.

Finally, I am deeply indebted to my parents, my family and those who love me for their love, trust, and encouragement. This dissertation is dedicated to my parents whose unconditional love and endless support enables me to purse my study oversea. I do hope that I have more time to be around them in the future.

YX


### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Dissertation Framework</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Scalable Mining of Large Graphs</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Mining Transformed Graphs by Summarization</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Indexing Applications</td>
<td>7</td>
</tr>
<tr>
<td>2 SCALABLE MINING OF LARGE GRAPHS</td>
<td>10</td>
</tr>
<tr>
<td>2.1 GConnect: The Connectivity Index</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1 Creating the Indexed Representation from Compressed Graphs</td>
<td>19</td>
</tr>
<tr>
<td>2.1.2 Query Processing from the Connectivity Index</td>
<td>20</td>
</tr>
<tr>
<td>2.1.3 Creating the Compressed Representation</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Analysis of Algorithmic Effectiveness</td>
<td>25</td>
</tr>
<tr>
<td>2.2.1 Further Optimizations</td>
<td>30</td>
</tr>
<tr>
<td>2.3 Experimental Results</td>
<td>30</td>
</tr>
<tr>
<td>2.3.1 Accuracy Analysis</td>
<td>32</td>
</tr>
<tr>
<td>2.3.2 Computational Efficiency</td>
<td>36</td>
</tr>
<tr>
<td>2.3.3 Sensitivity Analysis</td>
<td>38</td>
</tr>
<tr>
<td>3 MINING TRANSFORMED GRAPHS BY SUMMARIZATION</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Preliminaries</td>
<td>45</td>
</tr>
<tr>
<td>3.1.1 Problem Definition</td>
<td>45</td>
</tr>
<tr>
<td>3.1.2 Pattern Graph Construction</td>
<td>46</td>
</tr>
<tr>
<td>3.1.3 Problem Extension</td>
<td>50</td>
</tr>
<tr>
<td>3.2 Top-Down Graph-based Frequent Pattern Mining Framework</td>
<td>50</td>
</tr>
<tr>
<td>3.3 Clique Detection</td>
<td>52</td>
</tr>
<tr>
<td>3.3.1 The choice of s</td>
<td>54</td>
</tr>
<tr>
<td>3.4 Candidate Refinement</td>
<td>54</td>
</tr>
<tr>
<td>3.4.1 Item Removal</td>
<td>56</td>
</tr>
<tr>
<td>3.4.2 Candidate Extension</td>
<td>59</td>
</tr>
<tr>
<td>3.5 Experimental Results</td>
<td>59</td>
</tr>
<tr>
<td>3.5.1 Efficiency Analysis</td>
<td>61</td>
</tr>
<tr>
<td>3.5.2 Quality Measurement</td>
<td>63</td>
</tr>
<tr>
<td>3.5.3 Parameter Sensitivity Analysis</td>
<td>64</td>
</tr>
<tr>
<td>3.6 Related Work</td>
<td>66</td>
</tr>
<tr>
<td>4 INDEXING APPLICATIONS</td>
<td>68</td>
</tr>
<tr>
<td>4.1 Preliminaries</td>
<td>70</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>4.2 Graph Containment Search</td>
</tr>
<tr>
<td>4.2.1</td>
<td>4.2.1 Contact Preservation</td>
</tr>
<tr>
<td>4.2.2</td>
<td>4.2.2 Feature Generation and Selection</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>4.2.2.1 Dual Feature Generation</td>
</tr>
<tr>
<td>4.2.2.2</td>
<td>4.2.2.2 Feature Selection by Size-Increasing Bootstrapping</td>
</tr>
<tr>
<td>4.2.3</td>
<td>4.2.3 Empirical Studies of CP-Index</td>
</tr>
<tr>
<td>4.2.3.1</td>
<td>4.2.3.1 Results on Real Data</td>
</tr>
<tr>
<td>4.2.3.2</td>
<td>4.2.3.2 More Results on Synthetic Data</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3 Graph Location Search</td>
</tr>
<tr>
<td>4.3.1</td>
<td>4.3.1 Feature-Based Indexing</td>
</tr>
<tr>
<td>4.3.1.1</td>
<td>4.3.1.1 Graph Containment Search</td>
</tr>
<tr>
<td>4.3.1.2</td>
<td>4.3.1.2 Graph Location Search</td>
</tr>
<tr>
<td>4.3.2</td>
<td>4.3.2 Storage Efficient Static Index Construction</td>
</tr>
<tr>
<td>4.3.3</td>
<td>4.3.3 Query Answering</td>
</tr>
<tr>
<td>4.3.3.1</td>
<td>4.3.3.1 Query Decomposition</td>
</tr>
<tr>
<td>4.3.4</td>
<td>4.3.4 Experimental Results</td>
</tr>
<tr>
<td>4.4</td>
<td>4.4 Related Work</td>
</tr>
<tr>
<td>5</td>
<td>5 CONCLUSIONS</td>
</tr>
<tr>
<td>5.1</td>
<td>5.1 Summary of Contributions</td>
</tr>
<tr>
<td>5.2</td>
<td>5.2 Future Works</td>
</tr>
</tbody>
</table>

### CITED LITERATURE

- CITED LITERATURE | 132

### VITA

- VITA | 138
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>33</td>
</tr>
<tr>
<td>II</td>
<td>37</td>
</tr>
<tr>
<td>III</td>
<td>39</td>
</tr>
<tr>
<td>IV</td>
<td>48</td>
</tr>
<tr>
<td>V</td>
<td>123</td>
</tr>
</tbody>
</table>

I. SIZE REDUCTION AND OVERALL EFFECTIVENESS
II. INDEX CONSTRUCTION TIME
III. RUNNING TIME COMPARISON
IV. TRANSACTION DATABASE EXAMPLE
V. PERFORMANCE ON INDEXING CONSTRUCTION
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>63</td>
</tr>
<tr>
<td>10</td>
<td>65</td>
</tr>
<tr>
<td>11</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>74</td>
</tr>
<tr>
<td>13</td>
<td>75</td>
</tr>
<tr>
<td>14</td>
<td>77</td>
</tr>
<tr>
<td>15</td>
<td>78</td>
</tr>
<tr>
<td>16</td>
<td>91</td>
</tr>
<tr>
<td>17</td>
<td>92</td>
</tr>
<tr>
<td>18</td>
<td>94</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>95</td>
</tr>
<tr>
<td>20</td>
<td>96</td>
</tr>
<tr>
<td>21</td>
<td>97</td>
</tr>
<tr>
<td>22</td>
<td>98</td>
</tr>
<tr>
<td>23</td>
<td>101</td>
</tr>
<tr>
<td>24</td>
<td>105</td>
</tr>
<tr>
<td>25</td>
<td>106</td>
</tr>
<tr>
<td>26</td>
<td>118</td>
</tr>
<tr>
<td>27</td>
<td>119</td>
</tr>
<tr>
<td>28</td>
<td>120</td>
</tr>
<tr>
<td>29</td>
<td>122</td>
</tr>
<tr>
<td>30</td>
<td>124</td>
</tr>
</tbody>
</table>
SUMMARY

In recent years, as the semantics of real-world applications becomes more and more complex, graph has emerged as a basic data manifestation unit in many domains, where its expressiveness can provide the best mathematical model for advanced use cases that involve a set of individual entities plus their inter-relationships. Responding to this trend, it is critically important to transform state-of-the-art data mining and data management systems so that they are able to handle such data sources with rich interconnecting structures, e.g., those representing XML, circuits, cyber-networks, chemical compounds, biological data, social data and the Web.

One major challenge faced by the end-users of graph data is the large size of the graphs they want to analyze. Accompanying the arrival of the information era, huge repositories are created due to the constant flowing-in of data. People have been putting great emphasis on developing scalable methods to mine and manage large databases. Nonetheless, scalability is even more urgently expected in the graph scenario, since the structural complexity of graphs can often make computations more costly and thus the efficiency of algorithms is a critical concern. In this thesis, we are going to focus on scalable mining of large graphs. We propose a novel graph summarization technique to reduce the size of graphs, so that the computational complexity associated with processing the data is more manageable. In order to guarantee the result accuracy, we further relate patterns existing on the original graphs with those found on the summarized ones, and prove that they are very close to each other if certain conditions are met. This graph summarization concept has proved to be very useful for many concrete data
SUMMARY (Continued)

analysis tasks, even including some cases where the underlying information is not naturally represented as graphs. For these cases, we build graphs to reveal insightful observations of the data from alternative angles, while summarization techniques are applied on the resulted graphs to make the analysis scalable.

With efficient methods that are capable of extracting useful knowledge, one can also utilize the extracted knowledge to help other data management tasks on large graphs, as well. For the latter part of this thesis, we shall examine one application of scalable large graph mining algorithms, namely the indexing problem for graph search. We explore our research on both graph containment search and graph location search. In the first case, people care about a binary containment relationship between a query graph $q$ and a target graph $g$. Focusing on the indexing deficiencies brought up by the large size of graphs, we design and implement a novel indexing scheme called CP-Index, which summarizes graphs for feature extraction and preserves the contact information among features for further pruning opportunities. In the second case where it becomes necessary to locate all identical copies of $q$ in $g$ given graphs with larger size and more complex internal structures, we take advantage of feature-based indexing in containment search to tackle the mining deficiency and pattern recurrence curse. A composite dynamic-and-static indexing (DS-Index) is further proposed, which is proved to be both effective and storage efficient.
CHAPTER 1

INTRODUCTION

Graph is a data structure consisting of vertices and edges. Due to its generality, it is the best mathematical representation for complex data scenarios that comprise multiple objects and their inter-relationships. Recently, with the arrival of the information era, advanced applications generate more and more data with rich structures, making it necessary to use graph as the basic representation unit. For example, in a chemical compound dataset, vertices are individual atoms and edges are chemical bonds; in life sciences, different biological entities such as proteins coordinate with each other in a particular way to achieve certain biological functions; graphs have been used to represent XML, circuits and cyber-networks in computer science; while with the emergence of Internet and users’ on-line interactions, social networks have risen to prominence in the past several years.

Undoubtedly, all above real-world applications call for efficient and effective algorithms to examine graph data, making the mining and analysis of graphs an emerging hot topic in both academia and practice. A number of significant methods have been proposed in this context (5; 56; 54; 58; 59; 62), while a detailed survey can be found in (21).

Nowadays, one key characteristic of the information era is the fast-changing and wide-spread nature of data sources. It generates a vast set of data collected in repositories, which is both an opportunity and a challenge for data analysts. For this reason, scalable algorithms are extremely useful and urgently expected to solve such problems. This is especially true for graphs, due to
the structural complexity. There, since the involved entities are dependent on each other in an intriguing way through their mutual relationships, many graph pattern discovery algorithms incur substantial (sometimes even exponential) calculation costs and thus are extremely hard to be deployed on large datasets.

Inspired by this intrinsic bottleneck, we are going to focus on the scalable mining of large graphs and its related applications in this thesis. For many of the example applications we mentioned above, e.g., protein-protein interaction networks, the graph is very large (thousands of vertices/edges), and there are usually many such graphs in a single dataset. Also, with the ascending of Facebook, Twitter and the like, the size of social networks in the 21st century is without any doubt. More instances of large graph data exist in abundance, e.g., the network flow graph and system call graph that are going to be used as real testing data later, so it is hard to enumerate all of them here. In a word, we will spend time to develop novel techniques for efficiently extracting useful knowledge from large graphs. This will not only reveal insightful hidden patterns to analysts; what is more important, we also hope that applying the obtained knowledge could help various graph data management tasks, as well.

The rest of this chapter proceeds as follows. Since there are multiple aspects about scalable mining of large graphs we want to discuss, in Section 1.1, an overall framework outlining the organization of the thesis is presented first. After that, Sections 1.2, 1.3 and 1.4 will look into each of the aspects and give more detailed discussions.
1.1 **Dissertation Framework**

Data reduction is the key mechanism we want to leverage for tackling the difficulty associated with mining large graphs. Many pattern discovery tasks on graphs have pretty high time complexity, for example, consider mining frequent graph patterns from a database, subgraph isomorphism, the core routine of any mining algorithm, is an NP-hard problem. This means that, if graph size is not reduced, then the task we are facing is essentially intractable. Thus, what we propose is summarizing the graph in question to a much smaller instance, so that the pattern discovery algorithms can be efficiently executed on top of it. We achieve this by decreasing node/edge cardinalities and creating a correspondence between nodes/edges of the summarized graph and those of the original graph. This correspondence would help us infer patterns of the original graph based on patterns found on the summarized one. We admit that there will be some pattern loss because it is true that the summarized graph misses some information; however, as we shall mathematically prove in many cases, such loss is often minimal and could be further alleviated by randomizing the summarization process. This will be the focus of Section 1.2 and Chapter 2.

This graph summarization technology is not limited to applications where data is naturally modeled as graphs, as well. In many cases, manifesting the data as graphs not only offers an alternative way for examination; furthermore, graphical representation may also reveal profound observations from a different but insightful perspective, which could help the data mining tasks in hand. For these applications, we will build graphs to transform the underlying information; however, due to the scale of most problems nowadays, the size of the resulted graphs would not
be small. Facing the computational difficulty, we will again be leveraging the data reduction mechanism mentioned above to work on these large graphs, and this will be the focus of Section 1.3 and Chapter 3.

Successfully obtaining patterns from the data is never the end of story, after that, users want to understand patterns, and more importantly, apply the useful portion of knowledge to help the graph data management system in general. For example, it has been shown in the past that frequent subgraphs mined from a graph database are effective indexing units (57; 13; 46) and topological features (37; 23) for various kinds of graph search. In this thesis, we will use graph indexing as an example and explore the possibility of solving the indexing problem on large graphs. Here, we target at two types of graph search with indexing: one is graph containment search which finds containment relationship between a query and a graph database; while the other is graph location search which locates every embedding of a query in a large network. The large size of the graph can bring up a lot of issues during the traditional indexing procedure such as feature mining deficiency and low selectivity issue. To this extent, we propose two novel indexing schemes to tackle the two search problems respectively. Section 1.4 and Chapter 4 will give more details about this.

1.2 Scalable Mining of Large Graphs

The problem of connectivity is an extremely important one in the context of massive graphs. In many large communication networks, social networks and other graphs, it is desirable to determine the minimum-cut between any pair of nodes. The problem is well solved in the classical literature, since it is related to the maximum-flow problem, which is efficiently solvable. How-
ever, as we mentioned above, large graphs pose new challenges to the calculation of maximum flows, as the big size creates severe scalability problem. In addition, large graphs may often be disk resident, and such graphs cannot be efficiently processed for connectivity queries. This is because the minimum-cut problem is typically solved with the use of a variety of combinatorial and flow-based techniques which require random access to the underlying edges in the graph.

Our solution to this problem is to leverage graph summarization techniques and replace a single large problem, with multiple-smaller approximations which can be efficiently solved. The solutions from the different approximations can be combined in order to create a single robust solution. We will use an edge-sampling based approach to create compressed representations of the underlying graphs. Since these compressed representations can be held in main memory, they can be used to derive efficient approximations for the minimum-cut problem. These compressed representations are then organized into a disk-resident index structure. Since the sampling and compression procedures are randomly executed, we will also provide a theoretical analysis of the effectiveness of such an index. This also allows the index to provide an estimate of the accuracy of the cut-based approach. While the accuracy of the minimum connectivity estimation is different for different pairs of vertices, we will see that the estimation is particularly effective for the case of vertex pairs in which the value of the underlying minimum cuts is low. Such cuts are also the most critical in network connectivity applications, because they could lead to network disconnection by the removal of a few edges. The experimental results confirm that the resulting approach provides between two and three orders of magnitude more efficient query processing than a disk-resident approach at the expense of a small amount of accuracy.
1.3 Mining Transformed Graphs by Summarization

As introduced before, in some cases conventional data can also be modeled as graphs, and surprisingly, this novel graph representation offers us a new perspective of solving traditional data mining tasks. Taking frequent pattern mining as an example, in this part of the thesis, we present a top-down mining mechanism by transforming and summarizing transactional databases into a compact graph representation.

For almost all state-of-the-art algorithms, we note that they share one major disadvantage: It may not be possible for them to mine very long patterns in a large database with a huge set of frequent patterns. This shortcoming is mainly due to the fact that they do a bottom-up search, which leads to the sheer size of an extremely large pattern space to be explored. However, as we observe in many applications, mining tasks in practice usually attach greater importance to patterns with larger size. Thus, in this part of the thesis, we point our research to solve this difficult problem from a different perspective: In the case where the given database is large which contains a huge set of frequent patterns, we focus on mining top-k long maximal frequent patterns because long patterns are in general more interesting ones.

Considering the fact that all small-sized patterns are part of one or more large-sized ones, we explore the possibility of summarizing items and their co-existence relationships by a so-called pattern graph. Different from traditional level-wise mining or tree-growth strategies that operate in a bottom-up way, our method works in a top-down manner: We pull large maximal cliques from a pattern graph constructed after some fast initial processing, and directly use
such large-sized maximal cliques as promising candidates for long frequent patterns; we have a separate refinement stage to further transform these candidates into true maximal patterns.

1.4 Indexing Applications

Graph location search, i.e., finding all graphs in a database $D$ that contain the query graph $q$, is a classical primitive prevalent in various graph database applications. In the past, there has been an abundance of studies devoting to the construction of indices to facilitate searching on graphs, achieving great success as reported in previous literature.

In contrast, as we focus on large graph data in this thesis, we found that the same graph indexing task has been severely hindered in this new scenario. Most of the traditional indexing technology is feature-based and works in two steps, mining frequent subgraphs from the database and then picking a subset of them as indexing features. Our observations show that three major issues are associated with direct indexing using frequent patterns mined from large graphs (say using the summarization technique): low selectivity, inadequacy and redundancy.

Towards these challenges, we apply summarization techniques for efficient indexing of large graphs, and then propose a novel indexing scheme: CP-Index (Contact Preservation Index), which consists of contact preservation filtering, dual feature generation, and size-increasing bootstrapping feature selection to tackle each of them, respectively. The key concept behind our scheme is leveraging contact preservation among features for more effective indexing. Different from traditional schemes, we record every feature’s locations of occurrences in the database graphs off-line and check their relative positioning during querying time to make sure that, if two features touch upon each other in a query (e.g., they share some vertices/edges in some
particular way), then the same contacts are preserved in any true candidate graph in the database. With contact preservation, we alleviate the need to mine larger subgraph features in search for higher selectivity. To account for the potentially large variabilities among graphs because of their big size, a new dual feature generation method is proposed to address the inadequacy issue. In addition to generating shared features common to the whole database, we decide to mine graph-specific features that are particular to one single graph because they offer better query indexing for individual database entries. Furthermore, regarding the potentially explosive size of the subgraph feature collection, we introduce a strategy called size-increasing bootstrapping to select an optimal subset for index construction, which directly uses the CP-Index query processing framework to decide whether it pays off to index an additional feature.

Given the extensive experimental results, CP-Index is proved to work effectively for graph containment search on a database of large graphs. Meanwhile, there are some cases where people care about not only whether but also where identical copies of a query can be located in a given graph. This problem is referred to as graph location search which usually involves graphs with larger size and more complex internal structures. As we have mentioned, frequent fragment of a graph is shown to be effective as indexing feature because it is structurally expressive, and it is also the indexing unit in CP-Index. However, for indexing purpose, previous research works on graph location search only considered direct neighborhood information or compressed structure of shortest paths. This is due to the mining deficiency and pattern recurrence curse on the excessive and complicated internal network structures that prevent index construction and storage from being handled efficiently.
In this thesis, we propose an integrated dynamic-and-static indexing (DS-Index) scheme which can build a best-of-breed feature-based index on a large information network and let it work in the most suitable environment. To achieve this, an *r-enhanced partitioning strategy* is employed to transform the network into a database of small overlapped graphs. Therefore we can efficiently extract indexing features and bypass the storage difficulty of pattern recurrence curse when it is unaffordable to store all possible locations of each feature. Then a corresponding *r-decomposition strategy of queries into features* is further proposed to process queries in a decompose-and-recombine manner so that each feature can be *dynamically indexed on the fly*. The integrated framework will guarantee that recombining all look-up results returns complete query answers on the original network. This key property, i.e., *r-preservation principle of features*, cannot be achieved by brute-force partitioning and querying due to the loss of important structures. We refer to the overall approach as DS (dynamic-and-static) index, as different from traditional offline index construction, the indexing information of query features here is dynamically created on-the-fly by looking up the static database index. Experiments show that DS-Index is very space efficient and effective in the scenario of search on single large graph.
CHAPTER 2

SCALABLE MINING OF LARGE GRAPHS

Managing and mining graph data has seen renewed interest in recent years because of the increased interest in a number of structural applications such as chemical data, biological data, XML data, and computer networks (5; 21; 56; 54; 58; 59; 62). The problem of connectivity is an extremely important one in the context of massive graphs. In many large communication networks, social networks and other graphs, it is desirable to determine the minimum-cut between any pair of nodes. For example, it can be used to determine the number and identity of the minimum number of links which can be used to disconnect the communication network between a given pair of nodes. It can also be used to determine and isolate dense regions in large connected graphs.

The minimum-cut problem has been widely studied in the literature (9), and is well known to be the mathematical dual of the maximum flow problem. A variety of very efficient algorithms exist for solving the maximum flow problem in the memory-resident case. The best algorithms for the problem can solve it in (slightly worse than) $O(n \cdot m)$ time, where $n$ is the number of nodes, and $m$ is the number of edges.

