KeyLabel Algorithms for Keyword Search in Large Graphs

by

Yue Wang

B.Sc., Simon Fraser University, 2013
B.Eng., China Agriculture University, 2004

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Approval

Name:	Yue Wang
Degree:	Master of Science (Computer Science)
Title:	*KeyLabel Algorithms for Keyword Search in Large Graphs*
Examining Committee:	**Dr. Binay Bhattacharya** (chair) Professor

Dr. Ke Wang
Senior Supervisor
Professor

Dr. Wo-shun Luk
Supervisor
Professor

Dr. Jiangchuan Liu
Internal Examiner
Professor

Date Defended: 24 August 2015

Abstract

Graph keyword search is the process of extracting small subgraphs that contain a set of query keywords from a graph. This problem is challenging because there are many constraints, including distance constraint, keyword constraint, search time constraint, index size constraint, and memory constraint, while the size of data is inflating at a very high speed nowadays. Existing greedy algorithms guarantee good performance by sacrificing accuracy to generate approximate answers, and exact algorithms promise exact answers but require huge memory consumption for loading indices and advanced knowledge about the maximum distance constraint. We propose a new keyword search algorithm that finds exact answers with low memory consumption and without pre-defined maximum distance constraint. This algorithm builds a compact index structure offline based on a recent labeling index for shortest path queries. At the query time, it finds answers efficiently by examining a small portion of the index related to a query.

Keywords: Keyword Search; Large Graph; Indexing; Top-k Query

Acknowledgements

First and foremost, I would like to express the deepest appreciation to my advisor, professor Ke Wang, who has been supportive during my Master studies. I feel grateful for his understanding, patience and help, and I learned a lot from him in the past two years. Without his guidance and persistent help, this thesis would not have been possible.

I would also like to thank professor Ada Fu and associate Professor Raymond Chi-Wing Wong from the Hong Kong University of Science and Technology for their helpful suggestions and insightful feedback on my thesis.

I have met lots of interesting people since I moved to Vancouver. They made my life more colorful. I really appreciate their help, especially my colleagues in the Database and Data Mining lab.

Last but not least, I would like to express my gratitude to my family for their unconditional support. I would not be able to go this far without them.

Table of Contents

Approval **ii**

Abstract **iii**

Acknowledgements **iv**

Table of Contents **v**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Challenges 2

1.2 Keyword Search on RDF Data 4

1.3 Contributions 6

1.4 Outline of this thesis 6

2 Related Work **7**

2.1 RClique Algorithm 7

2.2 GDensity Algorithm 8

2.3 Hop Doubling Label Indexing 10

3 Problem Definition **12**

4 KeyLabel-Indexing **15**

4.1 Hop Doubling Label Indexing 15

4.2 Keyword Label Indexing 16

5 KeyLabel-Querying **18**

5.1 Label Retrieval 18

5.2 Depth-First Answer Search 20

5.3 Pruning Strategies 22

6 Experimental Results **23**

6.1 Tested Data Sets 23

6.2 Query Design 24

6.3 Indexing Comparison on DBLP Data 25

6.4 Querying Comparison on DBLP Data 26

6.5 Pruning Strategies Test on DBLP2 Data 28

6.6 Scalability Test on Large BTC 2009 Data 28

6.6.1 Accuracy Test on DBpedia RDF Data 29

7 Conclusion and Future Works **32**

Bibliography **34**

List of Tables

Table 2.1 Resulting 2-hop label lists of G 11

Table 3.1 Frequently used notations 12

Table 6.1 Information about data sets 23
Table 6.2 Indexing time, index size and memory usage comparison 25
Table 6.3 Querying memory usage comparison 26
Table 6.4 Indexing and querying statistics on BTC 29
Table 6.5 Accuracy test on RDF data 30

List of Figures

Figure 1.1 An undirected unweighted graph about a group of professors 1
Figure 1.2 A sample unweighted graph 3
Figure 1.3 RDF data example 4
Figure 1.4 RDF data to attributed graph conversion 5

Figure 2.1 Shortest distance histogram of node v 9
Figure 2.2 A sample weighted directed graph G 10

Figure 3.1 Another sample unweighted graph 13

Figure 4.1 (Offline) Hop Doubling Label Indexing 16
Figure 4.2 (Offline) KeyLabel Label Indexing 17

Figure 5.1 (Online) Label Retrieval 19
Figure 5.2 (Online) Depth-First Answer Search 20

Figure 6.1 Querying performance on DBLP1 27
Figure 6.2 Querying performance on DBLP2 27
Figure 6.3 Pruning strategies test on DBLP2 28
Figure 6.4 Scalability test of querying time on BTC 29

Chapter 1

Introduction

Graph keyword search finds a substructure from a graph, which could be modeled from relational databases [10, 20], XML documents [9, 11, 29], web pages [22], and social networks [15, 18], to cover a set of input keywords. Each node of the graph, with some attributes or keywords, represents a point of interest, an XML document, a web page, or a participant of social networks. Edges in such graphs could represent foreign key relationships, IDREF/ID, hyper-links, and friendships or other routes. A top-k graph keyword search is about retrieving k sets of closely connected nodes, where each set collectively covers some specific keywords, i.e., keyword constraint, by nodes within a maximum distance, i.e., distance constraint, specified in the query. For example, an answer set could be a set of related papers in a citation network that cover a specified set of topics, a set of well-connected experts in a social network that cover a set of skills, and a set of nearby point-of-interests that cover several themes. Notice that this type of search identifies query keywords, not query nodes, thus, should not be confused with nearest neighbor searches where some query nodes are specified, such as [24].

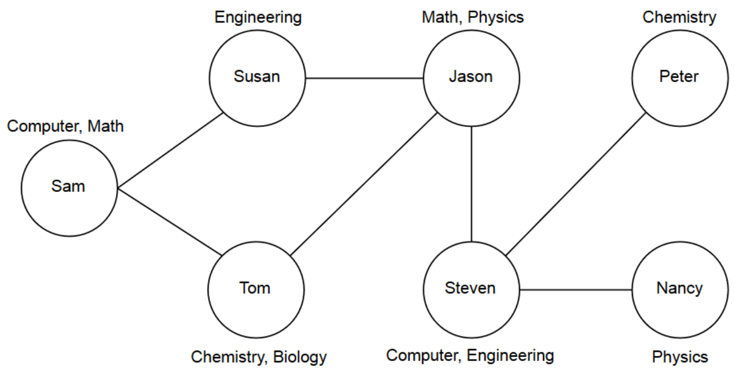


Figure 1.1: An undirected unweighted graph about a group of professors

Suppose we have an undirected unweighted graph as shown in *Figure 1.1*. Each node in the graph represents a professor. Each edge indicates the friendship between two professors.

The courses taught by a professor are labeled beside the node as its keywords. If we want to search for three best choices of teams of Chemistry and Engineering professors who can be introduced to each other via at most one other professor, we can apply a top-3 keyword search query with Chemistry and Engineering as query keywords and with distance constraint set to 2 on the graph. Then, the three sets {"Peter", "Steven"} with distance 1, {"Tom", "Steven"} with distance 2 and {"Tom", "Susan"} with distance 2 will be returned as the best three answers. The set {"Peter", "Susan"} with distance 3 won't be returned even if we set k to 4 because it violates the distance constraint.

1.1 Challenges

Most existing algorithms [2, 6, 7, 10, 14] for graph keyword search index keyword and/or connectivity information based on nodes. When a keyword query comes, full or partial indexed information need to be extracted for each related node. For large graphs, the number of nodes involved could be huge, and node-centralized indexing and query algorithms might require large memory space. Also, the random access of pages for reading index of unpredictable related nodes would lead to the reduction of query processing performance. Another challenge for graph keyword search is that the candidate answers could be anywhere in the graph, and many combinations of nodes will be examined.

One of the previous approaches to graph keyword search is finding Steiner trees to cover query keywords [2]. [7] provides a dynamic programming algorithm to find top-k minimum cost group Steiner trees, but its time complexity is exponential. In [16], the authors propose two algorithms that search for r-cliques, each of which covers all query keywords with distance between any pair of nodes no larger than r. The branch and bound algorithm gives exact answers by examining all possible candidates, which is impractical for large graphs. The polynomial delay algorithm is faster, but it's a 2-approximation where the distance between each pair of nodes in the answer produced is at most two times larger than the shortest distance constraint.