We note that most of the existing techniques (9; 18) are designed with the implicit assumption that the underlying graphs are memory-resident. However, in many application domains such as the web and computer networks, the underlying graphs are so large that they cannot be stored in main memory, but may need to be stored onto disk (27; 38; 39; 45). For example,
a typical communication network may have millions of nodes, and the number of edges are several orders of magnitude greater. The web-graph (38; 39) is even larger, and may require specialized hardware to enable storage of the underlying graph structures. In such scenario for disk-resident graphs, the problem is significantly more difficult. This is because most of the existing algorithms for this problem use combinatorial or flow-based techniques which can access the edges in the underlying graph in arbitrary order. Each operation such as updating the flows or node labels will lead to a random access on disk. This leads to impractical running times for a problem which is easily solvable in the memory-resident case. Furthermore, a user may repeatedly query the graph over different source-sink pairs, and may therefore expect online response times. In general, our goal is to create an index which can provide extremely efficient responses even for very large graphs.

In this chapter, we will study the minimum connectivity problem for massive disk-resident graphs. For a given pair of nodes $s$ and $t$, it is desirable to determine the minimum number of edges required in order to disconnect the graph into two components such one of them contains $s$ and the other contains $t$. This problem is identical to that of finding a minimum-cut between the pair $s$ and $t$.

One possible solution to this problem is to pre-store the minimum-cut between every pair of vertices. However, this is not a practical solution to the problem. This is because for large graphs, the number of possible source-sink pairs may be too large to store the underlying cut values explicitly. For example, for a graph containing $10^6$ nodes, the number of possible node pairs will be over $10^{11}$. The detailed information for the corresponding cuts may require storage
in the tera-byte order. This level of pre-storage may not be available on most platforms today. Furthermore, since the size of many graph data sets encountered in applications continue to increase over time, it is critical to have an approach which scales well with the size of the underlying data set.

In order to achieve this goal, we will design a \textit{disk-based query index for connectivity queries}. We will use a randomized contraction approach in order to create compressed representations of the underlying graph. Repeated probabilistic contractions are used in order to create different compressions of the overall graph. These compressions are used to create an index which provides high-quality responses. The idea behind \textit{repeated storage of probabilistically compressed data} is to ensure that each of the compressed graphs is small enough to handle effectively with the use of memory-resident algorithms. At the same time, the repetition in the storage is leveraged during query processing in order to provide robust responses. We will show that such an approach is extremely effective in creating an index for connectivity queries. At the same time, the approach retains its efficiency, because it can be decomposed into multiple problems of smaller size, rather than one graph of very large size. The savings in solving smaller combinatorial problems are far greater than the overhead of having to solve them multiple times.

This chapter is organized as follows. In section 2.1, we will discuss the overall approach for creating the index for connectivity queries. We will discuss the details for constructing the index and then show how to use it for query resolution. In section 2.2, we will provide an analysis of the approach. The experimental results are discussed in section 2.3.
2.1 GConnect: The Connectivity Index

In this section, we will discuss the method for creating the connectivity index. Before describing the algorithm for creating the index, we will introduce some notations and definitions. We assume that the connectivity queries are designed for a large graph with node set $N$ and edge set $A$. For simplicity of discussion, we assume undirected graphs, though the approach discussed in this chapter can be extended to the case of directed graphs. The number of nodes in the set $N$ is $n$. We assume that the size of $n$ is quite large, though it is typically dwarfed by the number of edges. Such large graphs can be stored only on disks. Typical maximum flow or minimum-cut algorithms cannot be efficiently applied to such cases, since the maximum-flow based methods (9) use combinatorial or flow-based techniques, which may require random access to edge information on disk. Such random access may cause so much performance deterioration, that it may become impractical to use these approaches effectively for disk-resident data. Furthermore, it is impossible to store pairwise information on minimum $s-t$ cuts effectively, since the number of possible pairs may be too large to store effectively.

A natural solution to this problem is to replace a single large problem, with multiple-smaller approximations which can be efficiently solved. The solutions from the different approximations can be combined in order to create a single robust solution. We will use a probabilistic compression approach in order to create the connectivity index. In order to design the connectivity index, we will use some contraction techniques which are also used in (50) for the memory-resident case. While the contraction ideas discussed in (50) are useful for finding a $global$ minimum-cut in a graph, they are not very useful for determining the minimum $s-t$ cut
between an arbitrary pair of vertices. Furthermore, we are interested in creating an index which allows responses to arbitrary source-sink pairs in online response times. In this chapter, we will show how to adapt some of the broad ideas in this approach effectively in order to create an efficient connectivity index. We will also provide a theoretical analysis of the effectiveness of such an index. This also allows the index to provide an estimate of the accuracy of the cut-based approach. While the accuracy of the minimum connectivity estimation is different for different pairs of vertices, we will see that the estimation is particularly effective for the case of vertex pairs in which the value of the underlying minimum cuts is low. Such cuts are also the most critical in network connectivity applications, because they could lead to network disconnection by the removal of a few edges.

The probabilistic compression approach creates multiple compressed representations of the underlying graph with the use of sampling. In order to create these representations, we need to sample edges from the underlying data. This is achieved by drawing on the concept of reservoir sampling. The compressed representations are used in order to create an index. The overall approach is to randomly sample the data in order to create edge sampled and node sampled compressions from the underlying data. We will first define edge-sampled compressions for creating compressed graphs:

**Definition 1** Let $G$ be a graph with node set $N$ and edge set $A$. For any fraction in $f \in (0, 1)$, an edge-sampled compression $H(f)$ of a connected graph $G$ is one in which we sample a fraction $f$ of the edges of the graph. Let this sampled edge set be denoted by $S$. We contract each connected component induced by the edge set $S$ into a single node. Then we use this contracted
set of nodes in order to reconstruct the contracted graph $H(f)$ using the original edge set $A$, which is denoted by $A'$ on the contracted node set. All self-loops (which will be created as a result of the contraction) from $A'$ are removed.

An example of graph coarsening process is illustrated in Figure 1. The set of edges sampled are $(1, 2)$, $(3, 4)$, $(5, 6)$ and $(5, 7)$ and are illustrated by bold lines. In this case the nodes within the dotted circles are compressed to super-nodes. This is because the nodes within the dotted circles remain connected after sampling. We will see that edge-sampled compressions have the property that the resulting compressed graph is probabilistically biased towards retaining cuts of lower value from the original graph. This is essentially because edges in dense components are more likely to be sampled, and will eventually result in a contraction. This is also why contraction based techniques (50) work well for determining global minimum cuts. However, the techniques of (50) cannot be directly used for disk-resident cases or for determining minimum
s-t cuts for a specific source or sink node. Furthermore, the aim of this chapter is to design an effective index for the disk-resident case. Next, we will define the concept of node-sampled compressions.

**Definition 2** Let $G$ be a graph with node set $N$ and edge set $A$. For any integer $m$, we can define the node compressed graph $J(m)$ as follows. For each node, we map it to one of the integers from 1 through $m$ by using an unbiased die. This is used in order to partition the nodes into $m$ sets. Each of the $m$ sets is then mapped into a contracted node. Then we use this contracted set of nodes in order to reconstruct the contracted graph $J(m)$ using the original edge set $A$, which is denoted by $A'$ on the contracted node set. All self loops (which will be created as a result of the contraction) from $A'$ are removed.

The overall algorithm uses the parameters $k_e$ and $k_n$ in order to regulate the number of edge-compressed and node-compressed graphs. The number of edge-compressed graphs is denoted by $k_e$, and the number of node compressed graphs is denoted by $k_n$. In addition, we use a target compression size $m_t$ which represents the target number of nodes in the compressed graphs. The optimal size of the compression is dependent upon the structure of the graph, and is therefore a hard problem. In general, we would like to pick a target compression size which is such that the resulting graph can be loaded in main memory. Note that the maximum size of the graph with $m_t$ nodes is at most $O(m_t^2)$. Therefore, for a memory size of $M$, it is best to pick a target size which is at least equal to $m_t \geq \sqrt{M}$. For the case of the node compressed graphs, this target is met in a straightforward way by choosing a die with $m_t$ sides. In the case
of edge-compressed graphs, this target is met by choosing the fraction $f$ of the edges carefully. We will discuss more on this issue slightly later.

The algorithm proceeds in phases. In the first phase the algorithm constructs $k_e$ edge-sampled graphs, each of which has a target size of $m_t$. Note that the definition of edge-sampled graphs is expressed as a function of the sampling fraction $f$ rather than a target number of nodes. Therefore, the sampling fraction needs to be picked carefully in order to obtain the desired target number of nodes. We will discuss this issue slightly later. In the second phase, we create $k_n$ node-sampled compressed graphs. In this case, we create $m_t$ nodes in each node sampled graph. We note that each of the compressed graphs typically required much lower space as compared to the original graphs. This is because the compressed graph will contain many parallel edges between the compressed nodes. Such parallel edges can be compressed into a single edge with the appropriate weight. Furthermore, a large fraction of the edges in the compressed graph will be self-edges which can be eliminated. For example, in the case illustrated in Figure 1, the compressed graph on the right hand side has fewer edges than the original graph on the left hand side.

Once all the compressed graphs have been created, we store the mapping of the original nodes to the new compressed node identifiers in a way which facilitates easy query processing. In order to achieve this goal, we use an inverted mapping from the original node identifiers to the compressed node identifiers of the different compressed graphs. Note that since a total of $(k_n + k_e)$ compressions are created, we need to store the mapping information for $(k_n + k_e)$ different graph compressions. For the node $i$, let the corresponding node identifiers in the
$h = (k_n + k_e)$ graphs be denoted $M(i, 1) \ldots M(i, h)$. Thus, we denote the mapping of the $i$th node in the $j$th compression by $M(i, j)$. We note that $i$ is an integer drawn from $\{1 \ldots n\}$, $j$ is an integer drawn from $\{1 \ldots h\}$, and $M(i, j)$ is an integer drawn from $\{1 \ldots m_t\}$.

**Algorithm 1 Overall Approach for Constructing the Connectivity Index**

Input: Graph $G$, Target node Size $m_t$.
Output: Edge-based compressions $k_e$, Node-based compressions $k_n$, Inverted mapping list.

\[
\text{GConnect}(G, m_t) \{ \\
1: \text{Create } k_e \text{ edge-sampled compressions of } G, \text{ each with target size } m_t; \\
2: \text{Create } k_n \text{ node-sampled compressions of } G, \text{ each with target size } m_t; \\
3: \text{Create inverted list on information for assignment of original nodes to node} \\
\quad \text{identifiers of compressed graphs; } \\
4: \text{return; } \\
\}
\]

The inverted index is defined as follows. For the $i$th node, we store the list of $h = (k_n + k_e)$ 2-tuples. Specifically, the list for the $i$th node contains the tuples $(1, M(i, 1)) \ldots (h, M(i, h))$. Thus, the size of each list is $h = (k_n + k_e)$. The overall approach for constructing the index is illustrated in Algorithm 1.

Once the index has been created, it can be used very effectively for query processing. For any user-specified pair of nodes $s$ and $t$ as source-destination pairs, we access the inverted lists for $s$ and $t$. These inverted lists provide us with the mapping to the node identifiers for the $h$
different graphs. It remains to describe how the index is used for query processing. We will also describe how the compressed graphs are constructed by edge and node sampling. In a later section, we will also provide an analysis as to why such an approach should be effective. Further, we need to design a way to implement this approach efficiently.

2.1.1 Creating the Indexed Representation from Compressed Graphs

In this section, we will discuss the process of creating the indexed representation from the compressed graphs. The actual process of creating the compressed graphs will be described slightly later. Let us assume that the total number of compressions is denoted by $h$. Then, for each of the $h$ compressions, a given node $i$ maps onto one of the $m_t$ partitions. As discussed earlier the mapping of the $i$th node for the $j$th compression is denoted by $M(i,j)$.

Given a massive graph, we create an inverted representation, in which each node $i$ points to a list denoted by $(1, M(i,1)) \ldots (h, M(i,h))$. Note that this inverted list provides the information necessary for determining the appropriate mappings to the sources and sink nodes in the compressed graphs for a given query. We also maintain a list of $h$ pointers to the disk-resident data structures for the compressed graphs. The compressed graphs are represented in the form of adjacency lists. For each node, we maintain a list of their edges, along with the corresponding weight, which is also the number of parallel edges which consolidate into a single edge. We note that these adjacency lists are much smaller than those of the original graph. This is because many of the edges in the graph are removed during compression, and other edges were consolidated during the compression process. While the graphs are stored on disk during the index creation, it is assumed that each individual graph is small enough to be read
into main memory for effective query processing. The additional space required by the inverted representation is \( h \cdot n \), where \( n \) is the number of nodes and \( h \) is the number of compressions. While this can be quite large, we will see that only two of these lists are accessed for a given query. This ensures that a given query can be resolved very efficiently.

2.1.2 Query Processing from the Connectivity Index

In this section, we will discuss the techniques for query-processing with the use of the connectivity index. The process of querying with the index is relatively straightforward. For a given query-node pair \((s, t)\), we first use the inverted representation in order to determine all the corresponding source-sink pairs in the compressed representation. These correspond to \((M(s, 1), M(t, 1)) \ldots (M(s, h), M(t, h))\) in the compressed representation. For a given graph index \(j\), we note that the pair \((M(s, j), M(t, j))\) represents the source-sink pair in the compressed representation. Also, we only need to use those pairs \((M(s, j), M(t, j))\) for which we have \(M(s, j) \neq M(t, j)\).

For each of these pairs, we determine the minimum-cut in the compressed graph. The minimum of all these cuts is reported as the final solution. The overall algorithm is illustrated in Algorithm 2. A key observation is that each of these graphs is assumed to be small enough to be read into main-memory. Since the minimum-cut \(s-t\) problem is known to be efficiently solvable for memory-resident graphs (9), it follows that each of these problems can be solved efficiently. Even though the minimum-cut may need to be determined multiple times over the different compressed graphs, it is still much more efficient to do so, because the disk-to-memory scale up for each individual problem needs to be taken into account in addition to the reduction
Algorithm 2 Query Processing with the Connectivity Index

Input: Inverted representation $I$, Source $s$, Sink $t$, Compressed graphs $H_1 \ldots H_r$.
Output: Value of minimum cut of the graph and the corresponding cut.

\[
\text{QueryProcess}(I, s, t, (H_1 \ldots H_r)) \{ \\
1: \text{Use inverted representation to determine mapping from nodes } s \text{ and } t \\
\quad \text{to corresponding nodes } M(s, r) \text{ and } M(t, r); \\
2: \text{for each compressed graph with index } r \in \{1 \ldots h\} \text{ do} \\
3: \quad \text{if } M(s, r) \neq M(t, r) \text{ then} \\
4: \quad \quad \text{determine minimum-cut } v_r \text{ using efficient in-memory algorithms; } \\
5: \quad \text{return the minimum among all values of } v_r \text{ and the corresponding cut; } \\
\}
\]

of the individual problem size. Most maximum-flow based techniques require augmenting path or other preflow-push techniques in order to determine the minimum-cut (9). Such techniques do not access the edges in any particular order which can be efficiently pre-stored contiguously. Therefore, such methods are not very practical on the original disk-resident graph, since they may require random accesses to disk. On the other hand, since our approach is able to read the entire compressed graph in main memory at one time, this results in much more efficient algorithm, even though we need to perform the operation on multiple graphs. The overall approach for query-processing is illustrated in Algorithm 2.

2.1.3 Creating the Compressed Representation

In this section, we will discuss the process of creating the compressed graphs from the data. We will construct the edge-sampled graphs with the process of reservoir sampling. The key difficulty is that the edge sampled graph $H(f)$ is defined in terms of sampling fraction $f$, but
we do not know what this sampling fraction should be in order for the compressed graph to have node cardinality which is approximately equal to the target $m_t$. We note that if we sample too many edges, this will typically result in some inefficiency in terms of storage, but we will see that the resulting reservoir can always be used to create a compressed graph of the appropriate size. Therefore, we will show how to use a logarithmic number of passes in order to achieve a reservoir of the appropriate target size. We note that the minimum number of edges required in order to create $m_t$ components is $n - m_t$, and such a graph would be a perfect forest with no cycles. Therefore, we first create a reservoir of $n - m_t$ edges in the first pass, and check if the resulting graph has no more than $m_t$ components. Typically, the number of components would be much greater than $m_t$ because of the presence of cycles in the sampled edges. Therefore, we will perform another pass and double the number of edges present in the reservoir, by adding an equal number of edges to those already present in the reservoir. We again check if the number of components in the graph is no more than $m_t$. We repeat the process of doubling the number of edges until the number of components in the resulting graph is no more than $m_t$. We then randomly delete edges one by one from the reservoir, until the number of components is exactly $m_t$. The final set of edges in the reservoir defines the $m_t$ connected components. We note that such an approach requires only a logarithmic number of passes over the data set in order to work effectively. We summarize as follows:

**Lemma 1** For a graph with $n$ nodes, the reservoir sampling approach requires $O(\log(n))$ passes over the data in order to determine the compressed representation of the graph for a target node size of $m_t$. 

The results of the lemma are easy to prove, since the size of the reservoir doubles in each pass, and an upper bound on the reservoir size is the maximum number of edges, which is at most $n^2$. Note however, that we need $k_e$ different reservoirs in order to construct $k_e$ different compressed graphs. Building the reservoirs sequentially would blow up the number of passes by a factor of $k_e$. Therefore, we can build all the $k_e$ reservoirs in parallel. In a given pass, we add to each of the reservoirs for which the number of components in the corresponding induced graph still exceeds $m_t$. Thus, the number of passes continues to be logarithmic, even when multiple compressions have to be handled at one time.

We note that the only purpose of the reservoir is to define the connected components in the graph. Once the connected components have been determined, we go back to the original graph, and perform the following steps:

- We compress each connected component in the original graph into a single node. We note that this will result in parallel loops and self-edges.
- We remove all self-loops. We also consolidate all parallel edges into a single edge with the appropriate weight.

The resulting graph is the edge-compressed graph. This graph is of significantly smaller size than the original graph. As in the previous case, the process of constructing the corresponding compressed graphs can be parallelized. The overall procedure for creating the edge-compressed graphs is illustrated in Algorithm 3.

Next, we construct the compression with the use of node sampling. In this case, each node is assigned to one of $m_t$ partitions by using random assignment, where the probability of each
Algorithm 3 Creating the Compressed Graphs

Input: Graph $G$, Target size $m_t$.
Output: $k_e$ Edge-based compressions.

EdgeCompress($G$, $m_t$) {
1: $s = n - m_t$;
2: Create $k_e$ edge reservoirs from $G$ with size $s$ each in one data pass;
3: while at least one of the $k_e$ reservoirs induces a graph with more than $m_t$ components do
4: Double the edges in the reservoirs which induce graphs with more than $m_t$ components;
5: end
6: Remove edges one-by-one from all reservoirs for which the induced graphs have less than $m_t$ components until the induced graph for each reservoir has exactly $m_t$ components;
7: Compress each connected component for the induced graphs into a single node;
8: Remove all self-loops from compressed graphs and consolidate parallel edges;
9: return $k_e$ edge-based compressions;
}

A question arises as to why we need both node-based and edge-based compression. As we will see, the process of edge-based compression is biased towards preserving cuts with low value. Dense subgraphs are typically compressed into a single node in edge-sampled compressions. This leads to the possibility that the source and sink may map into the same node for edge-sampled compressions. In this case, it is not possible to determine the minimum-cut over the compressed graph in order to estimate the minimum connectivity over the original graph. In order to guard against this possibility, we also add a few node-sampled compressions. As we will see in the analysis section, the addition of a small number of node sampled compressions allows us to
ensure with very high probability that the source and sink nodes map onto different nodes in at least one or more compressed graphs.

2.2 Analysis of Algorithmic Effectiveness

In this section, we will provide an analysis of the algorithmic effectiveness of the approach. We first make the following simple observation about the correctness of the approach.

**Observation 1** The query processing technique for the GConnect algorithm always results in an approximation, which is an upper bound to the true minimum-cut value.

The logic for this observation is as follows. For each cut in the compressed graph, a cut with equivalent value exists in the original graph. This ensures that the minimum-cut in the compressed graph is also represented in the original graph. On the other hand, the converse is not true. A cut in the original graph may not have a corresponding cut in the compressed graph. Therefore, the minimum-cut in the compressed graph is always an upper bound on the minimum-cut in the original graph. By using a larger number of compressed graph samples, this bound can be tightened. At a slightly later stage, we will investigate the nature of this bound with the use of different input parameters.

First, we will compute the probability that some cut can always be found with this sampling approach. We note that since the source and sink nodes may map onto the same node, a given compression may not yield a minimum-cut value in the compressed graph. If this is true across all compressions, then it will not be possible to provide an estimate of the minimum-cut value. Therefore, we define a \((s,t)\)-valid compression for a source-sink pair \(s-t\) as follows:
**Definition 3** A \((s, t)\)-valid cut for a given compression and a source-sink pair \(s\)-\(t\) is defined as a compression in which \(s\) and \(t\) map onto different nodes in the compressed graph.

We note that edge-sampled compressions typically tend to contract dense subgraphs into a single node. Note that when all edge-sampled compressions map the source \(s\) and sink \(t\) to the same node, it implies that the source and sink are densely connected. Such cuts are typically less important from the perspective of connectivity queries. This is because typical networking applications attempt to find communication bottlenecks with small minimum-cut value. Therefore, it is more important to determine those cases in which the connectivity between the source-sink pairs is as little as possible. However, even for such cases, we would like to be able to provide a reasonable estimate of the minimum-cut value. This is precisely the reason that node-sampled cuts are created. Since nodes are assigned randomly to the \(m_t\) different partitions, the probability that the source-sink pairs are assigned to different partitions is given by \(1/m_t\). When the sampling is repeated over \(k_n\) different independent compressions, the probability is given by \((1/m_t)^{k_n}\). We note that when for modest values of \(k_n\) and \(m_t\), such as \(m_t = 100\), and \(k_n = 10\), the probability of being able to obtain at least one \((s, t)\)-valid compression is \(1 - 10^{-20} \approx 1\). This probability is quite acceptable for most applications. In practice, a fraction \((m_t - 1)/m_t\) of the node-based compressions may turn out to be \((s, t)\)-valid, and this leads to an even more robust estimate. We summarize the result below:

**Lemma 2** Let \(k_n\) node-sampled graphs be created with node size of \(m_t\). Then, for any source-sink pair \((s, t)\), the probability of obtaining at least one \((s, t)\)-valid compression is given by \(1 - (1/m_t)^{k_n}\).
The entire purpose of the node-sampling approach is to allow a high probability of obtaining \((s,t)\)-valid compressions, even when the source and sink are chosen from the same densely connected component. Even a modestly small value of \(k_n\) is sufficient to achieve this goal.