Another approach is proximity based search in the greedy manner. [18] provides a greedy but NP-hard solution called RareFirst that will return 2-approximate top-1 answer. The GDensity framework introduced in [21] improves it by indexing the histogram of the discrete distance distribution for each node, i.e., the percentage of nodes that have the same shortest distance to the node being examined, and generating exact top-k answers starting from seed nodes, which are the nodes containing least frequent query keyword, in the order of the likelihood ranking. In this paper, we only analyze and compare our algorithm with the polynomial delay algorithm, marked as RClique, and the GDensity algorithm.

RClique. This 2-approximation polynomial delay approach in [16] requires a pre-determined distance upper bound for all queries to index the linkage information from each node to all other nodes within this upper bound neighborhood. For example, if the distance upper

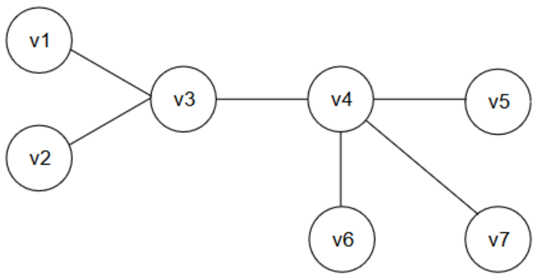


Figure 1.2: A sample unweighted graph

bound is set to 2, node v1 in *Figure 1.2* will have {(v1, 0), (v3, 1), (v2, 2), (v4, 2)} as v1's shortest distance to nodes within its 2-neighborhood, where an integer indicates a distance, and {(v1, -), (v3, v1), (v2, v3), (v4, v3)} as last node on the shortest path before reaching these neighbors. During query time, the indexed information of all nodes containing query keywords needs to retrieved for processing.

In a dense graph with a large average degree, the neighborhood of a node expands quickly as the distance increases. For big data, such full materialization of indexing, even only a limited neighborhood of each content node, which is a node that contains one or more keywords, will be indexed, might take very long time and need huge disk space to store index. Also, random access of pages of node-based index in query time will result in huge reduction of performance. Requiring knowing a maximum distance constraint leads to either a loose upper bound, or not being able to process all queries. With 2-approximation, some of the real top-k answers may violate the distance requirement and the best solutions may not be in the top-k answers.

GDensity. Same as RClique algorithm, GDensity requires to know a distance upper bound for indexing. The index size is small because it only records a histogram for each node. During query time, the search starts from seed nodes and increases the search range by 1 in each iteration. Taking node v1 in *Figure 1.2* as a seed node for example, the first iteration checks whether nodes 1 itself covers all the query keywords. If not, the search range will be increased to 1, and nodes v1 and v3 will be checked for query keyword coverage in the second iteration. The program terminates once the priority queue of answers is full.

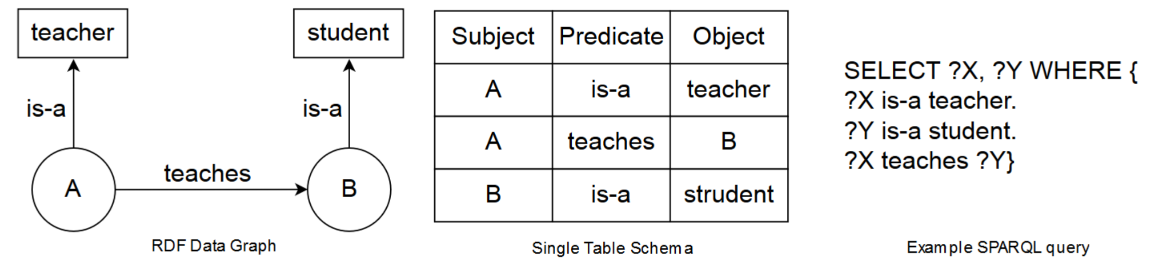
A drawback of GDensity is that the whole graph has to be loaded into memory before query processing, which is impractical for big data. The search space could expand quickly for a dense graph. In fact, the search is pruned mainly based on the least frequent query keyword and other query keywords are checked by the expensive shortest path computation. Also, GDensity might not be able to terminate early when the number of candidate answers is smaller than that of expected answers. If the graph is weighted or the distance constraint of the query is large, GDensity will waste time on re-examination of keyword coverage with the range increased by only 1 at each iteration.

To address the weaknesses of RClique and GDensity when dealing with big data, we intend to design a keyword-based algorithm, called KeyLabel algorithm, based on the recent labeling algorithm in [12] for answering shortest path queries, called Hop Doubling Label Indexing algorithm. The algorithm creates limited *2-hop label entries* for each node offline, instead of full shortest distances from each node to all other nodes. When a distance query comes, the indexed information of the related two nodes will be retrieved and used to compute the shortest distance between them efficiently. While 2-Hop Labeling algorithm considers only distance information, graph keyword search must consider both distance and keyword coverage. By associating the label entries to the keywords in a well-designed manner, our KeyLabel algorithm allows only a small portion of indexed information related to a few query keywords to be extracted during query time.

1.2 Keyword Search on RDF Data

Solutions to graph keyword search also serve to close the gap between querying structured data, which requires user knowledge about the underlying structure of the data, and the lack of such knowledge of the user. An example is the fast growing and widely used RDF data [17] designed for big data representation. The RDF - Resource Description Format - is a simple and flexible format that provides a new way to express big data by using three part statements called triples, including a subject as instance identifier, a predicate as property name, and an object as property value. Being built on web standards, RDF uses URIs as globally unique and unambiguous identifiers in the form of URLs for subjects and predicates. The triple indicates some directed relationship from its subject to its object if both of them are unique instance identifiers.

Applying RDF data structure has many advantages: 1. RDF schema standard allows brand-new properties to be added without modifying the schema, which enhances flexibility. 2. Triples can be split into any number of files with the same collective meaning, so the sharding and integration of data collections becomes easy. 3. The sharing of vocabularies makes it easier to find connections in data aggregated from disparate sources. RDF is an efficient way to store sparse datasets that don't store all the same properties for every instance of a given class.



Subject	Predicate	Object
A	is-a	teacher
A	teaches	B
B	is-a	strudent

```
SELECT ?X, ?Y WHERE {
?X is-a teacher.
?Y is-a student.
?X teaches ?Y}
```

Figure 1.3: RDF data example

The user can query such RDF data by constructing semantic queries like SPARQL [23], but this requires the user to have sufficient knowledge about the object-predicate-object structure that identifies entity-to-entity and entity-to-attribute relationships. *Figure 1.3* shows a simple RDF data graph, the related schema, and an example SPARQL query that can be used to find those triples. For queries with more complex structures, asking the user to provide such well-formatted SPARQL queries is unreasonable. Graph keyword search can help address this issue, as suggested by [26], by modeling RDF data as an attributed graph and utilizing graph keyword search to find subgraphs containing the user's interested keywords. Users can choose a query among the SPARQL queries reconstructed from the returned subgraphs and apply it on the SPARQL search engine to find related triples.

To convert the RDF tuples to an attributed graph for keyword search, we need to encode the predicate and object information based on the type of the object. The subjects in RDF data always represent entities, so we model all of them as nodes in the attributed graph. The object could be an entity or an attribute. When the object is an attribute, we add a (pk, ok, i) label to the subject node, where pk is the predicate keyword, ok is the object keyword, and i is the indicator that will be set to 0, indicating that this is an entity-to-attribute relationship. If the object is another entity, then we model it as a node and add a (pk, en, i) label to both the subject and object nodes, where pk is the predicate keyword, en is either the subject or the object node, and i is the indicator that has value 1 or 2 showing whether the entity is a subject or an object in the entity-to-entity relationship.