Next, we will study the effect of sampling on the quality of the underlying cuts. We would like to provide the end-user with some estimates on the quality of the underlying cuts for a given query. To this effect, we will use the edge-sampled cuts, since they are biased towards retaining cuts of low value. For each of the \(k_e\) edge sampled graph compressions, let \(f(1, m_t) \ldots f(k_e, m_t)\) be the fraction of edges (from the original graph) which are needed to be sampled in order to reduce the compressed graph to \(m_t\) nodes. Note that while \(f(1, m_t) \ldots f(k_e, m_t)\) cannot be controlled by the end-user (since they are dependent on a randomized sampling process), they can certainly be known a-posteriori, once the reservoir-based sampling process has been completed. This estimation is helpful in providing the user with the necessary data required in order to provide feedback about the quality of the cut. This data is stored after the creation of the compressed graphs. We make the following assertion:

**Lemma 3** Let \(V(s, t)\) be the minimum connectivity between the nodes \(s\) and \(t\) in the original graph \(G\). Let us consider an \((s,t)\)-valid compression for the \(j\)th graph, where \(1 \leq j \leq k_e\). Then, the probability that the minimum-cut is unaffected by the compression is given by at least \((1 - f(j, m_t))^{V(s,t)}\).

**Proof.** We note that a minimum-cut will survive in the compressed graph, if none of the edges in it are sampled. The probability that a particular edge is not sampled is given by \(1 - f(j, m_t)\).
Since the minimum-cut contains $V(s, t)$ edges, it follows that the probability that none of the edges in the cut are sampled in the reservoir is given by $(1 - f(j, m_t))^{V(s,t)}$.

An important observation is that since $V(s, t)$ represents the number of edges in the minimum-cut, it typically contains far fewer edges than an average cut. This increases the relative probability of the survival of the minimum-cut. This is also the reason why such an approach for minimum connectivity indexing is likely to work in practice. Furthermore, we will see that typical values of the sampled fraction $f(j, m_t)$ are quite small. In such cases, the value of $(1 - f(j, m_t))^{V(s,t)}$ can be modestly large. For example, in important connectivity applications, cuts containing a small number of edges are especially important from an application point of view. For example, consider a vulnerable cut containing at most 5 edges, and $f(j, m_t) = 0.1$. In such cases, the probability that the cut survives is given by $0.9^5 \approx 0.59$. The use of multiple samples can reduce the probability of survival to arbitrarily large values. This is the reason for using $k_e$ different edge-sampled compressions. For example, if we use $k_e = 10$, and each of these samples are $(s, t)$-valid with a similar value of $f(j, m_t)$, then the probability that the minimum-cut survives in none of the compressions is given by $(1 - 0.59)^{10} \approx 1.34 \times 10^{-4}$. This probability is sufficiently small to assure us of the minimum-cut with high probability. While the value of $f(j, m_t)$ may be different across different values of $j$, the aim of this example was to show that the approach retains its effectiveness over small values of $k_e$. Next, we will quantify the effect of using multiple compressions on the probability of the survival of the minimum-cut value. In order to achieve this goal, we will introduce some additional notations.
For a given source $s$ and sink $t$, and index $j \in \{1 \ldots k_e\}$, we define the binary bit $B(j, s, t)$ to be 1, if the $j$th edge-sampled graph is $(s, t)$-valid. Otherwise, we define $B(j, s, t)$ to be 0. We note that the value of $B(j, s, t)$ can be determined from the inverted representation for a source $s$ and sink $t$. Then, the estimated value of the probability that the minimum-cut survives is as follows.

**Lemma 4** Let $V(s, t)$ be the minimum connectivity between the nodes $s$ and $t$ in the original graph $G$. Then, the probability that the minimum-cut is obtained from the $k_e$ different edge-sampled compressions is given by at least $1 - \Pi_{j \in \{1 \ldots k_e\}, \{B(j, s, t) = 1\}}(1 - (1 - f(j, m_t))^{V(s, t)})$.

**Proof.** These results can be proved directly from Lemma 3. We note that the minimum of the cut values is the true minimum-cut, if a minimum-cut survives in at least one of the $k_e$ edge-sampled compressions in which $s$ and $t$ map onto different nodes. The complement of this event is one in which the cut survives in none of the edge-sampled compressions. We can obtain the value of this complement by multiplying together the probabilities for each of the indices $j$ for which the value of $B(j, s, t)$ is 1. The individual probabilities in this product may be determined by using the results of Lemma 3. The result follows. We note that $V(s, t)$ is not known explicitly. However, an over-estimate on $V(s, t)$ can be obtained by using the least of the minimum $s$-$t$ cut values across the different compressions. As long as we use an over-estimate on the value of $V(s, t)$, the results of Lemma 4 continue to hold true. We use this over-estimate in order to provide bounds on the quality of the results obtained by the index.
2.2.1 Further Optimizations

As discussed earlier, we use node-sampled compressions in order to reduce the probability that the source and sink pair map to the same partition in a given compression. This situation often arises in the case of edge-sampled compressions, but does not arise too often in node sampled compressions. The reason that this situation often arises in edge sampled compressions is that some of the dense subgraphs are very large, and may absorb too many of the nodes. In order to improve the results further, we put a constraint on the size of each edge-sampled component during the contraction process. In other words, during the process of contraction, we ignore edges which are incident on the components whose size has reached this maximum. We note that this option essentially incorporates some of the concepts of node sampled compression into the edge sampled compressions as well. Furthermore, the node-sampled compressions are used for estimation in query processing only for cases where a valid pair is not available from the edge-sampled compressions.

2.3 Experimental Results

In this section, we will present the experimental results of the GConnect index. We will present the effectiveness and efficiency on a number of real data sets. We note that the minimum-cut method needs to be implemented as a subroutine for query-processing in our disk based index. For this purpose, we used the HIPR implementation available in (1). This is essentially an efficient version of the push-relabel algorithm (18) for the s-t maximum flow problem. We also create a disk-based implementation of this algorithm, in which each access to a node or an edge was required to go back to the disk in order to determine the appropriate
parameters. This also means that all operations such as retrieving an edge, updating the capacity information, or performing a relabel need to access the disk in an order which cannot be controlled a-priori. Because of this random access, the disk-based version is orders of magnitude slower than the memory-based version. Nevertheless, it is the only reasonable solution for very large graphs. In this section, we will show that the repeated use of the memory-resident technique (which is required by our index structure) is much more efficient than even a single application of the disk-based algorithm.

The algorithm is tested on six real data sets. The first five are matrix data sets, which are downloadable from the University of Florida Sparse Matrix Collection web site\(^1\). The particular data sets used from that web site were graham1, ex3sta1, Andrews, gupta1, and cage13. The sixth data set is the well known DBLP data set\(^2\).

The overall process comprises the main steps of index construction and query processing. We will test the effectiveness of query processing, and the efficiency of both. In the index construction step, we generate \(k_e\) edge-based contractions and \(k_n\) node-based contractions. The default value of \(k_e\) was set to 100, and the value of \(k_n\) is set to \(0.2 \cdot k_e\) in our experimental setting. We will explicitly specify the places at which these parameters are set to different values. The query processing step is tested with 500 queries in order to illustrate the effectiveness and efficiency of our algorithm. We will provide detailed results in this section.

\(^1\)http://www.cise.ufl.edu/research/sparse/matrices

\(^2\)http://www.informatik.uni-trier.de/~ley/db/
2.3.1 Accuracy Analysis

The objective of the GConnect algorithm is to efficiently determine the connectivity of a large graph by shrinking its size without significantly compromising accuracy. The effectiveness of the GConnect algorithm in terms of the compression factor and the accuracy is illustrated in Table I. The table provides the size of the original data set, the compressed graph size, the compression ratio, and the accuracy in terms of the percentage of time that the minimum-cut is correctly determined. We note that the compressed graph size is the average behavior of a single graph, and the total size of all compressed graphs may possibly be larger than the original data. The raw sizes of the data sets are dominated by the number of edges rather than the number of nodes. In each case, the results are averaged over 500 source-sink pairs. In order to calculate the compression ratio, we computed the percentage reduction in the size of the graph size as a result of the compression process. It is evident from Table I that the reduction in the size of the compressed graphs is greater than 95% in almost all the cases. Thus, the compressed graph is at least an order of magnitude smaller than the original graph. We will see in a later section that this reduced size results in orders of magnitude improvement in the processing times of the GConnect algorithm. This is especially the case because the running times scale quadratically with the underlying graph size.

In the last column of Table I, we have also illustrated the accuracy of the minimum-cut with respect to the true value. This accuracy is expressed as the percentage of source-sink pairs for which a correct cut value is returned. In three of the data sets, we obtained the correct minimum-cut over all 500 cases. In the other data sets, there were errors in only a small
TABLE I
SIZE REDUCTION AND OVERALL EFFECTIVENESS

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Original Size</th>
<th>Compressed Graph Size</th>
<th>Reduction (%)</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes/Edges</td>
<td>Nodes/Edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>graham1</td>
<td>9K / 0.21M</td>
<td>0.6K / 0.9K</td>
<td>99.58</td>
<td>5.8</td>
</tr>
<tr>
<td>ex3sta1</td>
<td>16K / 0.33M</td>
<td>3K / 12.7K</td>
<td>96.16</td>
<td>0.0</td>
</tr>
<tr>
<td>Andrews</td>
<td>60K / 0.37M</td>
<td>10K / 24.1K</td>
<td>93.15</td>
<td>0.0</td>
</tr>
<tr>
<td>dblp</td>
<td>164K / 0.76M</td>
<td>16K / 19.8K</td>
<td>97.35</td>
<td>2.8</td>
</tr>
<tr>
<td>gupta1</td>
<td>31K / 1.06M</td>
<td>12K / 17.2K</td>
<td>98.39</td>
<td>0.0</td>
</tr>
<tr>
<td>cage13</td>
<td>445K / 3.52M</td>
<td>40K / 90.1K</td>
<td>97.44</td>
<td>1.4</td>
</tr>
</tbody>
</table>

percentage of the cases. While Table I provides a good overview of the accuracy behavior of the algorithm, it provides little understanding of how this accuracy is affected by the minimum-cut value and the behavior of the erroneous cases. The value of the minimum-cut plays a role in the accuracy level, since the probability of incorrectly contracting the true minimum-cut increases with the value of the minimum-cut.

In order to provide better insight, we first examine the distribution of the accuracy with different minimum-cut sizes. We binned the output values of the minimum-cut into different ranges and show the accuracy over each range. The results for the six data sets are illustrated in Figures 2(a), 2(b), 2(c), 2(d), 2(f) and 2(e) respectively. The dark bar in each figure represents the number of cases tested, and the white bar represents the number of cases for which the correct minimum-cut value was returned. In the case of Figures 2(a), 2(b), and 2(c), the two
bars are exactly the same, because all 500 source-sink pairs are solved correctly for these data sets. On the other hand, for the cases of Figures 2(d), 2(f) and 2(e), we can see that there are a few errors in some cases. Most of the errors were for cases where the minimum-cut value was large. This is because a larger value of the minimum-cut is more likely to cause an incorrect contraction. Even in the cases, where the minimum-cut was not correct, the relative errors which were obtained were relatively small. Another observation is that every single minimum cut returned by the index (throughout the experimental section) was from an edge-sampled compression rather than a node-sampled compression. Nevertheless, the use of node-sampled compressions is necessary in order to ensure that a theoretically valid cut is always available.
Next, we will examine the error distribution of the cases in which an incorrect result was obtained by the algorithm.

![Figure 3. Relative error distribution plot](image)

We plot the distribution of the relative error behavior with the output value of the minimum-cut. The relative error is defined as the ratio of the absolute cut error to the correct minimum-cut value. The results are illustrated in Figures 3(a), 3(c) and 3(b). In this case, we present the results only for the three data sets in which some of the source-sink pairs yield incorrect results. Furthermore, since the number of incorrect cases is much smaller than the number of correct cases, we present only the results for the incorrect cases in order to enable clarity of presentation. The X-axis in each figure contains the range of relative error, while the Y-axis indicates the number of cases for which this range of relative error was achieved. In all cases, most of the erroneous cases had relative error which was less than 0.6. For two of the data sets (dblp and cage13), the majority of the cases had relative error which was less than 0.4. Thus,
for the small number of source-sink pairs in which the correct cut was not obtained, the relative error was quite small and provided a good idea of the minimum-cut value.

2.3.2 Computational Efficiency

All experiments were performed on a Microsoft Windows XP machine with Intel Core2 Duo 2.5G CPU and 1.5GB main memory. The algorithm was implemented in C++. The existing minimum s-t cut algorithms require random access to disk for each operation such as performing a flow augmentation on an edge or a relabel on a node. This makes the execution time orders of magnitude slower than memory resident algorithms. The aim of the repeated compression approach in our technique is to reduce the underlying graph size so that it can be efficiently solved with memory resident algorithms. We further note that the running times of maximum flow algorithms scale superlinearly with the graph size. Therefore, the computational time in processing a reduced graph is reduced by a much greater factor than the reduction in size. It is evident from Table I that the reduction in sizes of the different data sets are typically by a huge factor of nearly 20. Therefore, the CPU time reductions are by even greater factors. When the combine the savings from this reduction and the memory-based implementation, we will see that the query processing time of the GConnect algorithm is superior to the disk-based implementation in spite of the fact that the GConnect algorithm needs to run the method repeatedly in order to effectively approximate the minimum-cut.

We will examine the times required for index construction and query processing. In order to generate these results, we used the default value of 100 edge-sampled and 20 node-sampled compressions. The time required for index construction is illustrated in Table II. The index
construction time in Table II, is dominated by two components. The first component is the creation of the connectivity index with the use of a sample-based compression approach. This step essentially determines the mapping of nodes to the different partitions. This mapping is then efficiently stored in the inverted representation. The second component processes this compressed graph, removes all self-loops, and consolidates all parallel edges into a single edge with an appropriate weight. It is evident from the results of Table II, that the running times are quite modest in all cases. Since the index construction step is a pre-processing step, we have greater leeway in allowing for larger running times than the query processing step. It is evident from Table II, that the construction phase required only a few minutes in most of the data sets.
Next, we will examine the query processing efficiency of the *GConnect* algorithm. We will compare the query processing times with a disk-based version of the algorithm. We also implemented a disk-based version with a cache which was 5% of the size of the original graph. The cache was used to speed up processing. The results are illustrated in Table III. Each row in the table illustrates the running time for executing the minimum-cut algorithm on 50 different source-sink pairs. It is evident that the *GConnect* algorithm is several orders of magnitude faster than (both versions of) the disk-based implementation, even though it needs to be run multiple times on several compressed graphs. This is quite reasonable because each compressed graph is almost two orders of magnitude smaller than uncompressed graph, and we do not need to perform any disk-based computations. The overall result is that the query-processing time is *between two and three orders of magnitude faster* than the disk-based version of the algorithm.

It is also evident from the results of Table III, that our approach provides the difference between an impractical disk-resident solution and an extremely efficient query processing index for the problem. We further note that the caching technique was not particularly helpful because of the random access behavior of the underlying edges. As a result, the advantage of the cache did not significantly outweigh the extra overhead of maintaining the cache.

### 2.3.3 Sensitivity Analysis

The experimental results presented so far show that the *GConnect* algorithm is extremely effective and efficient over the different data sets. In this section, we will also illustrate the robustness of the technique. Clearly, the effectiveness of the algorithm is sensitive to the number of compressions. Since edge-sampled compressions are more crucial to the effectiveness of our
TABLE III
RUNNING TIME COMPARISON

<table>
<thead>
<tr>
<th>Data Set</th>
<th>GConnect Query Processing Time (seconds)</th>
<th>Disk-based Running Time (seconds) No Cache</th>
<th>Disk-based Running Time (seconds) 5% Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>graham1</td>
<td>1.80</td>
<td>1,550</td>
<td>1,345</td>
</tr>
<tr>
<td>ex3sta1</td>
<td>7.59</td>
<td>2,300</td>
<td>2,164</td>
</tr>
<tr>
<td>Andrews</td>
<td>22.80</td>
<td>3,070</td>
<td>3,004</td>
</tr>
<tr>
<td>dblp</td>
<td>8.20</td>
<td>4,712</td>
<td>4,457</td>
</tr>
<tr>
<td>gupta1</td>
<td>12.66</td>
<td>5,245</td>
<td>5,076</td>
</tr>
<tr>
<td>cage13</td>
<td>78.37</td>
<td>59,339</td>
<td>58,710</td>
</tr>
</tbody>
</table>

approach, we will test its effectiveness with increasing number of edge-sampled compressions. We will compare the different techniques over various ranges of minimum-cut values. The results for the different data sets are illustrated in Figures 4(f), 4(c), 4(d), 4(a), 4(b) and 4(e). The X-axis corresponds to the minimum-cut size and the Y-axis corresponds to the accuracy. We have illustrated the accuracy for different number of compressions by bars of different colors. Clearly, the accuracy of the GConnect algorithm increases with increasing number of compressions. However, it is interesting to see that the GConnect algorithm was able to provide obtain modestly accurate results with as few as 30 compressions in some data sets. A particular example is the case of the gupta1 data set, which is illustrated in Figure 4(c). In this case, the accuracy is always 100% even when as few as 30 compressions are used. In
In general, the results are extremely robust across different numbers of compressions for the data sets illustrated in Figures 4(c), 4(a), and 4(b). In these cases, extremely robust results may be obtained by using as few as 100 edge-sampled compressions. These results indicate that the GConnect technique is an extremely robust method which is several orders of magnitude more efficient than currently available techniques, and can be used effectively in a wide variety of practical scenarios.
CHAPTER 3

MINING TRANSFORMED GRAPHS BY SUMMARIZATION

The graph summarization technology is also not limited to applications where data is naturally modeled as graphs. In many cases, manifesting the data as graphs may reveal profound insights into the data from a completely new perspective; but just due to the structural complexity of graphs and the large size of data under examination, one may feel reluctant to make this transformation happen. This is exactly where our summarization techniques come in. In this chapter, we explore the possibility of utilizing a compact graph representation to solve traditional frequent pattern mining problem. As we can see later, in a novel way we can summarize large transactional databases into a very small but informative graph. With such an effective transformation, we no longer have to suffer from the conventional time-consuming mining procedure that grows patterns from small size to large size. Instead, graph summarization offers us the opportunity to directly jump to the final stage of the conventional procedure without generating the sheer amount of mid-sized patterns.

Frequent pattern mining problem is considered as one of the fundamental research topics in data mining, whose applications extend to many domains, including but not limited to association rule mining, classification and clustering (6; 14; 24). Over the past decade, frequent pattern mining has been extensively studied by many data mining researchers, leading to a lot of publications and algorithms for mining frequent patterns and/or other extended objectives (30). Also, there are many related open source software available online (2).
Frequent pattern mining is essentially a challenging problem due to the huge number of item combinations that might exist. Agrawal et al. proposed Apriori, which leverages the downward closure principle to reduce search space (8). Based on the property that all subpatterns of a frequent pattern must be frequent as well, this principle significantly reduces the size of the candidate pattern set, leading to great improvement of mining performance. Later, more and more algorithms building on this same principle have been proposed, which further improve the computational efficiency (7; 31; 11; 63; 43). Here, one representative method is FP-growth (31): It deploys depth-first search instead of the breadth-first level-wise strategy of Apriori, and achieves much success by aggressively shrinking the projected database size.

So far, most of the existing algorithms we have mentioned mine the complete set of frequent patterns in a database. However, as (69) also suggested, when the length of frequent patterns becomes large, none of these methods can produce results within a reasonable amount of time, mainly because of the following reasons:

First, in a database, all possible combinations of items are exponential in size, which means that the complete set of frequent patterns might become too large to handle. The introduction of closed or maximal frequent patterns (44; 10) can alleviate the result size issue to a certain extent, but still there are many cases where the output pattern set is just too big, even though we only target closed or maximal patterns.

Second, to get the complete set of frequent patterns, one must go through the complete set of candidates in a bottom-up mining method. Here, the length of the longest frequent pattern is a good indicator regarding the size of such intermediate candidate set, for example, if there
is a frequent pattern with 100 items, then according to the downward closure principle, all its \(2^{100}\) sub-patterns are also frequent, which means that one has to fully traverse this huge pattern space before the fully-fledged 100-item pattern can be finally grown. To this extent, it is much more costly for existing methods to produce long patterns.

The above dilemma is intrinsic to the frequent pattern mining problem, if one wants the complete result set. However, as we observe in many applications, mining tasks in practice usually attach greater importance to patterns with larger size, for example, in bioinformatics, long sequences are usually of much higher significance compared to those short ones. Thus, if the complete set of results is prohibitively large and only a small number of them are of practical interests to us, it becomes problematic to wait for the mining algorithm to grow patterns from the bottom to the top, i.e., from short patterns to long patterns, as it may get stuck at the intermediate stage.

Motivated by the above analysis, we try to design a new method that can mine those interesting long patterns without going through the painful process of growing an explosive candidate set. In particular, we are interested in solving the following problem: For a database, can we efficiently get its top-\(k\) long maximal frequent patterns?

Compared to the state-of-the-art bottom-up approaches, we explore the opposite direction and propose a *top-down* mining strategy in this chapter. The main idea is as follows. We note that items and their co-existence relationships can be summarized by a so-called pattern graph, where frequent patterns will show up as *cliques*; this makes cliques promising frequent pattern candidates. Now, we can transform the problem of finding long maximal frequent patterns into
the detection of large maximal cliques: Starting from the pattern graph generated from the input dataset, we try to find a list of maximal cliques with large size, where each maximal clique is one that cannot be further enlarged by adding more vertices onto it. The essence of this step is to bypass the bottom-up pattern growth procedure since we have directly jumped to the top of the search space. As will be shown in the rest of this chapter, although it is not guaranteed that each maximal clique $c$ will represent a true pattern $p$ in this case, due to the filtering done by the pattern graph, it is likely that the set of items in $c$ are only slightly different from those in $p$; to this end, we develop a separate refinement step to effectively transform $c$ into $p$. Finally, the set of large cliques will give rise to a set of large frequent patterns, and we could follow this strategy to find the top-$k$ long maximal frequent patterns.

Recapitulating the above discussions, we outline our contributions in this chapter as follows.

First, unlike existing bottom-up approaches, we try to solve the frequent pattern mining problem from a totally different perspective that is top-down in nature. In this way, we avoid the intrinsic complexity of generating the vast majority of frequent sub-patterns while still being able to reach those interesting long ones. Usually, this latter part of the pattern space is not very large, which can be easily accessed by our top-down mining algorithm.

Second, an innovative two-pronged frequent pattern mining framework (clique detection, followed by candidate refinement) is introduced. We transform the transaction database into a pattern graph and perform mining on it instead. As the pattern graph size is only determined by the number of frequent items in the database, the time spent to work on these graphs is
independent of either the database size (i.e., number of transactions) or the complete pattern set size (as in bottom-up approaches). This is very desirable in terms of computation time.

Third, we test our proposed idea and algorithm by conducting extensive experiments on various real datasets. The results show strong evidence of our method’s efficiency and effectiveness. Our top-down algorithm is able to accurately pinpoint top-\(k\) long maximal frequent patterns when state-of-the-art methods fail to generate any interesting results, since bottom-up methods are ultimately stuck in the pattern growth process.

This chapter is organized as follows. Section 3.1 gives the preliminaries, which includes definitions of the problem and the pattern graph we will be using. We introduce the overall top-down mining framework in Section 3.2, followed by Section 3.3 and Section 3.4 that focus on each of the two stages of the algorithm: clique detection and candidate refinement, respectively. Section 3.5 presents experimental results and demonstrates the efficiency and effectiveness of our algorithm. Section 3.6 discusses related work.

### 3.1 Preliminaries

#### 3.1.1 Problem Definition

Let \( I = \{i_1, i_2, \ldots, i_m\} \) be the set of all the \( m \) items in a given database, then a \textit{pattern} or an \textit{itemset} refers to a subset of \( I \). Particularly, in this chapter, a \textit{pattern} including \( l \) items is called an \( l \)-item \textit{pattern}. The database \( D = \{t_1, t_2, \ldots, t_n\} \) is basically a collection of itemsets. The support set \( D_p \) of a pattern \( p \) is the set of transactions that contain the entire \( p \), i.e., \( D_p = \{t_i \mid p \subseteq t_i \text{ and } t_i \in D\} \), whose size is denoted as \(|D_p|\).
Definition 4 (Frequent Pattern)  Given a transaction database $D$, a pattern $p$ is frequent if and only if $|D_p| \geq \sigma$, where $\sigma$ is a user-specified support level threshold that satisfies $0 \leq \sigma \leq |D|$.

As we have already mentioned, since mining of the complete set of frequent patterns might not always be doable and long patterns are in general more interesting, our discussions will be focusing on those frequent patterns having large size. Specifically, our objective in this chapter is to mine top-$k$ maximal frequent patterns, where top-$k$ refers to the $k$ patterns with biggest length, and a frequent pattern is maximal if there is no super-pattern of it which is also frequent.

3.1.2 Pattern Graph Construction

Following previous discussions, we will make use of a so-called pattern graph to register the mutual relationships among items in a transaction database, which can guide us through the search for potential long frequent patterns. This proposed concept is inspired by the thoughts of merging small patterns to quickly generate large pattern candidates. We outline it as follows.

Most of the existing pattern mining algorithms produce candidate patterns by growing one item at a time from scratch. This is undesirable and we went on to consider the possibility of adding several items together in one shot. Based on the downward closure property, these several added items must form a frequent pattern by themselves. This hints us to first mine all small frequent patterns $P^s$ up to a predetermined size $s$, and then different patterns in $P^s$ can be further combined to grow the target pattern in a much faster manner. Since $s$ is usually not very large as we choose it, this step of generating small frequent patterns can be efficiently performed by state-of-the-art algorithms.
We draw a pattern graph to understand when several small patterns can be combined and merged together. Essentially, a pattern graph is a compact representation of the complete set $P^s$ of $s$-item frequent patterns. To start with, each item is represented by a node in the pattern graph, and an edge is drawn between two nodes if the corresponding items co-exist in at least one pattern of $P^s$. To measure how often two items appear together with each other, we also associate a weight with each edge that equals the number of times its two end items show up in the same pattern of $P^s$.

Table IV and Figure 5 illustrate an example of constructed pattern graphs. The original transaction database is shown in Table IV, where 2-item and 3-item frequent patterns are also given; the support level has been set to 1. With such a database, Figure 5 draws the pattern graphs derived from 2-item and 3-item frequent patterns, respectively. For example, $a$ and $c$ are both in a 2-item $ac$, and thus there is an edge $a-c$ in the 2-item pattern graph; $b$ and $d$ are both in a 3-item $bcd$, and thus there is an edge $b-d$ in the 3-item pattern graph. Moreover, since $b$ and $d$ both exist in another 3-item $bde$, they appear in the 3-item frequent patterns for a total of two times, which means that the weight of edge $b-d$ in the 3-item pattern graph is 2. Note that, since each pair of nodes can co-exist in at most one 2-item frequent pattern, the weight of an edge in a 2-item pattern graph is always 1 (if there is such an edge), which we have chosen to omit in the picture.

As we can see from the above example, the size of a pattern graph is independent of the number of transactions in a database as well as the choice of $s$. This advantage means that we can compress a huge transaction database into a compact representation, on which further
TABLE IV
TRANSACTION DATABASE EXAMPLE

<table>
<thead>
<tr>
<th>Transactions</th>
<th>2-item patterns (σ = 1)</th>
<th>3-item patterns (σ = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b, c}</td>
<td>ab, ac, ae, af, bc, bd, be, cd, ce, cf, de, ef</td>
<td>abc, ace, aef, bcd, bde, cef</td>
</tr>
<tr>
<td>{b, c, d}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{b, d, e}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{a, c, e, f}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Sample pattern graph constructed with 2-item patterns (a) and 3-item patterns (b)

mining can be performed. The cost of constructing a pattern graph is equal to that of mining the whole set of $s$-item frequent patterns. Given a database of considerable size, mining all frequent patterns up to size $s$ is much more doable than mining every single pattern targeted by traditional algorithms. Thus, it should be able to finish in a reasonable amount of time.

Taking Figure 5 as an example, we examine the condition under which several small $s$-item patterns can be merged into a large one. First, we have a frequent pattern acsf in the
transaction database; now, it is interesting to see that acef happens to correspond to a clique in the pattern graph, which is the case in both 2-item and 3-item pattern graphs. We formalize this observation into the following lemma, and it will be useful in guiding us through the search for large-sized maximal frequent patterns.

**Lemma 5** If a pattern \( p = \{i_{q_1}, i_{q_2}, \ldots, i_{q_n}\} \) is frequent, then in the \( s \)-item pattern graph constructed, there should be a corresponding clique consisting of all items in \( p \), plus that the weight of each edge in the clique is greater than or equal to \( \binom{n-2}{s-2} \).

**Proof.** Suppose a pattern \( p = \{i_{q_1}, i_{q_2}, \ldots, i_{q_n}\} \) is frequent, then according to the downward closure property, any of its \( s \)-item subpattern is also frequent. Consequently, when we construct the \( s \)-item pattern graph, all items in \( p \) will be connected with each other, which naturally forms a clique. Meanwhile, considering the link weight on the edge between item \( i_{q_1} \) and item \( i_{q_2} \) in pattern \( p \), \( i_{q_1} \) and \( i_{q_2} \) should appear together for no less than \( \binom{n-2}{s-2} \) times in a \( s \)-item frequent pattern (each time they show up with \( s - 2 \) other items in the set \( \{i_{q_3}, i_{q_4}, \ldots, i_{q_n}\} \), the count increment by 1). Finally, we conclude that the weight of links between any two items in \( p \) is no less than \( \binom{n-2}{s-2} \).

Following above discussions, in a pattern graph constructed from \( s \)-item frequent patterns, if a large clique meets the condition specified in Lemma 5, then it may quite likely lead to a potential frequent pattern with large size. Although this condition is not sufficient, it is a very good starting point and our empirical studies have confirmed the pruning power of this compact but surprisingly effective graph representation. With the pattern graph, now instead
of walking through the pattern growth procedure step by step, we can directly jump to the top of the pattern space as indicated by the clique constraint and start exploration right from there; this is a clear advantage offered by our top-down mining strategy.

### 3.1.3 Problem Extension

In above, we defined our target as finding top-\(k\) maximal frequent patterns. Note that, this definition is actually very general in the sense that there are many variants of it that can be used in other important applications. For example, if item coverage is emphasized, we can find top-\(k\) maximal frequent patterns covering any particular item \(i\) in the database. This can be easily achieved by working on a projected version of the pattern graph that contains \(i\), its neighbors and all edges among them. More generally, such targeted mining can be extended to the scenario where we want the patterns found to include a collection of predetermined items.

### 3.2 Top-Down Graph-based Frequent Pattern Mining Framework

Figure 6 depicts the overall working flow of the mining algorithm we propose. It has two stages: clique detection and candidate refinement.

**Clique Detection.** We first mine all frequent patterns of size \(s\) and construct the \(s\)-item pattern graph for the transaction database. Usually, the value \(s\) is chosen to be relatively small so that state-of-the-art algorithms can perform well. We shall further discuss the choice of \(s\) in Section 3.3. Now, the mining of long maximal frequent patterns in the database becomes the problem of finding large-sized maximal cliques, and we could generate a list of maximal cliques having the largest sizes as potential candidates. Here, the nicety is that pattern graph size is only determined by the number of different items in the database (which should be a
Figure 6. The top-down mining framework

moderate number), and thus finding maximal cliques is independent of either the database size (i.e., number of transactions) or the complete pattern set size. This is very desirable in terms of processing time.

**Candidate Refinement.** In the clique detection step, we only target large maximal cliques, where one reason is that a clique with \( m \) vertices could generate \( 2^m \) cliques as its subparts and we cannot afford checking all of them including non-maximal ones. From Lemma 5, we know that a maximal frequent pattern \( p \) will surely induce a clique that satisfies certain constraints in the pattern graph. However, this clique might not be maximal: Suppose \( p \) contains the set of items \( I(p) \), it is possible that there exists a set of other items \( I(o) \) in the pattern graph, which seemingly pass the connectivity and weight constraints with regard to \( I(p) \) as indicated in Lemma 5, but actually cannot be merged with \( I(p) \) to form a bigger frequent pattern. For
example, in Figure 7, \( I(p) = \{a, b, c, e\} \) is a frequent pattern, but the maximal clique candidate is \( abcd e \) while \( abcd e \) is not frequent and unrelated items \( I(o) = \{d\} \) have been incorrectly merged. The pattern graph pruning is not a sufficient condition here and could generate false positives. To cope with this possibility, we have to come up with a refinement step that can drop \( I(o) \) out of the clique candidate \( c \) and recover the pattern \( p \) we want. Note that, due to the effective pruning performed by pattern graph, it is very hard for a relatively small pattern \( p \) to combine with a substantial number of unrelated items and form a large-sized maximal clique, which implies that the item difference between the observed large clique candidate \( c \) and actual target long pattern \( p \) should not be too big. This makes the refinement a doable task, which will be covered with more details in Section 3.4.

### 3.3 Clique Detection

As we introduced above, the candidate generation step detects cliques in the constructed pattern graph. Since our objective is to mine large patterns without going through the time-consuming pattern growth procedure, naturally, we want to find those cliques with large size. Moreover, we will only focus on maximal cliques in the following discussions. Non-maximal ones are completely covered by other cliques and thus less interesting.

Therefore, what we will be looking for are those largest cliques in the pattern graph, say the top-\( k \) largest. This is a classic NP-hard problem. However, as we have already mentioned, the size of the pattern graph generated for a particular database is independent of the database size as well as the value of \( s \) when constructing the \( s \)-item pattern graph, which should be no
greater than the total number of frequent items in the database. With a moderate graph size, it allows for fast deployment of clique detection routines.

Bearing this in mind, the algorithmic details of finding largest cliques is not a particular concern of us. For experiments, we will use a modified version of the clique detection algorithm proposed in (42) along with the free software igraph introduced in (22). Note that, because of Lemma 5, if a clique of size $n$ is going to be a valid pattern candidate, then each of its edges must have a weight of at least $\binom{n-2}{s-2}$ in an $s$-item pattern graph; this could serve as a further pruning condition to integrate into any existing clique detection method.
3.3.1 The choice of $s$

An important parameter we need to decide in our method is the value of $s$ when constructing an $s$-item pattern graph. Naturally, one might expect that the pattern graph’s pruning power would become stronger as $s$ gets larger. For instance, in the example of Figure 5, the 2-item pattern graph on the left indicates that $abce$ is a possible clique candidate; however, $abce$ will be disqualified in the 3-item pattern graph on the right, since the weight on $ab$ is 1, which is smaller than $\left(\frac{4-2}{3-2}\right) = 2$ and thus violates the condition specified in Lemma 1. This also applies to the clique $bcde$.

However, the construction of an $s$-item pattern graph requires us to mine from the database all frequent patterns up to length $s$, and this will for sure cost more computational resources when $s$ gets larger. In this chapter, we intend to use up to 3-item frequent patterns for pattern graph construction, as they can be computed quickly and experimental studies have also confirmed the strength of their pruning power. Of course, things always have to be balanced, and one can freely choose other values of $s$ to trade off the construction time and pruning power of pattern graphs.

3.4 Candidate Refinement

As we pointed out earlier, this section shall target the problem that a maximal clique candidate found may not identically correspond to a frequent pattern we want to see. Briefly mentioned in Section 3.2, we know that any maximal clique $c$ we observe in the pattern graph
is the combination of a true frequent pattern $p$ and some other unrelated itemset $o$, where $o$ is the root cause of possible false positives. In such a relationship:

$$c = p \cup o,$$

two interesting questions to consider are:

1. How large is the true pattern part $p$.

2. There might be multiple ways of writing $c$ as the combination of a pattern and other items, i.e., $c = p_1 \cup o_1 = p_2 \cup o_2 = \cdots$.

For question 1, because of the usage of pattern graphs, $p$ should not be too small and $o$ should not be too large; if $o = \phi$, then the clique corresponding to $p$ is just itself, i.e., no false positives. Based on this assumption, our algorithm works as follows. We extract top-$k'$ largest maximal cliques from the pattern graph, and try to recover the top-$k$ longest maximal frequent patterns from them. Here, $k' \geq k$ is a parameter, and our belief is that longest maximal patterns should appear as largest maximal cliques, because otherwise a shorter pattern needs to absorb more unrelated items in order to make its clique as large as that of those longer patterns, and this is not very likely due to the filtering of pattern graphs.

For question 2, we need to keep in mind that our candidate refinement procedure should be able to recover multiple possible patterns from a single clique. To make it clearer, let us still look at the example in Figure 7, where a transaction database, all of its 3-item frequent patterns when support equals 1, and the corresponding pattern graph are displayed. As we
can see, patterns \{a, b, d, e, f\} and \{c, d, e, g\} have their own maximal cliques abdef and cdeg. However, abcd is a maximal clique of size 5 and the weight of each link is no less than 3; according to Lemma 5, it should be a qualified candidate, although \{a, b, c, d, e\} turns out to be a false positive. We have two true frequent patterns embedded in the clique: \{a, b, c, d\} and \{a, b, c, e\} that need to be recovered, nonetheless.

Now let us take a closer look at how the refinement step should actually be implemented. For a clique candidate c, our major concern is to correctly identify the set of unrelated items o and remove them, so that we can find the target frequent pattern p. The naive strategy here is to exhaustively try out all possible item combinations in c for removal; however, this is not going to be very efficient and we shall propose a heuristic below to guide us through. Note that, although the heuristic is pretty effective in deciding the correct removal order as our experiments show, it is still possible that it will make mistakes and remove the wrong items; to this extent, we need a complementary phase to add these items back after the removal happens. Also, as an answer to the question 2 in above, we will add some amount of randomization into the whole process so that our search should not just try one particular direction, which can only recover one potential pattern.

Two subsections are given next to go over more details about each of the two phases in this candidate refinement step.

3.4.1 Item Removal

The open question in this first phase is about the order in which we remove different items. Definitely it is not ideal if one just randomly picks items and removes them one by one. Be-
fore moving to discussions on our strategy, we first introduce an alternative way of counting a candidate pattern $c$’s occurrences in the input database, where $c$ corresponds to a maximal clique detected before it is passed to the refinement step. Specifically, we will count not only transactions that contain all items in $c$, but also those including a substantial portion of $c$. The size of this portion is determined by a parameter $t$, i.e., a transaction would only be counted if it has more than $t$ fraction of items that are present in the itemset of $c$. To formalize it, we call each of these transactions a $t$-valid supporting transaction of $c$ with regard to the threshold $t$, which collectively form $c$’s $t$-valid support set.

**Definition 5 (t-valid Support Set)** Given a candidate pattern $c$, its $t$-valid support set includes all $t$-valid supporting transactions of $c$ in the database, where $t$ is a percentage parameter that is often set close to 1.

Given the $t$-valid support set of a candidate $c$, for each item $i$ in the candidate, we count the number of times it appears in some $t$-valid supporting transaction. This counting indicates the frequency $i$ appears in $c$’s $t$-valid support set, which can be used to assign removal priority to different items.

The reasoning is as follows. Since our target $p$ is a subpattern of the candidate $c$ and $p$ is usually not too different from $c$, a $t$-valid supporting transaction “approximately” covering $c$ actually corresponds to one that may potentially support $p$ after some items are removed. If an item appears in most of the transactions in the $t$-valid support set, then it is less likely that
it does not appear in the supporting transactions of the target pattern we want to recover, and consequently, we may want to eliminate those items that appear less often in the \( t \)-valid set, which probably have been wrongly merged into the target, giving rise to a false positive.

The overall removing procedure works as follows. First, given a candidate maximal clique \( c \), if the corresponding pattern is frequent, then it means that we have already found a valid long maximal frequent pattern and thus nothing needs to be done. Otherwise, items are set to be removed from \( c \) based on the priority specified above. Here, in order to not have the removal follow just one direction, which may only recover one potential pattern, we integrate a randomizing factor \( r \in [0,1) \) as described below. When taking on an item in the ordered list pending removal, there is a chance of \( r \) that we bypass it and jump to the next item in the list: In this way, a diversification effect exists if we iterate the whole process several times, since now the removal can go into multiple directions while still preserving the nice property of prioritizing those more promising ones. The removing of items from \( c \) is stopped as soon as the new candidate \( c' \) becomes frequent: Because we are trying different ways of removing items from a candidate maximal clique to achieve maximal frequent patterns, it is not meaningful to remove items from an already frequent pattern as this can only generate non-maximal patterns. In other words, items removed beyond this point will be recovered by the later extension phase anyway, and thus it is better to retain them in the first place.

Also, in order to accelerate the process, we do not have to strictly follow the above process by removing one item at a time, and thus the recalculation of \( t \)-valid support set can be saved for intermediate rounds. Specially, in each iteration, we intend to remove \( m_{del} > 1 \) items,
and our experiments confirm that the resulting quality is almost as good. It is possible that removing multiple items in one round may not let us stop right at the boundary of frequent patterns, however, this is OK given the adding phase we will be discussing next, where items unnecessarily removed are finally recovered.

### 3.4.2 Candidate Extension

After the last phase of removing items, we get a new candidate $c'$ that is frequent. The process of extending $c'$ to make it maximal is much more straightforward. We project the original database against $c'$ to drop all transactions that do not contain $c'$, remove items that appear in $c'$, and then mine maximal frequent patterns on the projected database. The result patterns are concatenated with $c'$ to form maximal frequent patterns of the original database at last. Note that, since we stop item removal at the frequent boundary (or, almost at the boundary when multiple items are removed in each round), it is reasonable to expect that the $c'$-projected database would be small and result maximal patterns mined from it would be short. To this extent, we will just utilize the existing method to work in this phase.

To end the discussions of this section, we summarize the candidate refinement process in Algorithm 4.

### 3.5 Experimental Results

In this section, we test our proposed algorithm in terms of efficiency and effectiveness. For the purpose of evaluation, we use LCM (51), the winner of FIMI 2004, as our baseline. As one of the traditional bottom-up mining methods, LCM traverses the whole pattern space and produces a complete set of maximal frequent patterns, which can then be filtered for an exact
Algorithm 4 Candidate Refinement

Input: \(cList\) – a list of top-\(k'\) cliques.
Output: \(pList\) – a list of maximal frequent patterns.

CandidateRefinement\((cList)\) {
    1: \(pList = \emptyset\);
    2: repeat
    3: for each \(c \in cList\) do
    4:     while \(c\) is not frequent do
    5:         sort items in \(c\) according to their frequencies in \(c\)'s \(t\)-valid support set;
    6:         repeat
    7:             remove the current lowest frequency item from \(c\) with probability \(1 - r\);
    8:         until \(m_{del}\) items have been removed;
    9:         build projected database w.r.t \(c\) and mine largest frequent pattern \(ext\);
    10:        \(p \leftarrow c + ext\);
    11:        add \(p\) into \(pList\);
    12: until \(pList\) has \(k'\) different patterns;
    13: return \(pList\);
}

answer of top-\(k\) longest ones. This can be used as the ground truth when testing the result
quality of our algorithm.