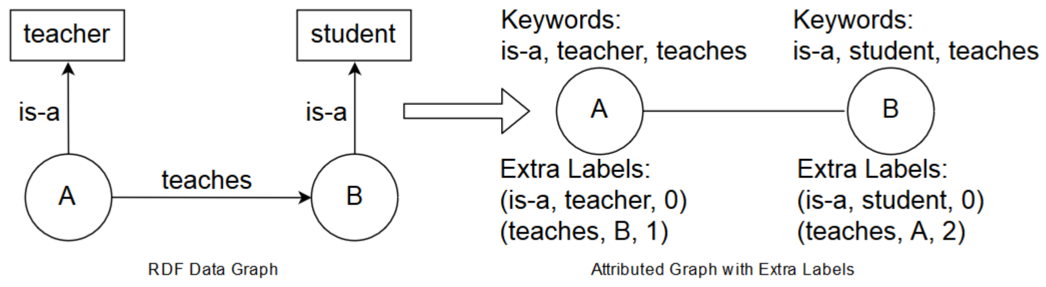


Figure 1.4: RDF data to attributed graph conversion

Figure 1.4 shows the conversion from a RDF data graph to an attributed graph with 2 nodes. After the conversion, users can simply provide a keyword search query with some keywords, {student, teacher, teaches} for example, on the attributed graph. The keyword search algorithm will return the node set {A, B} as an answer. Then, we can reconstruct the original SPARQL query as shown in *Figure 1.3* by using the query keywords {student, teacher, teaches} and the extra labels associated with each node. For example, the keyword "teacher" and label (is-a, teacher, 0) associated with node A gives us enough information to reconstruct the "?X is-a teacher" element of the SPARQL query. From the keyword "teaches" and the two labels (teaches, B, 1) and (teaches, A, 2) of nodes A and B, we can also reconstruct the "?X teaches ?Y" element.

1.3 Contributions

We design a new graph keyword search indexing and querying algorithm, called KeyLabel, to address the weaknesses of RClique and GDensity. All three algorithms have an offline indexing stage and an online querying stage. Unlike RClique-Indexing that materializes full shortest path information, KeyLabel-Indexing utilizes the index result of 2-Hop Labeling to associate label entries for nodes to related keywords. The result is a new labeling index that that can be used to support both distance constraint and key constraint, and have a much smaller size than the index computed by RClique-Indexing. Unlike GDensity-Querying that keeps the whole graph in memory or RClique-Querying that loads the whole neighborhood of content nodes in memory, KeyLabel-Querying retrieves only the label entries related to the query keywords and within the maximum distance specified, and finds exact answers with small search time and memory space usage. KeyLabel also does not require a pre-determined upper bound on distances for all queries like RClique and GDensity.

In particular, our contributions are summarized as follows: (1) We introduce a new keyword-distance centralized KeyLabel-Indexing algorithm, based on the Hop Doubling Label Indexing algorithm [12], to overcome the bottlenecks of node-centralized RClique-Indexing and GDensity-Indexing. (2) We design a fast querying algorithm, KeyLabel-Querying, to speed up query processing and reduce memory space cost at the same time as compared to RClique-Querying and GDensity-Querying. (3) We make various comparisons among these three algorithms on real-world graphs to support that our KeyLabel algorithm does outperform the other two in most cases. (4) We test the scalability of KeyLabel algorithm on large graphs. (5) We verify the efficiency and effectiveness of KeyLabel in reconstructing SPARQL queries on RDF data.

1.4 Outline of this thesis

Our method is based on the Hop Doubling Label Indexing algorithm and will be compared with RClique and GDensity algorithms, so in Chapter 2 we discuss the related work on these three algorithms. In Chapter 3, we specify definitions and formal problem statements.

The details of our KeyLabel algorithm are introduced in Chapter 4 and 5. Chapter 4 covers the KeyLabel-Indexing method with Hop Doubling Label Indexing and Keyword Label Indexing two parts. Chapter 5 illustrates the KeyLabel-Querying procedure by first retrieving indexed labels and then performing Depth-First Search for top-k answers.

In Chapter 6, we evaluate the experimental results of the comparison among RClique, GDensity and KeyLabel algorithms, the scalability test of KeyLabel algorithm, the efficiency test of two pruning strategies and the correctness test on RDF data with RClique and KeyLabel algorithms. We conclude in Chapter 7 and list several future work directions.

Chapter 2

Related Work

In this Chapter, we first discuss the details of RClique and GDensity algorithms to reveal how their mechanisms lead to their drawbacks mentioned in Section 1.1 when dealing with large and dense graphs. Then we'll look into the technique and complexity of Hop Doubling Label Indexing algorithm to explain why we apply its result as a pre-condition of our KeyLabel algorithm to overcome the shortages of RClique and GDensity algorithms.