As we mentioned in Section 3.1, our method is in fact general enough to specifically find
patterns containing particular items. Here, to extensively experiment with our Max-Clique
algorithm, we will sample a subset of items in the test datasets, mine the top-\(k\) maximal
frequent patterns covering each of them, and then report the result quality and running time of
Max-Clique on an average basis. All three steps: pattern graph construction, clique detection,
and candidate refinement, are accounted for when recording the running time of Max-Clique.
We use two real datasets: chess and accidents, which are both available from the Frequent Itemset Mining Dataset Repository\(^1\). The chess dataset contains 3,196 transactions, and each is a board description for endgame and has a length of 37. There are 75 items in total. Since we are particularly interested in large data, we replicate everything in chess for 100 times.

The accidents dataset (26) records traffic accidents that occur with injured or deadly wounded casualties on a road in Belgium. In total, 340,184 traffic accident records are included, each of which contains a subset of 572 attributes (i.e., items), with an average size of 45. 20 items have been sampled from both datasets to mine top-k maximal patterns containing each of them.

All experiments are done on a Debian GNU/Linux server with two dual-core Xeon 3.0GHz CPUs and 16GB main memory. The program is written in C++. Max-Clique has a few parameters: the number \(k'\) of largest maximal cliques to extract, the threshold \(t\) in \(t\)-valid support set, the randomizing factor \(r\) when recovering multiple patterns from a single clique, and the number \(m_{del}\) of items removed in one round for acceleration purposes. Due to space limit, we will present parameter sensitivity analysis for \(m_{del}\) later in this section, while the other parameter values being used are: \(k' = 2k\), i.e., twice the number of patterns we want to find, \(t = 0.8\), \(r = 0.3\), and \(m_{del}\) is defaulted to 5 except in the sensitivity analysis.

### 3.5.1 Efficiency Analysis

We compare the computational efficiency of Max-Clique with LCM. The running time of both algorithms increase with decreasing support level, which are drawn in Figures 8(a) and

\(^1\)http://fimi.cs.helsinki.fi/data/
8(b) for *accidents* and *chess*, respectively. \( k \) is set to two values: 1 and 10. As we can see, the execution time of LCM has a dramatic upward trend toward the right hand side of the graph:

When the support level goes to as low as 10k, we had to omit one point for LCM since the program could not finish within 10 days. In comparison, *Max-Clique* remains stable, due to the advantage of our top-down mining framework that only picks the “iceberg” of the pattern space, which is orders of magnitude faster than the baseline LCM.

Unlike LCM, since the running time of *Max-Clique* depends on the choice of \( k \), it is interesting to see how this trend looks like, as well. The running time on two datasets are shown in Figure 9(a) and Figure 9(b). Note that, we have only included the execution time of clique detection and candidate refinement in these two graphs, because varying \( k \) has no impact on
the execution time of pattern graph construction, plus that it only takes a small fraction of the total time used, anyway. As can be seen in the figures, in most cases we are able to get the top-10 target maximal frequent patterns within 10 minutes, even if the support level is very low. Meanwhile, the curves move like linearly with regard to \( k \), which is expected after taking out the time to construct pattern graphs.

Figure 9. Efficiency on different \( k \) values

### 3.5.2 Quality Measurement

In order to quantify the effectiveness of our approach, we use a ranking based precision measure. First, each of the top-\( k \) maximal frequent patterns in the database is ranked in descending order of size and associated with a weight. Specifically, the \( i \)-th pattern in the list
will have a weight of $k - i + 1$, for example, the largest pattern in a top-5 list is assigned 5, while the smallest one is assigned 1. The score used to measure the quality is computed as follows,

\[ Q = \frac{\sum_i (I_i \ast Weight_i)}{\sum_i Weight_i}, \]

where $I_i$ is an indicator function denoting whether Max-Clique can successfully output the $i$-th pattern in the true top-$k$ ranking list. The idea behind this measure is to attach more importance to larger patterns, because presumably, more information will be lost if Max-Clique cannot catch them.

Figures 10(a) and 10(b) depict the resulted quality scores over different $k$ values for the two datasets accidents and chess. Note that, in Figure 10(a), we do not have data for the support level of 10k, as LCM cannot finish within 10 days and thus we are not able to get the ground truth for measurement. Overall, Max-Clique is pretty effective in mining the top-$k$ patterns, given the fact that it can run much faster than LCM. However, we also observe that, the algorithm’s performance is a little worse on chess than on accidents, and this is due to the reason that patterns in chess overlap more often, which is the case we discussed in Section 3.4 that one has to recover multiple patterns from a single clique. Though we proposed the randomized scheme to alleviate this issue, it is hard to eliminate the problem completely, and we would like to investigate this further in future works.

### 3.5.3 Parameter Sensitivity Analysis

In this subsection, we focus on $m_{del}$ which has been introduced to accelerate item removal. We also tuned the other parameters $k', t, r$ a little, but in general, Max-Clique is not very
We focus on top-5 maximal frequent patterns in this experiment. Let us first look at the results on *accidents*: Figure 11(a) illustrates the quality scores over various values of \( m_{del} \), and Figure 11(b) shows the corresponding running time. As seen from the graphs, our algorithm works nearly perfect at all tested support levels even when we try to remove \( m_{del} = 6 \) items at a time. In the meantime, the procedure becomes faster if \( m_{del} \) gets larger. The cutting of running time is more obvious when the support level is low, say 10k, when patterns are bigger and it takes more time to refine candidates. The fact that removing more items in one round may result in a longer candidate extension phase later also contributes to the less effectiveness of increasing \( m_{del} \) when support is higher. Nonetheless, the acceleration of item removal seems quite beneficial in overall. Similar results are also seen for the *chess* dataset.
Figure 11. Sensitivity on different $m_{del}$ values (dataset accidents)

3.6 Related Work

The problem of frequent pattern mining was first brought up by Agrawal et al. in 1993 (6). Since then, many data mining researchers have put a lot of efforts on this topic and published a good amount of papers (8; 7; 31). There are many efficient implementations, as well, including FPClose (29), LCM (51), and those appearing in FIMI workshops. Meanwhile, frequent patterns have had many important applications such as classification/clustering (14; 24), and some efforts have been spent to extent it to more general scenario like uncertain data (4).

Most of the algorithms we mentioned above leverage the Apriori principle. At the same time of improving efficiency, this downward closure property leads to an explosive number of frequent patterns as the complete pattern set is mined. This becomes more problematic when the database is large and the support level is set low to catch long patterns. Researchers tried to overcome this issue by introducing the concepts of closed and maximal frequent patterns.
(44; 10); however, it is still a partial solution without directly tackling the root cause of the problem. Works dedicating to the mining of maximal/closed patterns include MAFIA (11), CHARM (63), CARPENTER (43), and so on.

Different from state-of-the-art algorithms, we focus on more interesting long patterns in this chapter and we want to directly jump to these long ones without following the pattern growth approach that essentially generates the complete frequent pattern set. In the sense of not exploring the whole pattern space, (69) is closest to ours in spirit, which does some leap on the pattern lattice by fusing small core patterns. In comparison, Max-Clique is even more aggressive by reverting the search order, since the improvement of (69) is still achieved within the bottom-up framework.
CHAPTER 4

INDEXING APPLICATIONS

Given the proliferation of graph data like 2D/3D objects, program execution traces, chemical compounds, social/biological networks and XML, there has been an increasing need for data management tools that can facilitate the storage and utilization of graph structured data. Search, as one of the most basic primitives, is a functionality that is urgently needed. For instance, for a database of graphs representing the bond structures of chemical compounds, one may want to find those database entries that contain some particular structural fragment with interesting properties; on the other hand, for a large biological network, one may want to find certain regions that involve a representative substructure, so that more can be understood about the possible functional implication of such structures. There are more real-life examples in various applications such as pattern recognition and structured information querying (20; 64), which cannot be enumerated here.

There are two different types of search people often want to perform on graph data. For a query graph $q$ and a target graph $g$, in some cases, one may only care about the binary answer of whether an identical copy of $q$ can be found in $g$; while in some other cases, which usually involve graphs with larger size and more complex internal structures, it becomes necessary to get not only whether but also where such identical copies of $q$ can be located in $g$. For instance, in the biological network example we mentioned above, the searching application requires the
discovery of the query $q$ in possibly more than one confined region of the graph $g$, so that relevant information could be gathered regarding the local aspects of $g$’s networked structure.

In the following, we shall refer to the first type of graph search that only cares about a binary containment relationship between query $q$ and target $g$ as graph containment search, while the second type of graph search that looks for specific locations in $g$ with an identical copy of $q$ is referred to as graph location search. Obviously, in both cases, it is inefficient to perform graph search by checking subgraph containment relation by direct subgraph isomorphism test since the problem is NP-complete. To this extent, there have been a lot of works that tried to build various kinds of indices on either graph databases or single large networks so that the overall searching process can be accelerated. Specifically, in graph containment search, there have been quite a lot of existing studies devoted to this front with feature-based methods being the majority (28; 57; 15; 68; 46; 65; 67). In order to accelerate searching, these feature-based methods construct index with frequent structure that can help prune unpromising candidates during the on-line querying phase, and those frequent patterns are shown to be effective as indexing features.

Given the thoroughness of the research works, it might seem that the problem of graph containment search has been well solved. For graph location search, intuitively, if we build an index and again leverage small frequent fragments especially frequent subgraphs as indexing features, the problem can also be effectively solved. However, nowadays more and more graph data come with a large size, and the need to perform search on graphs with a large size is urgently expected. For instance, a software system call graph describes the dependencies among
various system call procedures in a software package, and one may want to determine which programs have been contaminated by viruses or security holes by performing a graph search. Unfortunately, as we encounter more real examples, traditional indexing method, originally designed for graphs with moderate size, do not fit well in the large graph scenario. In this chapter, we are going to examine the difficulties of indexing in the context of both database of large graphs and single large network. Towards those difficulties, we further propose two novel algorithms that well employ frequent subgraph as indexing features and thus effectively tackle the searching problem.

The remainder of this chapter is organized as follows. Preliminary concepts are reviewed in Section 4.1. The problem of graph containment search and location search are studied in Section 4.2 and Section 4.3, respectively. We discuss related works in Section 4.4.

4.1 Preliminaries

In this chapter, we use the following notations: For a graph $g$, $V(g)$ is its vertex set, $E(g) \subseteq V(g) \times V(g)$ is its edge set, and $l$ is a label function mapping a vertex or an edge to a label.

**Definition 6 (Subgraph Isomorphism).** For two labeled graphs $g$ and $g'$, a subgraph isomorphism is an injective function $f : V(g) \rightarrow V(g')$, s.t., $\forall v \in V(g)$, $l(v) = l'(f(v))$; and second, $\forall (u, v) \in E(g), (f(u), f(v)) \in E(g')$ and $l(u, v) = l'(f(u), f(v))$, where $l$ and $l'$ are the labeling functions of $g$ and $g'$, respectively. Under these conditions, $g$ is called a subgraph of $g'$, denoted as $g \subseteq g'$, while $f$ is called an embedding of $g$ in $g'$, and $f(g)$, i.e., the image of $g$ in $g'$ under the injection $f$ is called an occurrence of $g$ in $g'$. 

In general, there could be many possible ways to construct the injective function $f$ between a query graph $q$ and a target graph $g$, resulting in more than one occurrences of $q$ in $g$. Depending on different application requirements, there are two different types of graph search we formally define below.

**Definition 7 (Graph Containment Search)** Given a graph database $D = \{g_1, \ldots, g_n\}$ and a graph query $q$, find all graphs $g_i$ in $D$, s.t., $q \subseteq g_i$.

**Definition 8 (Graph location Search)** Given a single large network $G$ and a graph query $q$, find all occurrences of $q$ in $G$, represented as respective embedding functions from $V(q)$ to $V(G)$.

In Section 4.2, we target at the graph containment search problem whose definition is exactly identical to normal graph containment search tasks discussed in the past, except that the graphs we look at have much larger size, usually orders of magnitude larger than those targeted in previous studies, e.g., one typical dataset people used for testing is the graph structures of chemical compounds, which have around 20 vertices and 20-30 edges on average.

In Section 4.3, we will specifically examine the index construction problem to facilitate graph location search on a single large network.

### 4.2 Graph Containment Search

As introduced before, graph containment search, i.e., given a graph database $D$ and a graph query $q$, find all graphs in $D$ that contain $q$ as a subgraph, can be achieved efficiently with feature-based indexing being the key idea. Current feature-based indexing technique often
works in a filtering-and-verification framework. During off-line index construction, a set of indexing features are selected carefully. For each feature $f$, we build an index which points to those database entries that contain $f$. When an on-line query $q$ is issued, a graph $g_i$ in the database can be safely pruned from the result set without performing any subgraph isomorphism checking, if feature $f$ is contained in $q$ ($f \subseteq q$) but is not contained in $g_i$ ($f \not\subseteq g_i$). Suppose a good set of features is indexed, the graph feature based scheme described above has been shown to result in a significant query processing speed-up. However, as we will see soon, the traditional method does not work well when directly applied to the case of containment search on databases with large graphs.

To start with, generating an initial set of indexing features is a bottleneck for the case of large graphs. Historically, people have been using frequent subgraphs for indexing features which were shown to provide satisfactory performance. However, the time complexity of subgraph isomorphism, the core routine of any frequent subgraph mining algorithms, grows exponentially with the graph size. As shown in (32), all the representative graph indexing schemes are tested for databases of small graphs only. Though not aimed at indexing, there exist several studies to conduct mining on large graphs, e.g., the one using a summarization technique for frequent subgraph mining (12). But even that, directly using frequent subgraphs thus mined will not lead to satisfactory indexing performance as will be shown in the experiments of Section 4.2.3, with the major reasons analyzed as follows.

The first obstacle we are facing is that the query processing in the large graph scenario may be less efficient. Consider the following example in Figure 12, given a query $q$ that contains
$f_1$ and $f_2$, any graph $g_i \in D$ not indexed by $f_j(j = 1, 2)$, i.e., $f_j \not\subseteq g_i$, can be safely pruned. However, as the database graph $g_i$ becomes larger, more diverse internal structures are added. Consequently, it becomes more and more likely for any small size feature $f_j$ to appear in $g_i$, limiting $f_j$’s selectivity and generating much more candidates that require direct verification against $q$. The root cause behind this inefficiency is that, during query processing, different features, e.g., $f_1$ and $f_2$, are always treated separately. Looking at Figure 12, the only instance of $f_1$ in $g_i$ is at its top-left corner, while the only instance of $f_2$ is at the bottom-right. Treating $f_1$ and $f_2$ separately leads to the conclusion that $g_i$ is a candidate containing $q$ since $f_1 \subseteq g_i$ and $f_2 \subseteq g_i$, and traditional graph feature based indexing inevitably breaks down here. But does that really mean no pruning opportunities? For the situation shown in Figure 12, we are sure that $q$ is not a subgraph of $g_i$; otherwise, it would imply a non-disjoint pair of $f_1$ and $f_2$ appearing in $g_i$, and this is contradicting to what we observe. So, interestingly, if we consider $f_1, f_2$ in an interdependent context and remember the way they combine into the full query $q$, then it provides important hints as to whether $g_i$ is a valid candidate containing $q$. Essentially, the relative positioning of features in $g_i$ has to be exactly the same as that in $q$.

Apart from the low selectivity issue above, another disadvantage of conventional methods is its inability to provide effective indexing features. There are two issues here: inadequacy and redundancy. When mining indexing features from large graphs, we want to set the support level somewhat high to curb the computational complexity. Because of the high support, this kind of features cannot prune much as they tend to be small and focus more on the commonalities between different graph instances. We need a mechanism to consider the potentially substantial
differences among large graph instances. Meanwhile, given the large size of graphs, it is quite possible that we end up with an explosive set of frequent subgraphs that are quite similar to each other in terms of their indexing utilities. This introduces a huge amount of redundancy.

The key contributions of our proposed algorithm are outlined next. We are the first to identify the three major issues associated with direct indexing using frequent patterns mined from large graphs (say using the summarization technique): low selectivity, inadequacy and redundancy. Towards the resolution of these issues, we propose a novel indexing scheme: CP-Index (Contact Preservation Index), which consists of contact preservation filtering, dual feature generation, and size-increasing bootstrapping feature selection to tackle each of them, respectively. Our newly designed CP-Index framework is depicted in Figure 13.

The key concept behind our scheme is leveraging contact preservation among features for more effective indexing. Different from traditional schemes, we record every feature’s locations of occurrences in the database graphs off-line and check their relative positioning during query-
ing time to make sure that, if two features touch upon each other in a query (e.g., they share some vertices/edges in some particular way), then the same contacts are preserved in any true candidate graph in the database. In another word, these contacts work just like nails, which can correctly glue multiple features together and form a valid occurrence image of the query. Fortunately enough, with contact preservation, we alleviate the need to mine larger subgraph features in search for higher selectivity.

To account for the potentially large variabilities among graphs because of their big size, a new dual feature generation method is proposed to address the inadequacy issue. In addition to generating shared features common to the whole database, we decide to mine graph-specific features that are particular to one single graph because they offer better query indexing for individual database entries. Furthermore, regarding the potentially explosive size of the subgraph feature collection, we introduce a strategy called size-increasing bootstrapping to select an opti-
mal subset for index construction, which directly uses the CP-Index query processing to decide whether it pays off to index an additional feature. Note that our size-increasing bootstrapping strategy is general enough to handle both shared and graph-specific features.

4.2.1 Contact Preservation

As mentioned before, we propose utilizing the contact preservation property of features to test their compatibility with each other and explore further pruning opportunities, in an attempt to augment feature selectivity. This requires us to first pre-compute each feature's locations in the database graphs, so that its position relative to others can be quickly assessed during on-line query processing.

Given a feature $f$ and a graph $g_i$ in the database $D$, an embedding of $f$ in $g_i$ records one location where $f$ exists in $g_i$. We can surely store every embedding of $f$ in $g_i$ before queries come in, but in order to keep things compact, we decide to summarize the full information as follows. Suppose $f$ has $t$ vertices, i.e., $V(f) = \{v_1, v_2, \ldots, v_t\}$; for each $v_k \in V(f)$, we store a list of vertices in $g_i$ which $v_k$ can be mapped to. Denote this list as $L^k_f(g_i) \subseteq V(g_i)$, where $1 \leq k \leq t = |V(f)|$, then for each vertex $u \in L^k_f(g_i)$, there must exist a subgraph isomorphism from $f$ to $g_i$ under which $u$ is the image of $v_k \in V(f)$. In the following, we shall refer to $L^k_f(g_i)$ as the CP-Index of node $v_k$ in $g_i$, the set $\{L^k_f(g_i)|v_k \in V(f), k = 1, 2, \ldots, t\}$ as the CP-Index of feature $f$ in $g_i$. Similarly, when all graphs in $D$ are considered, we will obtain the CP-Index of a node or a feature in the whole database.

In Figure 14, feature $f$ is a clique with 6 vertices, while database graph $g_i$ has a clique with the same size in it. All vertices in $f$ and $g_i$ have the same label. If we record all embeddings of
Figure 14. Constructing the CP-Index

In $g_i$, there will be $6! = 720$ embeddings, and to store each of them, a list of 6 IDs is needed, e.g., $(u_2, u_5, u_6, u_3, u_1, u_4)$. In comparison, any of $v_1, v_2, \ldots, v_6$ has a CP-Index of

$$\{u_1, u_2, u_3, u_4, u_5, u_6\},$$

so in total the space saving of CP-Index is $\frac{720 \cdot 6}{6 \cdot 6} = 120$ fold.

Let us use an example to demonstrate how the CP-Index could help graph containment search query processing. Looking at Figure 15, which is redrawn based on Figure 12, if $f_1 \subseteq g_i$, $f_2 \subseteq g_i$, then the traditional graph feature based indexing paradigm cannot prune $g_i$ based on the respective containment relationships between $f_1$ or $f_2$ and $g_i$. However, with CP-Index, we know that if the query $q$ is contained by $g_i$, then the subgraph isomorphism that embeds $q$ in $g_i$ must map $f_1$’s 3rd vertex and $f_2$’s 2nd vertex to the same vertex $u$ in $g_i$, i.e., $u \in L^3_{f_1}(g_i) \cap L^2_{f_2}(g_i) \neq \emptyset$, because they both correspond to $v \in V(q)$. The same holds for $f_1$’s 1st vertex and $f_2$’s 1st vertex. Based on this non-empty requirement, $g_i$ can now be
easily pruned. With $f_1$ only existing at the top-left corner of $g_i$ and $f_2$ only existing at the bottom-right corner, the CP-Index of $v_3$ in $f_1$ is

$$L^3_{f_1}(g_i) = \{u_3\},$$

while the CP-Index of $v_2$ in $f_2$ is

$$L^2_{f_2}(g_i) = \{u_5\}. $$

The two intersect to $\phi$. This clearly shows that no valid embedding of query $q$ should exist in the target graph $g_i$.

Figure 15. Query processing in CP-Index

Now we are ready to formally introduce our proposed CP-Index framework. It consists of three steps: index construction, search and verification, with enhanced capabilities compared to traditional schemes.
1. **Off-line index construction**: Generate and select a feature set $F$ from the graph database $D$. For feature $f \in F$, let $D_f$ be the set of graphs in $D$ that contain $f$, we compute the CP-Index of $f$ for each $g_i \in D_f$, which can be easily implemented as inverted vertex ID lists over $V(g_i)$.

2. **Search**: Test indexed features in $F$ against the query $q$. It returns all features that are a subgraph of $q$, $F_q = \{f | f \in F, f \subseteq q\}$. Then we compute the candidate query answer set $C_q = \bigcap_{f \in F_q} D_f$. To make use of the CP-Indices, we consider $F_q$ as a whole and check whether the contacts made among features in $F_q$ are preserved in each of the candidates. This will further cut down $C_q$’s size, just as what we saw in Figure 15.

3. **Verification**: Directly check subgraph isomorphism between query $q$ and each graph $g_i$ in the final candidate set $C_q$ to see whether $q$ is contained in $g_i$.

Before moving forward, let us spend a bit more time on each of the steps above. For index construction, we have assumed that a feature set $F$ is already available, after which the CP-Index of each feature $f \in F$ are computed and stored. However, generating such an $F$ is never a straightforward task. We will come back to this in Section 4.2.2 for a more detailed discussion.

To make the framework clear, we have intentionally omitted elaborated descriptions in the search step. As can be seen, the first phase of this step is identical to traditional feature based indexing, but since for each $f \in F$, we have further saved the CP-Index of $f$ in $g_i$ instead of simply a graph ID $i$ that indicates $f$ is a subgraph of $g_i$, more pruning possibilities exist. The whole procedure of leveraging CP-Indices during query processing is outlined below.
• After getting the set of features contained in \( q \), we locate each \( f \in F_q \) in the query, which results in a mapping between \( f \)'s vertices and \( q \)'s vertices.

• Considering all these mappings for all features \( f \in F_q \), now for each of query \( q \)'s vertices \( v \in V(q) \), we have a set of vertices, each of them belongs to some feature \( f \in F_q \) and has been mapped to \( v \) when we located \( f \) in the query, e.g., in Figure 15, for \( q \)'s vertex \( v \), we have \( f_1 \)'s \( v_3 \) and \( f_2 \)'s \( v_2 \) corresponding to it. These vertices all have their associated CP-Indices pointing to the list of nodes in a candidate graph \( g_i \) where they can appear. Let us intersect all CP-Indices, and the resulting vertex list should narrow down the possible nodes in \( g_i \) which \( v \) could be mapped to if \( q \) is embedded in \( g_i \).

• After multiple intersections, the list mentioned above should have minimal length, and could sometimes be empty, e.g., in Figure 15, the CP-Index of \( v_3 \) in \( f_1 \) and the CP-Index of \( v_2 \) in \( f_2 \) intersect to \( \phi \), hinting that there is no single vertex in \( g_i \) which \( v \) in the query \( q \) can be mapped to. Thus, if there exists some \( v \in V(q) \) such that the intersection result for \( v \) is empty after all aforementioned processing, then \( g_i \) is safely pruned from the candidate set \( C_q \).

Alternatively, we may understand this pruning strategy as follows. Given a query \( q \), we leverage the location information entailed in the CP-Indices of features in \( F_q \) to pinpoint \( q \)'s location in the graph \( g_i \). In a word, the contacts made among features in \( F_q \) must be fully preserved, as otherwise, they cannot combine into a full query graph \( q \) in \( g_i \). This contact preservation property among features is checked by examining the compatibility of their associated CP-Indices. Now, every vertex \( v \in V(q) \) works like a nail that glues multiple indexed
features together: It nails down their relative positioning, which must be held intact as we try to embed \( q \) in \( g_i \). This is why we could intersect relevant CP-Indices to narrow down the possibilities of mapping \( q \)'s vertices to \( g_i \)'s, and also prune \( g_i \) if the intersection result is empty.

Finally, verification works as a last resort by directly executing a subgraph isomorphism test when contact preservation based query processing still could not prune a candidate graph out of consideration. Hopefully, at the time of verification, the candidate set has been shrunk to a very small portion of the initial database. Note that, for any remaining candidate say \( g_i \), the contact preservation property can only yield a non-empty subset of vertices for each node \( v \) in the query \( q \), denoted as \( V_{v\rightarrow} (g_i) \subseteq V(g_i) \), which points to all possible nodes in \( g_i \) which \( v \) can map to. We call \( V_{v\rightarrow} (g_i) \) as \( v \)'s location list in \( g_i \). Interestingly, even though the location lists are not null, they still prune away lots of possibilities. In fact, now the subgraph isomorphism test routine could limit its scope by only mapping \( v \) to some vertex in \( V_{v\rightarrow} (g_i) \), instead of trying all possible vertices in \( V(g_i) \) without any guidance. In other words, the pruning we have achieved in the search step can to a large extent cut down the computation time that has to be spent in the verification step. Considering an extreme situation, if the set \( V_{v\rightarrow} (g_i) \) for every \( v \in V(q) \) contains one single vertex, then there is only one probable way to embed \( q \) in \( g_i \) by mapping each \( v \in V(q) \) to that single vertex, and our verification procedure just needs to test this sole potential embedding and verify whether it is true or not.

### 4.2.2 Feature Generation and Selection

As we pointed out in the beginning, using frequent subgraphs as indexing features on a database of large graphs is inflicted by the inadequacy and redundancy issues, even if summa-
rization techniques could help reduce the time complexity to mine out an initial feature set. In this section, we introduce two novel concepts to address these two problems in separate steps: 1) *dual feature generation*, which summarizes the database graphs and then mines both shared and graph-specific features to tackle the inadequacy issue, and 2) *size-increasing bootstrapping*, which treats the output of step 1 as an initial set, and then performs contact preservation based redundancy analysis to select an optimal subset for index construction. The details of both steps will be given in Section 4.2.2.1 and Section 4.2.2.2, respectively.

### 4.2.2.1 Dual Feature Generation

Given the big graph size, most frequent graph pattern mining algorithms would experience a hard time to produce results efficiently due to the high time complexity of subgraph isomorphism. To solve this dilemma, the authors of (12) proposed a way to summarize graphs in the database by randomly merging groups of nodes with identical labels and collapsing respective edges. With the graph summarization technique, it is feasible to obtain an initial frequent subgraph set $F_0$ in a given database with large graphs. However, unfortunately, using this initial set $F_0$ to build indexing for large graphs as in traditional graph feature based schemes will not lead to satisfactory indexing performance. One issue that might stem from it is that, we may have to set the minimum support threshold $\text{min\_sup}$ somewhat high as compared to the regular settings in previous works. Since small structures are very easy to be contained by a large graph, there would be too many patterns if the support threshold is low. Now, adjusting $\text{min\_sup}$ upwards definitely makes the mining much easier, but it also negatively influences the pruning power of obtained features, because they are contained in many entries of the database.
and thus not very selective. In Section 4.2.1, we proposed to leverage contact preservation properties and alleviate this issue by considering multiple features inter-dependently during query processing. Here, we further suggest another round of graph-specific mining to directly tackle the low selectivity problem.

As a large graph offers the opportunity to use its own subgraph features to improve pruning, graph-specific mining generates a feature set specific to a single graph $g_i$ in the database $D$. Considering a $g_i$-specific feature, we simply compute and store its CP-Index for only one graph $g_i$. With the additional graph-specific indexing, this is referred to as the dual feature generation approach. Now, suppose we have a shared feature set $F^D$ for the database and $n$ graph-specific features sets $F^{g_i}$ for each $g_i \in D$ (details about how $F^D$ and $F^{g_i}$’s are determined will be given in Section 4.2.2.2), the search process of a query graph $q$ proceeds as follows.

1. **Shared feature indexing.** With the index on shared features $F^D$, follow the search process described in Section 4.2.1 and obtain a candidate query answer set $C_q \subseteq D$.

2. **Graph-specific feature indexing.** For each remaining candidate $g_i \in C_q$, we have a non-empty location list $V_{v \rightarrow}(g_i)$ for every $v \in V(q)$ after phase 1. Now consider $g_i$-specific features $F^{g_i}$ and determine the subset $F^{g_i}_q$ that are subgraphs of $q$. We use $F^{g_i}_q$ and their associated CP-Indices on $g_i$ to further prune the location lists above by taking intersections, so that some list would finally drop to empty, letting us remove $g_i$ from $C_q$.

As one could easily see, this query processing procedure is very similar to what we outlined in Section 4.2.1, except that it works in two phases and the pruning only happens for $g_i$ (instead
of the whole database) when doing $g_i$-specific feature indexing. Back to how graph-specific features should be mined in the first place, we still leverage graph summarization techniques, transform $g_i$ to its summary $s_i$ and mine the database $\{s_i\}$ with $min\_sup = 100\%$. Strictly speaking, we are only borrowing frequent pattern mining algorithms here to extract subgraphs of $g_i$ for use as indexing features, because every subgraph of $g_i$ would have a support of 100% in such a single-entry database. So, to avoid pattern explosion, we decide to only find patterns up to $maxL$ edges and terminate the mining process beyond. Meanwhile, one can also repeat the summarization-mining workflow for multiple rounds, i.e., transform $g_i$ to $s_i^1, s_i^2, \ldots$ and merge the result patterns obtained on $\{s_i^1\}, \{s_i^2\}, \ldots$. Since summarization may cause pattern loss, i.e., a pattern in $g_i$ might not necessarily be a pattern in $s_i$, it will be exponentially harder for a $g_i$-pattern to miss from the output of all mining rounds given that the merging/collapsing of nodes/edges during summarization is fully randomized. Consequently, patterns found on summarized graphs can go arbitrarily close to the true ones, which is desirable for indexing purposes. At last, when it is appropriate, either early termination after $maxL$ edges or repeated mining can also be applied in generating the shared feature set for indexing the whole database.

4.2.2.2 Feature Selection by Size-Increasing Bootstrapping

To make things compact, we do not want to keep everything output from step 1 in the index. Suppose the initial feature set is $F_0$, in this section, we are going to perform redundancy analysis and select a non-redundant set of features $F \subseteq F_0$ for CP-Index construction. The result set should preserve the utility of $F_0$ to the largest extent. Together with the query
processing procedure discussed previously, this completes our indexing and search pipeline on large graphs.

As the name implies, if a feature $f$ is redundant with regard to a feature set $F$, it means that most of its utility can be well substituted by those already in $F$. Referring to our query processing procedure, maintaining $f$ in the index allows us to use $f$’s CP-Index and directly point to the locations where $f$ occurs in the database graphs. How can we not index $f$ but still achieve almost the same benefit? For an on-line query $q$, feature $f$ is only useful when $f \subseteq q$. Due to the chain effect, each $f' \in F, f' \subseteq f$ must be contained in $q$, as well, implying that $f$’s subgraphs in the feature set $F$ can also be used when searching $q$ in the database. Now, a natural question to ask is, can the utility of $f$ be substituted by the set $\{f' | f' \in F, f' \subseteq f\}$?

Keeping $f$ out of the index, the impact is low if we can reconstruct $f$’s CP-Index with little effort during on-line query processing. Note that, with an existing set of features $F$, we do not have to compute from scratch the locations where $f$ can appear in a target graph, but rather with the help of CP-Indices maintained for $\{f' | f' \in F, f' \subseteq f\}$, which is exactly what we are going to do when searching $f$ on the graph database $D$. So, interestingly, we could reuse the search framework proposed above to help decide an optimal set of features. This is referred to as the size-increasing bootstrapping approach of feature selection.

**Definition 9** (Redundant Feature). Given a database $D$, an existing feature set $F$ and another feature $f$, treat $f$ as the query and perform a search of it in $D$, with $F$ being the set of indexed features. Now, for each candidate graph $g_i$, we obtain a set of location lists for vertices in $f$ by intersecting the CP-indices of features in $F$. Comparing with the true set of $f$’s location lists
in $g_i$ (i.e., $f$’s CP-Index), we say that $f$ is redundant with regard to $F$ on graph $g_i$, if these two sets are very similar to each other.

There are many possibilities of defining the above similarity between location lists and aggregating the metric over all graphs in $D$, which is the reason why we did not want to stipulate a particular way in Definition 9. In this section, we choose to use the following specification for redundancy. Referring back to the end of Section 4.2.1, we used $V_{v \rightarrow} (g_i), v \in V(f)$ to denote the location list of $v$ in a target graph $g_i$. Multiplying the length of all location lists for $f$, we can model the search space of testing whether $f$ is a subgraph of $g_i$ as

$$\prod_{v \in V(f)} |V_{v \rightarrow} (g_i)|,$$

because a potential embedding of $f$ in $g_i$ may result from taking one vertex from each of the location lists and using them to comprise a full image of $f$. If this metric of search space size is further aggregated over the whole database $D$, we have

$$\text{Cost}(f) = \sum_{g_i \in D} \prod_{v \in V(f)} |V_{v \rightarrow} (g_i)|$$

to model the cost of searching $f$ in $D$. Note that, if there exists some $v \in V(f)$ such that $V_{v \rightarrow} (g_i) = \emptyset$, i.e., the location list of $v$ in $g_i$ is empty after pruning, then the cost entry corresponding to $g_i$ is 0, which is inline with our indexing framework because it means that CP-Index can directly remove $g_i$ from the candidate set without checking further.
Obviously, the cost of searching a query in the database depends on the composition of available indices. So, we want to write $\text{Cost}(f)$ as $\text{Cost}_F(f)$ to signify that $F$ is feature set that has been indexed. We now define the redundancy of a feature $f$ with regard to $F$ on database $D$ as

$$R_D(f, F) = \frac{\text{Cost}_{F \cup f}(f)}{\text{Cost}_F(f)},$$

meaning that if the cost of searching $f$ in $D$ decreases a lot after $f$ is integrated into the index, i.e., $\text{Cost}_{F \cup f}(f)$ and thus $R_D(f, F)$ is small, then the redundancy of $f$ is low and it is worthwhile to index $f$ in addition to $F$.

One might have already noticed that $\text{Cost}_{F \cup f}(f)$ cannot be larger than $\text{Cost}_F(f)$, and thus $R_D(f, F) \leq 1$. This is because, if the index includes $f$, then the pre-computed CP-Index associated with $f$ would directly point to the true location lists of searching $f$ in $D$, which cannot be shrunk any further. In comparison, $\text{Cost}_F(f)$ refers to a larger search space as we need to use features that are: 1) already indexed in $F$, and 2) contained by $f$, to deduct $f$’s location lists. Finally, the ratio between $\text{Cost}_{F \cup f}(f)$ and $\text{Cost}_F(f)$ is a good measure of how redundant $f$ is if one wants to merge it into the indexed feature set.

A formal description of the feature selection process is given in Algorithm 5, where we sequentially go through the set of available features and only retain those non-redundant ones. Note that, since calculating the redundancy of $f$ with regard to $F$ requires all features in $F$ that are subgraphs of $f$, we want to follow a predetermined order in traversing $F_0$ so that subgraphs are always visited before supergraphs. This could be easily achieved by checking features that
have a smaller number of edges first, because a feature with $e + 1$ edges cannot be contained in a feature with $e$ edges for sure.

As proposed in Section 4.2.2.1, in order to enhance index performances, apart from a set of features shared among all graphs in the database $D$, we want to pick some graph-specific features for each $g_i \in D$. Interestingly, Algorithm 5 is general enough to handle dual feature generation. The only difference is instead of starting from the empty set, we will use $F = F^D$ in line 1 of Algorithm 5, because graph-specific features are picked on top of shared features.

**Algorithm 5** Size-Increasing Bootstrapping

Input: Graph database $D$, Initial feature set $F_0$, The redundancy threshold $r$.
Output: A subset of selected features $F \subseteq F_0$.

Feature_Selection($D, F_0, r$) {
1: $F = \emptyset$;
2: for each $f \in F_0$ do
3: Search $f$ in $D$, obtain its location lists, with $F$ indexed;
4: Calculate $f$’s true location lists as well as $R_D(f, F)$;
5: if $R_D(f, F) < r$ then
6: $F = F \cup f$;
7: Integrate $f$ and its true location lists (i.e., CP-Index);
8: return $F$;
}
4.2.3 Empirical Studies of CP-Index

In this section, we will provide empirical evaluations of CP-Index. We have two kinds of databases with graphs of large size: a real program analysis database and a synthetic database. We note that there is no indexing algorithm dedicated to indexing a database with large-sized graphs because of the computational complexity involved. Consequently, for comparison purposes, we choose a baseline that follows the framework of traditional graph feature based indexing but conducts mining on summarized database graphs (12) to generate the indexing feature set. To be more specific, in the baseline algorithm, we first summarize each database graph for off-line feature selection and index construction, and then apply normal filtering and verification procedures (57) for on-line query processing. This method is referred to as direct indexing on summarized graphs (DI Summarization). All experiments are done on a Debian GNU/Linux server with two dual-core Xeon 3.0GHz CPUs and 16GB main memory, and the program is written in C++.

4.2.3.1 Results on Real Data

Program Analysis Data. Program dependence graphs appear in software-security applications that perform characteristic analysis of malicious programs (19). In a dependence graph, vertices are different types of system calls and edges representing the dependency relationships among them. For example, an edge with label \( y = f(x) \) between two vertices \( v_1 \) and \( v_2 \) means that the system call at \( v_1 \) assigns the variable \( x \) and the system call at \( v_2 \) uses the variable \( y \) whose value is derived from \( x \).
Graph containment search is an important analytical task to perform on such system call graphs. It can reveal latent security holes if we search the database using a known malicious pattern. Our testing data has 238 graphs, with an average size of around 300 vertices and 600 edges. In addition, following what traditional schemes did, queries are randomly drawn from the database graphs as connected subgraphs. Note that, the size of the graphs here is 15-20 times larger than those in a typical chemical compound dataset that was used to test various traditional indexing methods. Also, with 238 graphs in the database, it might seem small at a first glance. In fact, this is to keep our experiment more manageable. Given the large size of graphs, many involved graph routines such as isomorphism checking would understandably be much slower; moreover, as we show later in the scalability analysis of Figure 20, CP-Index has a linearly scalable trend, which means that experiment results achieved on a smaller database is representative, as well.

We set parameters in our algorithm as follows. For mining, we summarize each graph to 20 vertices, i.e., the compression ratio is about $\frac{300}{20} = 15$, and $\minsup$ is set to 50%. To avoid pattern explosion, we only mine features up to $\maxL = 4$ edges, and later we shall have an experiment for the synthetic data (Figure 22(b)) to show how this parameter $\maxL$ would impact actual indexing performance. Finally, the redundancy thresholds is $r = 0.5$.

Figure 16(a) and Figure 16(b) well demonstrate the usefulness of CP-Index. We generate five groups of queries, with query size ranging from 20 to 60 (in terms of number of edges), which is drawn on the $x$-axis. We then process all these queries, and report the average candidate answer set size for each of the groups on the $y$-axis. Since it represents the number of time-consuming
subgraph verification tests that need to be executed, we have chosen to use the candidate set size for performance gauging. To set up an objective standard concerning the index pruning power, we also draw a line with label “Actual Match” in the figures, which displays the number of true answer graphs that really contain the input query. Since no index could/should prune away any true answer graph, the “Actual Match” line is a firm lowerbound for all indexing schemes.

We divide all queries into two categories: high support and low support. If a query has an “actual match” number that is greater than 5, then it belongs to the high support category. Figure 16(a) and Figure 16(b) display the candidate answer set size for high support and low support queries, respectively. As can be clearly seen, even with shared features only, CP-Index is already much better than DI Summarization; if we further use both types of features, then the candidate set achieved by CP-Index becomes nearly indistinguishable from the true answer
set. This is regardless of whether the query is small or large and whether the query is frequent or not. It proves the comparative advantage of leveraging contact preservation properties and building CP-Indices, as the pruning power of DI Summarization methods severely degrades in the large graph scenario.

Figure 17 depicts the query performance in a different way, where $x$-axis is the true query answer set size, i.e., its support. Still, we could see a much better performance of CP-Index, lying very close to the theoretical optimum. The gap between CP-Index and DI Summarization narrows at the right end of the picture, mainly due to a constant increase of the “actual match” number, which is the best our CP-Index can possibly achieve.

Figure 17. Performance on query support levels
Directly watching response time measured in seconds could be a more intuitive way to understand the query answering cost of searching on large graphs. So let us take a look at Figure 18, where the average query response time for each group of queries is drawn against its size. Note that, one may think that the several hundred seconds on y-axis is a little too large; however, considering that at least for each of the “actual match” we still need to run an isomorphism test for verification, it does not sound a big surprise since one might need tens of seconds to verify a large, complex graph. As can be seen, with DI Summarization, queries will take much longer to complete. Also, the gap between shared features only and dual features is more prominent here than in Figures 16(a) and 16(b). For example, when the query size is 20, indexing graph-specific features reduces the response time from 270 seconds to 45 seconds. Considering that only 34% more space is needed (1182 shared features, 405 graph-specific features), we can see that dual feature generation is a very cost-effective way to promote index performance.

Lastly, we focus on the sensitivity to redundancy threshold $r$ used in our size-increasing bootstrapping and the result is depicted in Figure 19. Here we set the query size to be 20. Figure 19(a) illustrates the impact of $r$ on the number of output index features. As expected, the number of features decreases monotonously while $r$ becomes smaller. When $r$ is 1, the output set is exactly the same as the one we could get by performing dual feature generation step without any selection. On the other hand, as showed in 19(b), the size of the candidate set after filtering gradually increases when $r$ decreases. It is a natural trade-off between the performance and the cost. However, the result of CP-Index is very close to the true answer set.
anyway no matter how small $r$ is. For example, when $r$ decreases from 1 to 0.6, the feature set size reduces almost six folds from 18,653 to 2,745, but the candidate set size remains nearly the same.

4.2.3.2 More Results on Synthetic Data

**Synthetic Generator Description.** The synthetic graph generator follows a similar mechanism as the one used to generate itemset transactions, where we can set the number of graphs ($D$), average size of graphs ($T$), number of seed patterns ($L$), average size of seed patterns ($I$) and number of distinct vertex/edge labels ($V/E$). To begin with, a set of $L$ seed patterns are generated randomly, whose size is determined by a Poisson distribution with mean $I$; then, seeds are randomly selected and inserted into a graph one by one until the graph reaches its size, which is the realization of another Poisson variable with mean $T$. Due to lack of space, we refer interested readers to (40) for further details.
The synthetic dataset we take is D200T400L100I50V4E1, i.e., 200 transactions with 400 vertices on average, which are generated by 100 seed patterns of average size 50; the number of possible vertex/edge labels is set to 4/1. For scalability and index size tests in Figure 20 and Figure 21, we vary $|D|$, the number of graphs in the database, and generate multiple datasets $D(|D|)T400L100I50V4E1$. Queries are generated similarly as in the real data case. Algorithm parameters: Each graph is summarized to 20 vertices, i.e., the compression ratio is $\frac{400}{20} = 20$; $min\_sup = 70\%$, which is a bit higher than that in the real data, because the synthetic data generator produces graphs based on a pool of seed patterns, thus tending to have higher similarities among database entries; maximum indexed feature length is $maxL = 5$; redundancy threshold is $r = 0.5$.

The scalability of CP-Index is shown in Figure 20. We vary the database size, i.e., the number of graphs inside, from 200, 400, $\ldots$, to 1,000, and measure the respective index construction
time. As demonstrated, the implementation is highly efficient and our method is linearly scalable with regard to the data size. This is in strong contrast to state-of-the-art graph indexing methods. We tested gIndex, due to the mining bottleneck on large graphs, it cannot finish in one day.

Figure 21 displays the index size (in terms of number of features being indexed). The same experiment setting as testing scalability in Figure 20 is used: We vary the number of graphs from 200 to 1,000, and see how the index size responds. The observation is that the index size grows sub-linearly, definitely a desirable property when dealing with a large graph database. Also, we could see that the gap between “Dual Features” and “Shared Features Only” is not very large, meaning that the addition of graph-specific features does not cost too much index space indeed.
Figure 21. Index size

Figure 22(a) shows the performance of CP-Index on synthetic data. Trends depicted here are quite similar to those in the corresponding Figure 17 of real data empirical studies. We also tested other datasets generated with different parameter settings and obtained similar results.

Fixing all other parameters, in Figure 22(b), we want to examine the impact of \( \text{maxL} \), maximum indexed feature size, on the performance. Here, \( x \)-axis is \( \text{maxL} \), \( y \)-axis is the candidate set size, and the queries we use have 40 edges. Both the “CP-Index” and “Di Summarization” lines have a downward trend because the index pruning power increases when more features are considered. The “Actual Match” line is horizontal. Interestingly, the performance of CP-Index seems to have almost reached its peak at \( \text{maxL} = 5 \), as features with at most 5 edges have provided enough index coverage, without resorting to larger ones. Even at \( \text{maxL} = 4 \), CP-Index is already very close to “Actual Match”, while “Di Summarization” lags behind by
a large gap. This suggests from yet another angle the stronger pruning power of CP-Index and contact preservation.

4.3 Graph Location Search

Nowadays, the surge of big and complex structured data in the form of large-scale information networks puts new challenges to the research community for advanced searching and indexing mechanisms. As stated in previous section, feature-based indexing schemes that leverage frequent patterns as indexing features were shown to be effective on graph containment search with a transaction database of small to medium sized graphs. However, graph location search focuses on exploring the internal structures of a graph, it is usually performed on a single large network where the inside is in general complex. There has been much less research on the problem of graph location search (33; 65; 66; 67). Intuitively, we would like to build an index and again leverage small frequent fragments especially frequent subgraphs as indexing features,
because structurally they are more expressive and have stronger pruning power. However, due to the complicated internal structure, the algorithms mentioned above only consider direct neighborhood information or compressed structure of shortest paths. Moreover, the previously proposed feature-based methods for graph containment search do not apply directly on this new scenario either, because

1. Index building starts with mining frequent subgraphs from the underlying graph dataset as indexing features. Unfortunately, for the subject of location search, mining a large-sized network is in general a very difficult problem, due to the NP-hardness of subgraph isomorphism, which is the core routine of any frequent subgraph mining algorithm.

2. Storing the index to facilitate search on large network is usually very expensive, since we need to record the full list of occurrences for each indexing feature. Unfortunately, as it becomes large, a network may contain an explosive number of embeddings that are isomorphic to any indexing feature. It becomes unaffordable to store all possible locations of each indexing feature. This is referred to as the pattern recurrence curse, which is especially severe in a large network with a small number of labels such as protein networks (48).

The storage issue invoked by the pattern recurrence curse also occurs in other application like querying of RDF graph which is very popular in the research of graph search (60; 34). An RDF graph is constructed from millions of RDF tuples \((s, p, v)\) where \(s\) is a subject, \(p\) is a property and \(v\) is a value. In common RDF search setting, each vertex is labeled with its distinct value so that there is no structure replication at all for any feature which makes
graph search on RDF graphs doable. However, note that each node also has a subject, if the graph is generalized to the subject level, it will dramatically shrink the set of node labels in the graph and thus introduce a huge number of structure replications. Consequently, the index construction on the generalized RDF graph would still suffer from the pattern recurrence curse.

The two problems were not issues before when building index for containment search on a graph database of small graphs because 1) frequent subgraph mining is much easier on a set of small graphs, and 2) index storage is not explosive for containment search, since for each feature, we only need to track the ID list of graph transactions containing it, with linear (instead of exponential) space requirement.

Observing that it is much easier to do everything on graph databases and in the containment search way, in this section, we propose an integrated dynamic-and-static indexing (DS-Index) scheme which can build a best-of-breed feature-based index on a large information network and let it work in the most suitable environment. The overall roadmap is illustrated in Figure 23. As can be seen, firstly we use an \textit{r-enhanced partitioning} strategy to transform a large information network into a graph database by partitioning the network and also extending each partition. In this way, a best-of-breed containment search index can thus be built on the resulted graph database $D$. Also, since each feature is only associated with a list of transaction IDs instead of full location list, the static index bypasses the \textit{pattern recurrence curse} and is very storage efficient. Secondly, we utilize index built to answer queries issued against the original network in a \textit{decompose-and-recombine} manner. When an on-line query $q$ comes, we decompose it according to a so-called \textit{r-preservation principle} which guarantees that each fragment appears
in at least one enhanced partition. Then we search for each fragment using the static index for $D$. The resulting location dynamically indexed on the fly provides information that is further assembled with little effort to pinpoint the actual locations of $q$ in the original network $G$.

Among the steps above, mining frequent subgraphs and using them as indexing features are no longer problems on the transformed graph database, given the amount of past research that is present (47; 57; 49; 15; 68; 53). There are questions regarding the other steps, though. To start with, direct partitioning of the network graph does not work, as it will break many internal structures of the network. To remedy for this, we utilize a novel $r$-enhancement partition which
expands each graph partition to an extent of $r$ so that the boundary regions will have their structure information reflected in at least one of the overlapped partitions, which is referred to as the $r$-preservation principle. In order to take advantage of it, an $r$-decomposition strategy of queries into features is further proposed: We first decompose the query into fragments to make each fragment searchable using the static index on the transformed database. With the search results dynamically obtained on the fly, i.e., the full list of occurrences of all fragments, they are recombined together so the original location search query can be solved efficiently. As we specifically make the decomposed query fragments highly selective, it is efficient to search them via index and recombine the results thereafter.

To summarize, we make the following contributions towards efficient graph location search on large networks in this study.

- Unlike traditional static indexing method, we propose a composite dynamic-and-static indexing framework referred to as DS-Index which combines dynamic index and static index together to achieve the goal of searching on a single large network. In particular, static index is built with a transformed database, and during query processing, dynamic indexing information is further utilized.

- Targeting the deficiencies in handling index construction and storage on a large network caused by the pattern recurrence curse, we adopt an $r$-enhanced partition approach to transform it into a graph database and build a frequent-subgraph-based index on that instead. By doing this, we compress full location list of features into inverted list of transaction IDs, so the storage required to store the index structure can be very small.
• An $r$-decomposition querying mechanism is proposed to utilize the transformed database index thus built. A dynamic on-the-fly indexing capability is provided: If we think of decomposed query fragments as some kind of intermediate units, then this is like picking high-selectivity indexing features specifically useful for a particular query and constructing location search indices on-the-fly, which takes advantage of such features’ high efficiency without concerning about their low coverage as we mentioned above.

• We present comprehensive empirical studies on both real and synthetic datasets. Our experimental results demonstrate that DS-Index outperforms state-of-the-art graph search methods. Moreover, it exhibits excellent space efficiency and pruning power.

4.3.1 Feature-Based Indexing

In Section 4.3.1.1, we give a brief recap about the traditional feature-based indexing method for containment search on a graph transaction database. To accommodate location search for large networks, we need a little more extension to the original scheme, which will be covered in Section 4.3.1.2.

4.3.1.1 Graph Containment Search

In graph containment search, during off-line index construction, a set of indexing features are selected carefully. For each feature $f$, we build an index which points to those database entries that contain $f$ as a subgraph. When an on-line query $q$ is issued, the indices provide a possible shortcut for query result indication: A graph $g_i$ in the database can be safely pruned without performing any subgraph containment checking, if feature $f$ is contained in $q$ ($f \subseteq q$) but is not contained in $g_i$ ($f \not\subseteq g_i$). This effectively transforms the query processing procedure into two
steps: 1) candidate pruning by index manipulation, and 2) result verification on the remaining candidates, whereas only in the second step do we need to execute a subgraph isomorphism routine of NP-complete complexity(55). Suppose a good set of features are picked, the feature-based scheme described above can save a substantial amount of work involved in graph search, resulting in a significant query processing speed-up(57).

4.3.1.2 Graph Location Search

To apply feature-based indexing on graph location search, for a large network $G$, the stored index of feature $f$ must contain all possible locations $f$ might exist in $G$. This is equal to recording all subgraph isomorphisms from $f$ to $G$, i.e., the index might looks like:

$$f \rightarrow \text{image}_1, \text{image}_2, \ldots$$

As we have seen in Section 4.1, an image can be represented as a sequence of vertices in the target network.

**Example 1** Looking at Figure 24, assuming all nodes share the same label, a length-2 path is a feature $f$ we index, which is drawn on the left. To make it more discernable, part of the target network $G$ is magnified, while $f$’s images within this subpart is shown below the feature. There are 14 images in total, each image $m$ is recorded as a sequence of vertices: $v_1, v_2$ and $v_3$ in $G$, indicating how $f$’s nodes: $u_1, u_2$ and $u_3$ are mapped under subgraph isomorphisms. Take the first two images for example, 8-3-9 and 9-3-8 are two different ways of mapping $u_1, u_2$ and $u_3$ to the target network, which are both recorded to guarantee the completeness of the index.
Now suppose we have a set of features $F$ indexed as above. When a query $q$ comes, similar to graph containment search, we are going to view $q$ as a collection of indexed features it contains, and then leverage information stored in the index to accelerate processing. The easiest way to consider this is in terms of decomposition: We decompose $q$ into a set of fragments \{\(q_1, q_2, \ldots, q_t\}\), where each fragment $q_j$ is indexed by some feature $f_{p_j} \in F$, i.e., $q_j \cong f_{p_j}$, for $1 \leq j \leq t$.

**Example 2** In Figure 25, a 5-node query is decomposed into two length-2 paths (1-2-3 and 3-4-5) on the right. Note that, $f_{p_j}$ might be the same as $f_{p_j'}$ for two different fragments $q_j$ and $q_{j'}$ ($j \neq j'$), e.g., the two fragments in Figure 25 actually correspond to the same indexed feature.

According to the decomposition of $q$, any image $M$ of the query can be decomposed in the same way: \(\{m_1, m_2, \ldots, m_t\}\), where each $m_j$ ($1 \leq j \leq t$) is an image of $f_{p_j}$ in the target network.
Figure 25. Decomposing a query into two fragments

Thus, we can reconstruct every possible $M$ by merging a set of feature images from each of $I(f_p) = I(q_j)$; and due to the off-line processing we have done, $I(q_j)$ is easily accessible from the index. However, for \{$m_1, m_2, \ldots, m_t$\} to really combine into a full image of the query, there are some additional constraints that need to be satisfied.

**Example 3** After decomposing Figure 25’s query into two length-2 paths, which we index in Figure 24 as a feature $f$, consider $m_1, m_2$ as two images of $f$ in the target network $G$:

1. It is not guaranteed that, in $G$, the 3rd node of $m_1$ is the same vertex as the 1st node of $m_2$; however, this is required by the query. In Figure 25, such nodes are marked by dotted lines;

2. It is not guaranteed that, in $G$, the 2nd node of $m_1$ and the 2nd node of $m_2$ has an edge in between; however, this is also required by the query. In Figure 25, such edges are marked by dotted lines as well.

For the combination of $m_1$ and $m_2$ to be a valid image of the query, the above two conditions must be enforced. Also, in order to ensure that the subgraph isomorphism found in this way is
an “injective” mapping, except the 3rd node of \( m_1 \) and the 1st node of \( m_2 \), the other 4 nodes of \( m_1 \) and \( m_2 \) need to be distinct.

Not surprisingly, we can iterate the process and join multiple features together to reconstruct \( q \) in the target network. Clearly, the constraints needed to further confine each joining step are determined by the decomposition of \( q \), i.e., by looking at how fragments collectively construct the query.

### 4.3.2 Storage Efficient Static Index Construction

With the preparation done in Section 4.3.1, we can actually see that given a feature-based index in place that records the occurrences of each feature in the large network, the location search steps followed are well-defined and should be able to execute very efficiently. However, the real difficulty remains in how we could arrive at the assumption and provide such an index available to use. As we mentioned in the introduction, initial feature set mining and index storage are the two major difficulties we are facing in index construction for location search on a large network. Apart from mining, which has been a long standing problem, Figure 24 of Section 4.3.1.2 well exemplifies how space consuming it might be to maintain the full index of an index feature \( f \)'s possible occurrences in the target network \( G \). First, we need to record node mappings so the space demand is proportional to the size of \( f \). Second, a feature’s images in the network could be overlapping substantially (with the wildest case being automorphism, i.e., the same set of vertices in \( G \) can provide more than one subgraph isomorphic mapping from \( f \) to \( G \)), and it greatly increases the resource requirement. Actually, the space needed to store all possible images of them in the graph would be horrendous. Unfortunately, given
the complex internal structure of large information networks, this pattern recurrence curse is inevitable, and is especially severe in the network with a small number of labels.

Aspiring for the same level of success for containment queries, we naturally ask the following question: Can we somehow adapt the tough problem setting of location search on a large graph and leverage the high efficiency of existing containment search indices on small graphs? To feed above indexing methods with desirable data, we decide to break a large graph under consideration into many small pieces, and each of the pieces will become a transaction of a graph database on which a traditional feature-based index for containment search can be built. Now, the off-line index construction difficulties have been fully bypassed, with the ease of building a relevant index on small-sized graphs as we mentioned in the introduction. Ultimately, our proposed transformation will be a successful one if we can show how on-line queries might get efficiently executed given the information stored in this alternative index.

So there are a few issues we still need to resolve. In the whole scenario of Figure 23, with the part dealing with containment search indexing and querying well researched (we can choose a best-of-breed containment index from the state of the art for our purpose), two major questions remain.

First of all, the containment search index only tells us whether or not the given query appears in a partition (i.e., a database transaction), which cannot give specific locations as required by the location search. This limitation is not very hard to fix, though. Remembering that when processing queries for containment search, a subgraph isomorphism testing phase has to be carried out anyway against a few plausible candidates to confirm if they really contain
the query. We can slightly modify the testing routine and make it output the corresponding set of target graph vertices whenever an exact match of the query is encountered. Each of these vertex sets pinpoints to a particular location where the query appears, and we are getting this almost for free for graph location search, as subgraph isomorphism testing has to be executed anyways for containment search.

Secondly, partitioning breaks part of the internal structures of the original information network, meaning that query result on the partitioned version of data can be incomplete. Think of an exact match of the query in the non-partitioned graph, if the border line between partitions happens to pass through and cut a few edges away from it, then the match would disappear in all related partitions. To recover the loss, we choose to enhance what is stored in each partition, so that all query occurrences can be obtained by looking at the enhanced version of the partitions. Our way of enhancing the resulted graphs obtained from data partitioning is called $r$-enhanced partitioning as mentioned previously and formally outlined below. Before that, we also mathematically define the partitions of a network as a prerequisite.

**Definition 10** To partition a network $G$, we distribute $G$’s vertices $V(G)$ into disjoint sets: $V(G) = V_1(G) \cup V_2(G) \cup \cdots \cup V_n(G)$, and for each $V_i(G), i = 1, 2, \ldots, n$, a graph $g_i$ is defined with $V_i(G)$ being its vertex set and all edges in $G$ with both ends in $V_i(G)$ comprising its edge set, i.e., $g_i = (V(g_i), E(g_i))$, where $V(g_i) = V_i(G)$, and $E(g_i) = \{ e | e = (u, v) \in E(G), u, v \in V_i(G) \}$. Together, $g_1, g_2, \ldots, g_n$ form a partition of $G$, while each of them is of moderate size.

**Definition 11** To obtain a $r$-enhanced partition of $G$, we start from a usual partition of $G$: $g_1, g_2, \ldots, g_n$, and expand each $g_i$ with radius $r$, that is, all vertices in $G$ that can be reached
through a path with length at most \( r \) from some vertex \( v \in V(g_i) \) is further absorbed into \( g_i \).

Let this new vertex set be \( \hat{V}(g_i) \), which gives rise to a graph \( \hat{g}_i \) in the same way as we did in Definition 10, i.e., \( \hat{g}_i = (V(\hat{g}_i), E(\hat{g}_i)) \), where \( V(\hat{g}_i) = \hat{V}(g_i) \), and \( E(\hat{g}_i) = \{e|e = (u,v) \in E(G), u,v \in \hat{V}(g_i)\} \). Now, each \( \hat{g}_i \), \( i = 1, 2, \ldots, n \) is called an enhanced partition of the original network, and all enhanced partitions actually comprise the graph transaction database we are going to build a containment search index on: \( D = \{\hat{g}_1, \hat{g}_2, \ldots, \hat{g}_n\} \).

**Definition 12** A \( r \)-query is a query graph \( q \) whose “radius” is no more than \( r \). That is, there exists a vertex \( u \in V(q) \) such that for all vertices in \( q \), its distance to \( u \) is less than or equal to \( r \).

Intuitively, the enhancement mechanism extends each partition of the original network outward by drawing in more nodes and edges from neighboring partitions, and the parameter \( r \) is used to control the range of such extension. Now, if one occurrence of the query happens to exist at the boundary among partitions, it may no longer disappear from all partitions as the partition border cuts through, since the partition on each side has included an extended “cushion” of structures from the other side after enhancement, to make the full query occurrence available from a single partition after it is enhanced. Of course, there is an easy-to-understand caveat that the query occurrence we consider must fall in the “cushion” layer thus defined, so that by enhancing, it can still stay inside at least one partition. The thickness of this layer can be measured by \( r \), which has an impact on the query size. We summarize everything above as a so-called \( r \)-preservation principle which claims that a \( r \)-query should be contained by at least one of the \( r \)-enhanced partition. A formal theorem is given below.
Theorem 1 *r-preservation principle*

A *r*-enhanced partition is information-complete with regard to the searching of *r*-queries on top of it. Essentially, every location search result of a *r*-query *q* in the original network *G* can be retrieved from some partition \( \hat{g}_i \in D = \{ \hat{g}_1, \hat{g}_2, \ldots, \hat{g}_n \} \), helped by the containment index built for *D*.

Due to the limit of space, the proof of Theorem 1 is omitted. Basically, Theorem 1 establishes that by enhancing normal partitions of the original network to a certain extent (measured by *r*), we can actually recover all the information loss and yield a complete set of location search results for *r*-queries. However, queries are decided by end-users, and thus it might be difficult to confine their size to some “radius” threshold. To this end, we come up with an *r*-decomposition strategy of queries into features. This is exactly where the dynamic part of our DS-Index scheme comes from. We will give more details and integrate all aspects about query answering in the next section.

To summarize, our static index construction procedure starts by transforming a large network into a graph database with *r*-enhanced partition. Then, any existing containment search algorithms such as (57) can move forward to extract frequent structures as indexing feature. Instead of remembering every location required by traditional location search, each feature *f* is only associated with an inverted list of enhanced partition IDs where *f* is a subgraph. Therefore, the size of the index is proportional to the number of partitions which should be very small, whereas the pattern recurrence curse in large network would prevent storage of full list of *f*’s locations from being handled efficiently.
4.3.3 Query Answering

As discussed above, suppose we have built a containment search index on the transformed database, an \( r \)-decomposition of queries with dynamic indexing is proposed during the query answering procedure of DS-Index. Specifically, when a query \( q \) comes, the following procedure is executed.

1. **Containment Search.** If \( q \)'s radius does not exceed \( r \), use the constructed containment index to search \( q \) in the database and return partitions \( g_i \)'s that contain \( q \) as a subgraph. During the verification step of index processing that confirms \( q \in g_i \), we output each valid mapping that maps \( q \) to a subgraph of \( g_i \) which is structurally equivalent to \( q \).

2. **Query Decomposition.** If \( q \)'s radius is greater than \( r \), we need to decompose \( q \) into a collection of query fragments \( \{q_1, q_2, \ldots, q_t\} \) so that each of them is with radius less than or equal to \( r \) to be processed by the containment search index, and the fragments collectively cover all vertices and edges of the original query.

3. **Dynamic Location Search Index Construction.** For each query fragment \( q_j \) after decomposition, we pass it into step 1 and obtain the full list of locations \( q_j \) appears in the original network \( G \). Although we have not directly included \( q_j \) as an indexed feature, the above is equivalent to building a location index for each \( q_j \) on the fly, which specifically targets the given query and is ready for use in the following step.

4. **Location Index Join for Query Answering.** Following the index join procedure we outlined in Section 4.3.1.2 and using \( q_1, q_2, \ldots, q_t \) as the fragments mentioned there, we will be able to reconstruct the query \( q \) and recover all locations where it appears in \( G \).
To summarize, query fragments with radius $r$ will act as an intermediate level of indexed features, and their indexing information is dynamically obtained on-the-fly. For a general query $q$ with radius greater than $r$, the static index we have built can help build an intermediate level of index on-the-fly by providing the occurrences of $r$-query fragments in the network as search results. This index specifically adapted to $q$ will finally be utilized for query processing.

Remember that in Section 4.3.2, we pointed out that it encounters insurmountable challenges when directly building a graph location search index on the original graph due to the pattern recurrence curse. Now, with the dynamic-and-static framework, our new location index constructed in an on-line fashion is no longer inflicted by this issue: First, the task of indexing more common features with small size as static is taken over by the more space-efficient containment index; second, we are able to take advantage of the high selectivity of larger features (implying efficiency in performing search on the containment index as well as recombining results thereafter) without worrying about the index coverage, because the index dynamically constructed on the fly is adapting to queries and will work best for specific queries.

With the whole integration of $r$-enhanced partition based index construction and $r$-decomposition of query answering process being laid out, the last piece that remains to be filled into the patchwork is how the query decomposition should be done. This will be discussed in Section 4.3.3.1.

### 4.3.3.1 Query Decomposition

Given a query $q$, we propose a preprocessing step that will first determine whether $q$ is an $r$-query, i.e., whether there exists a vertex $v \in V(q)$ such that the distance from all other vertices in $q$ to $v$ is no more than $r$, and if not, decompose $q$ into pieces of $r$-queries, so the containment
search outcome of different pieces combined will collectively determine the complete set of q’s
occurrences in the original network G. Finally, we can write the total query processing cost as
the sum of the following three steps.

1. The time to decompose (if needed) q into r-queries. For a predetermined r, there are many
   ways to cover q with sub-queries whose radius is less than or equal to r. These r-queries
   might be associated with different sizes and characteristics, so we need a mechanism to
   pick a few of them.

2. The time to search each r-query in the graph transaction database resulted from network
   partitioning, using the containment index we built.

3. With the location index constructed on-the-fly after step 2, where each r-query above is
   an indexing feature, the time to join all information together and obtain the final result
   of searching q on G.

Among the three, the cost of step 1 is negligible compared to step 2 and step 3. The cost
of step 2 is proportional to the number of r-queries after decomposition, so we prefer bigger
pieces as that will save the number of r-queries needed to fully cover q; on the other hand,
for a given r, larger r-queries in general imply higher selectivity, which may also decrease the
searching time with the static index since the end result would be much smaller. In the same
way, step 3 also prefers less number of r-queries to combine information from, and if an r-query
is highly selective and thus only maps to a limited list of locations in the original network G,
then the index join phase would also accelerate since there are less possibilities to try out. Since
keeping a low query processing cost is our objective, for a given r, the most appropriate way is
to decompose $q$ into as few $r$-queries as possible and make each of them as large as possible for better selectivity in the original network. This is reminiscent of the classical set-cover problem, and we are going to apply its traditional greedy solution for finding a (hopefully) minimal set of $r$-queries that can cover the full query $q$. The whole step-by-step procedure is given in Algorithm 6.

**Algorithm 6 Query Decomposition**

Input: Query graph $q$, Partition enhancement threshold $r$.
Output: Decomposed query fragments $\{q_1, q_2, \ldots, q_t\}$.

\[
\text{Query Decomposition}(q, r) \{ \\
1: \text{for each } v \in V(q) \text{ do} \\
2: \quad \text{start breadth-first search and generate a } r\text{-query} \\
3: \quad \text{with } v \text{ being the center;} \\
4: \quad \text{put it into the fragment candidate set } FC(q); \\
5: \quad t = 1; \\
6: \text{while there are still nodes (edges) in } q \text{ do} \\
7: \quad \text{let } q_t \text{ be the fragment in } FC(q) \text{ that can cover} \\
8: \quad \text{the most number of nodes (edges) in } q; \\
9: \quad \text{remove from } q \text{ nodes and edges covered by } q_t; \\
10: \quad FC(q) = FC(q) - \{q_t\}; \\
11: \quad t = t + 1; \\
12: \}
\]

As can be seen, for a given $r$, we start by deciding the largest possible $r$-query with each of the vertex $v \in V(q)$ being the center. This is achieved by doing a breadth-first search in $q$
with \( r \)-levels. Then at each step, from the whole collection computed above, we pick an \( r \)-query that can cover the biggest fraction in the remaining part of \( q \), until there is nothing left. The most time-consuming part in Algorithm 6 is doing DFS as an initial set-up. Since query size (in terms of either \(|V(q)|\) or \(|E(q)|\)) is usually not very big, in general it will not take too much time.

### 4.3.4 Experimental Results

In this section, we will provide empirical evaluations of DS-Index. We have two kinds of large graph datasets: a real yeast protein-protein interaction network and a synthetic dataset. All experiments are done on a Debian GNU/Linux server with two dual-core Xeon 3.0GHz CPUs and 16GB main memory, and the program is written in C++.

**Yeast Protein-Protein Interaction Network.** Protein-protein interactions occur when two or more proteins bind together, often to carry out their biological function. The topological pattern of interactions is a rich source of biological functional information (48), and graph location search is an important analytical task to perform on such networks. For our study here, we use the protein-protein interaction network of *S.cerevisiae* mentioned in (48). Vertices here are different types of proteins that belong to different functional classes (i.e., their node labels), and edges here represent the interactions among them. Our *S.cerevisiae* dataset has 2,361 vertices and 7,182 edges, and all proteins are from one of 13 functional classes, which is a very small set. In addition, following what traditional schemes did, queries are randomly drawn from the database graphs as connected subgraphs. Note that, such a graph size is much bigger than those we saw in a typical chemical compound dataset (57; 68).
The whole set-up of DS-Index is as follows. We use the METIS algorithm (3) to partition network into pieces, where we can use a parameter $n$ to adjust the number of partitions (as well as average partition size) and see the impact on index performance. We set parameter $r$ to 1 as default value, while sensitivity analysis will be presented later. gIndex (57), the representative feature-based index for graph containment search, fills in as our best-of-breed indexer. For comparison in various aspects concerning index construction and query processing, we do that with SPath (67), a recently proposed indexing method on large networks with superior performance reported, while more specific settings will be discussed below as we go through different experiments.

Figure 26(a) describes the index construction time. For DS-Index, this is the total time including $r$-enhanced partitioning and index construction thereafter. The minimum support level we use in creating the static index is drawn on the $x$-axis, and the index construction time shows a upward trend as the minimum support level increases, which is easily understandable since there are more indexed features involved. SPath indexes all paths irrespective of their support in the data, and thus its time curve is a horizontal line. We can see that the time needed to construct an $r$-enhanced partitioning-based index is quite reasonable and basically resides in the same range as other alternative methods.

Figure 26(b) compares the space usage of indices. Again, DS-Index shows an upward trend while the other is horizontal. Note the log scale on the $y$-axis that displays the index size, which clearly demonstrates that DS-Index is much superior in terms of storage requirement and the problem of pattern recurrence curse is well resolved. Because the underlying containment
search index is very space efficient as we only store graph ID lists, it can usually be kept in main memory. However, SPath might need to move to disks for larger graphs, incurring further I/O costs for query processing.

Figure 26. Index construction performance (yeast)

Figures 27(a) and 27(b) are the graphs that test query performance. We generate five groups of queries, with query sizes 5, 7, 10, 12 and 15 (in terms of number of edges) indicated on the x-axis. We then search all of them via index built with a minimum support of 10%, and report the average time taken to answer queries in each of the groups on the y-axis. We divide all queries into two categories: high support and low support, and if a query has an extremely high support (greater than 10,000), it is simply ignored due to the efficiency concerns of running the experiments. Also typical searching should be looking for representative structures, so such
queries might not make sense as well. It is evident that DS-Index is consistently better than SPath, regardless of the size or frequency of the query. This proves the efficiency our adaptive index can achieve, as it picks high-selectivity indexing features specifically useful for a particular query and constructs corresponding indices on-the-fly.

Figure 27. Query response time (*yeast*)

In Figure 28, we further analyze the performance of our proposed algorithm with respect to an increasing number of partitions. To better illustrate the result, two minimum support levels are tested, namely 10% and 15%. As shown in Figure 28(a), an interesting observation is that the index construction time gradually decreases as the number of clusters increases from 100 to 500, even though a larger number of clusters indicates more transactions in the corresponding transformed graph database. This decreasing trend is mainly due to the fact that when we have
more partitions, the size of each partition becomes smaller which makes the internal structure simpler and easier to mine. This observation is further testified in Figure 28(b) where we demonstrate how the number of features varies as the number of clusters changes. Clearly, with more partitions and thus simpler structure of that, the number of indexing features also becomes smaller. Figure 28(c) depicts the query response time with respect to different number of clusters. Specifically, 500 queries of 10 edges are tested in this experiment. As we can see, there is a trade-off in determining the number of clusters. With fewer partitions, we have more features each of which requires a containment test against the query fragment. But we may do less times of verification for the fragment because of both higher pruning power and smaller number of partitions. On the other hand, with more partitions, definitely, there is less number of features to be tested. Meanwhile, the selectivity power of indexing feature is potentially lower and there are also more transaction graphs.

**Synthetic Generator Description.** The synthetic graph generator follows a similar mechanism as the one used to generate itemset transactions (8), where we can set the number of
graphs ($D$), average size of graphs ($T$), number of seed patterns ($L$), average size of seed patterns ($I$) and number of distinct vertex/edge labels ($V/E$). To begin with, a set of $L$ seed patterns are generated randomly, whose size is determined by a Poisson distribution with mean $I$; then, seeds are randomly selected and inserted into a graph one by one until the graph reaches its size, which is the realization of another Poisson variable with mean $T$. Due to lack of space, we refer interested readers to (40) for further details.

The synthetic dataset we take is $D1T(|T|)L200I10V5E1$, i.e, a single graph with $|T|$ vertices, which are generated by 200 seed patterns of average size 10; the number of possible vertex/edge labels is set to 5/1. For scalability tests, we vary $|T|$, the number of vertices, to generate multiple graphs. Algorithm parameters: Each tested graph is partitioned into a set of clusters each of which has about 10 nodes; Then all the cluster are further enhanced with $r = 1$; $min_{-sup}$ as in index feature extraction is set to be 10%.

The scalability of DS-Index compared with SPath is shown in Figures 29(a) and 29(b), respectively. We vary the size of the graph, i.e., the number of nodes inside, from 4,000, 8,000, . . . , to 64,000, and measure the respective index construction time. As demonstrated, DS-Index consistently outperforms SPath in terms of both efficiency and storage. More importantly, the method is highly efficient and linearly scalable with regard to the data size. Figure 29(b) displays the index size (in terms of memory required for indexing features). We observe that the index size remains stable as the graph size grows, definitely a desirable property when dealing with large networks. Since we only store a simple inverted list of each feature, only a very small portion of required memory is used for the inverted list part whose length grows
linearly with graph size. The major part of memory is occupied by storing the structure of index feature whose number remains stable as the graph size increases.

![Figure 29. Scalability test (synthetic dataset)](image)

We also did some experiments on synthetic data to tune the enhancement radius $r$. As in enhanced partition, each of them is expanded to a degree of $r$, so that resulted transactions are information complete to answer queries up to a certain size. Such queries are called $r$-queries, and the larger $r$ is, the more queries that can be directly answered by the index without going through the decompose-and-recombine mechanism. On the other hand, a larger $r$ will also incur more overlap and increase the total size of partitions, which count towards necessary overheads in exchange for the benefit. This trend is obvious in Table V, where $r = 1$ and $r = 2$ make some difference in terms of index construction time and space occupied. However, we can also
see that even for \( r = 2 \), the index building cost concerning these two metrics is still much better than SPath. The corresponding query performance comparison is shown in Figure 30. Both \( r = 1 \) and \( r = 2 \) are superior than SPath, but the advantage of \( r = 2 \) is limited, which seems to be suggesting that a radius of 1 is enough for our test data. Nonetheless, for less dense graphs and bigger queries, a larger \( r \) might help while limiting the accompanying overhead within a reasonable range.

<table>
<thead>
<tr>
<th></th>
<th>Construction Time (in Seconds)</th>
<th>Memory (in KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS-Index ( (r=1) )</td>
<td>0.02</td>
<td>47.2</td>
</tr>
<tr>
<td>DS-Index ( (r=2) )</td>
<td>0.08</td>
<td>139.8</td>
</tr>
<tr>
<td>SPath</td>
<td>0.59</td>
<td>448.6</td>
</tr>
</tbody>
</table>

4.4 Related Work

Graph search is a basic primitive in the management and analysis of complex structured data, which has stimulated much interest on indexing techniques from the research community. It has been discussed for both graph containment search on a database of small graphs (28; 57; 68; 15; 46) and graph location search on a single large graph (33; 65; 66; 67)
In the context of graph containment search, most of the traditional graph indexing algorithms are graph feature based, as they follow the strategy of filter-and-verification. These methods achieved great success by exploiting different types of features and various verification strategies. GraphGrep (28) is the first work of this kind, which uses paths as indexing features. gIndex (57) makes a break-through by indexing frequent subgraphs of the database (55), offering much better selectivity than paths. gIndex defines a framework representative of many later methods that use graphs as features, and provides very good performance in many different situations, according to the empirical studies of (32). Tree+Delta (68) only extracts tree fragments for indexing, since the cost of tree mining is much lower; in addition, it also generates on-demand a small delta of graph features during query processing to compensate for the limited pruning power of trees. Regarding verification strategies, FG-Index (15) introduces a verification-free concept, while SwiftIndex (46) proposes an encoding plan based on the fre-
quencies of vertices/edges, etc. to improve the efficiency of subgraph isomorphism tests. Quite unfortunately, none of the methods here is suitable for the large graph scenario, because of the problems shared by all graph feature based indexing schemes as we mentioned in the beginning. Moreover, it overlooks the inside structures of individual graphs, and thus is not applicable to the scenario of graph location search.

There are also graph indexing methods that are not filter-and-verification based. Williams et al. store all unique, induced subgraphs of the database graphs in a directed acyclic graph for query processing (52). gCode (70) computes vertex signatures according to local information and adopts spectral graph coding to encode the topology of both database graphs and queries in a numerical space. Empirical studies of (32) show that the pruning of gCode is not very powerful for sparse datasets.

Compared to the easier setting that comprises a database of small-sized graphs, there has been much less research focusing on large graph search and indexing, namely, graph location search. GraphQL (33) leverages neighborhood subgraphs and vertex profiles for global and local pruning, respectively. GADDI (65) defines NDS (neighboring discriminating substructure) distance for every neighboring vertex pair in the graph and based on that develops some inequality property for candidate elimination. Both GraphQL and GADDI uses a vertex-at-a-time mechanism when searching for a query in the graphs; essentially, this is like using single vertices as indexing features, which might not be very selective compared to the subgraph features deployed in our algorithms. In SAPPER (66), although the authors further consider noises in the graph, the index structure is still based on vertex neighborhood unit. Making a step forward
in this respect, (67) proposes a path-at-a-time query processing strategy. The low mining cost and more manageable structures all make paths very desirable indexing features. However, as we already pointed out in this chapter, feature selectivity is very important in the large graph scenario. Given their better selectivity, one might want to look at subgraph features if the mining and indexing and space complexity can be well controlled.

There are some other research works doing graph search on a particular type of graph such as RDF graph (60; 34) and dynamic social network (61), while the focus of both is not on pattern recurrence curse. (34) shares some similarity with our method by partitioning graph into small pieces. However, RDF graph is a very special kind of graph where each vertex has a distinct label. Therefore, embedding replication or pattern recurrence will not be an issue, if we are not generalizing the graph to search at the subject level, which is not considered in (34). However, the graphs we consider here only have a small set of vertex labels and the number of replications can be explosive. Moreover, the SPARQL query issued for RDF graph is generally star-shaped with a small number of nodes. When the query is of any type and becomes larger, it cannot be processed efficiently. BR-Index (61) focuses on dynamic changing social network. It assumes that the graph is empty at the very beginning and builds index incrementally as nodes and edges are inserted. Even though graph is stored into many parts, it updates parts with insertion rather than partition the graph with structural information. Consequently, the part each node belongs to totally relies on the order of inserting nodes/edges. Another major issue of these methods is that they do not take storage efficiency into consideration. For example, in BR-Index, the features are enumerated as new node/edge is inserted without any selection
criterion. Given the fact that the number of possible subgraph structures can be prohibitively large, such enumerations may lead to unaffordable storage costs.

Quite a few studies, e.g. (16), have been dedicated to the acceleration of subgraph isomorphism testing, which can definitely help the verification step of CP-Index. Besides the containment queries discussed in this study, there are works that try to handle other types of queries on graphs, e.g., keyword search (35), correlation search (36), reachability queries (17) and bounded graph simulation (25). These are all of important use and greatly enhance the primitives available for graph database management.
CHAPTER 5

CONCLUSIONS

5.1 Summary of Contributions

In this thesis, we have explored scalable mining focusing on large graphs along with its related applications. Towards this direction, we thoroughly analyzed the challenges of traditional well-solved problems in the new scenario of graphs with large size, and proposed several novel algorithms. The contributions we have made are summarized as below:

- We presented a connectivity index for massive disk-resident graphs. The goal of the approach is to create a compressed representation of the underlying graphs, and use it as an index for connectivity query processing. Even though such an approach requires multiple applications of a minimum-cut algorithm over the compressed graph, the ability to do so over a memory-resident graph is an enormous advantage for the technique. The results show that the approach is an extremely effective technique which improves the performance by orders of magnitude for the disk-resident case. At the same time, the approach provides extremely high accuracy in terms of quality.

- By a creative transformation from large database to graph, we tackled an important issue in frequent pattern mining: The difficulty to mine very long patterns in a large database. A novel top-down mining framework is proposed which starts working from the top of the pattern lattice to discover the top-k longest maximal frequent patterns. This is achieved
by mapping the database to a so-called pattern graph and performing maximal clique detection on it. We introduce a separate refinement stage to further transform these clique candidates into true maximal patterns.

- In the context of graph containment search in large graphs, we identified three challenging issues: low selectivity, inadequacy and redundancy, and proposed a new indexing technique, CP-Index, to overcome them. A new graph search framework is presented, which leverages the contact preservation properties we introduced to enhance the selectivity of small features that have been severely impacted in the large graph scenario. Also, we use this search framework and develop a size increasing bootstrapping scheme to decide a compact but rather optimized graph index. Accounting for the much bigger variabilities among different graphs due to their size, we devise a dual feature generation approach to address the inadequacy issue.

- Since the two difficulties, namely the mining deficiency and pattern recurrence curse, prevent frequent fragment from being applied in graph location search as indexing feature, we proposed an integrated dynamic-and-static indexing (DS-Index) which is proved to be both effective and storage efficient. Firstly, an r-enhanced partitioning transforms a large network into a database of small sized graphs where a best-of-breed graph containment index is constructed since it is most suitable for the scenario. Secondly, we also integrate the static index previously built with an adaptive r-decomposition query into features procedure to answer queries in a dynamic way.
5.2 Future Works

In traditional frequent pattern mining tasks, in order to prune search space and find interesting patterns, a uniform user-specified minimum support level is applied for the given database. However, there are some cases where a setting of multiple minimum support levels is more suitable instead of a uniform threshold for every single item. For example, in the shopping basket analysis of a supermarket, it is quite reasonable to see that people buy milk much more often than television. Obviously, if a uniform minimum support level is used for every item, we will either end up with a combinatorial explosion of patterns with no interest given a low minimum support level, or fail to find any interesting pattern related to television given a high minimum support level. This is where frequent pattern mining with multiple minimum support levels comes from as in (41). Unsurprisingly, even with the multiple minimum support level setting, the mining procedure still suffers from mining in a tremendously huge search space. Since we proposed the top-down framework with pattern graph, we can further explore the possibility of again using pattern graph to represent the frequency relationship among items in this scenario. Hopefully, with pattern graph, this issue can be easily tackled. To accomplish this, we may follow a very similar mining procedure as discussed for the case of the uniform threshold, but a different constraint may be applied when detecting maximal cliques since we change the problem setting.

Another direction of our future research would be further improvement and exploration in graph search and indexing, which is a significant application in the field of graph mining. For example, to improve the performance of our proposed algorithm for graph location search in
large networks, we would like to further pursue the proposed idea of utilizing a pre-computed high-coverage index to construct high-selectivity indices on-the-fly that specifically target a given query, and possibly apply it on other graph data management applications, as well. One possible direction might be the introduction of a caching mechanism, so that high-selectivity features discovered before might get reused later for other queries (may have similar characteristics) without re-computation from scratch.

Given the success of feature-based indexing and the effectiveness of frequent subgraphs as indexing features, we have already shown that frequent subgraphs still can provide high selectivity and achieve great indexing performance in the scenario of large graphs with the proposed indexing frameworks. In both graph search problems we studied, we have been largely focusing on exact graphs which are precise and complete. In recent years, there has been an increasing research interest in uncertain graphs, i.e., those graphs with noise and inaccuracy. For example, the protein-protein interaction (PPI) network which we used as test dataset involves proteins (as vertices) and interactions among them (as edges). It has been noted that most of current PPI networks are captured with noise and thus it is more appropriate to present the vertex/edge in PPI networks with some probability. It would be quite interesting to pursue our research towards the direction of graph search on such kind of graphs. Since the graph is uncertain and the definition of fragment frequency is different from that in traditional setting of exact graph, we may need a new mechanism to extract indexing feature. Also, in most of the cases, node labels have to be matched even when the graph is uncertain, thus we can explore the possibility of apply the concept of contact preservation in the context of uncertain graphs.
CITED LITERATURE


132


VITA

NAME: Yan Xie

EDUCATION:


M.S. in Computer Science, Zhejiang University, April, 2006.


PUBLICATIONS

- Yan Xie, Philip S. Yu, “Max-Clique: A Top-down Graph-based Approach to Frequent Pattern Mining”, in submission to TKDE.


• Charu Aggarwal, Yan Xie, Philip S. Yu, “Towards Community Detection in Locally Heterogeneous Networks”, *Proceedings of the 11th SIAM International Conference on Data Mining (SDM 2011)*, Mesa, Arizona, Apr 2011.

• Yan Xie, Philip S. Yu, “Max-Clique: A Top-down Graph-based Approach to Frequent Pattern Mining”, *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM 2010)*, Sydney, Australia, Dec 2010.