2.1 RClique Algorithm

The RClique algorithm we talk about here is the Polynomial Delay method with 2-approximation answers described in [16]. We don't compare our algorithm with the Branch and Bound approach because its brute-force mechanism is impractical for large and/or dense graph. When a keyword search query comes, the Branch and Bound algorithm first adds content nodes that contain the first query keyword to a list, named *rList*. Then it computes the shortest distance between each node containing the second keyword and each node in *rList*. Only those node pairs that have shortest distance no more than r, which is the distance upper bound defined by the query, will be kept and used to update *rList*. The same procedure are repeated for the remaining query keywords. When the process terminates, *rList* will contain all suitable node sets that cover the query keywords without violating the distance constraint. The shortest distance between each pair of nodes can be indexed in advance to speed up the querying procedure, but the indexing takes very long time and results in extremely large index size for large and/or dense graph. Also, ranking of answers is not supported.

The 2-approximation RClique algorithm adapts Lawler's method [19] for generating the top-k answers to discrete optimization problems. The idea is to divide the search space into a number of sets, each containing all the nodes with the same query keyword. The candidate answers resides in the production of these sets. Suppose we have three keywords k_1, k_2 and k_3 and corresponding node sets S_1, S_2 and S_3. The algorithm first finds the best (approximate) ranked answer $\{v_1, v_2, v_3\}$ by applying a procedure in polynomial time in the size of nodes and the number of keywords. The detailed pseudo-code can be found in [19].

After the best global answer is found, the whole search space can be represented as the union of four sets, including $\{\{v_1\} \times \{v_2\} \times \{v_3\}\}$, $\{(S_1 - \{v_1\}) \times S_2 \times S_3\}$, $\{\{v_1\} \times (S_2 - \{v_2\}) \times S_3\}$ and $\{\{v_1\} \times \{v_2\} \times (S_3 - \{v_3\})\}$, where the first set is the best global answer. Then we can use the same procedure to find the best answer in each of the rest three sets to decide the next best global answer. Suppose the second best global answer $\{v_1', v_2', v_3'\}$ is from the set $\{(S_1 - \{v_1\}) \times S_2 \times S_3\}$. It will be further divided into four sets, including $\{\{v_1'\} \times \{v_2'\} \times \{v_3'\}\}$, $\{(S_1 - \{v_1\} - \{v_1'\}) \times S_2 \times S_3\}$, $\{\{v_1'\} \times (S_2 - \{v_2'\}) \times S_3\}$ and $\{\{v_1'\} \times \{v_2'\} \times (S_3 - \{v_3'\})\}$. The later three sets will be examined together with sets $\{\{v_1\} \times (S_2 - \{v_2\}) \times S_3\}$ and $\{\{v_1\} \times \{v_2\} \times (S_3 - \{v_3\})\}$ to find the next best global answer.

To find the best answer in each set in polynomial time, the shortest distance between each pair of nodes must be indexed in advance. In order to reduce the indexing time and index size, RClique apply *neighbor index*, which only focuses on the pairs of nodes whose shortest distance is within a diameter upper bound θ. The space complexity then can be reduced from $O(n^2)$ to $O(nm)$, where n is the number of nodes, and m is the average number of content nodes within a node's θ-neighborhood. Such simplification causes the queries with diameter constraint larger than θ can't be answered. For large and dense graphs, the neighbor index of shortest distance could still become impractical.

Other than the shortest distance indexing, RClique algorithm also requires the information of the shortest path from one content node to other content nodes within its θ-neighborhood to be indexed. For the shortest path from node v to node u, the last node right before reaching u will be recorded so that the path from v to u can be retrieved recursively later. The purpose is to generate Steiner trees instead of subgraphs to represent the top-k answers. However, such index doubles the size of index and also becomes impractical for large and dense graphs.

2.2 GDensity Algorithm

The GDensity algorithm is a proximity search algorithm inherited from a 2-approximation algorithm called RareFirst. RareFirst algorithm first finds the rarest query keyword that has the smallest frequency. For each node containing the rarest keyword, it searches for the nearest neighbors that contain the remaining query keywords. The top node set found by RareFirst must have a diameter no larger than twice of that of the real top set. If all pairwise shortest distances have been indexed in advance, which is impractical for large graphs, RareFirst could be very fast. Unlike RareFirst, GDensity algorithm finds the top-k exact node sets covering all the query keywords with a distance constraint. To find the best node sets in an efficient and accurate way, GDensity applies several mechanisms such as density indexing, likelihood ranking, partial materialization and representative nodes. GDensity also requires a diameter upper bound θ to be predefined for indexing like RClique.

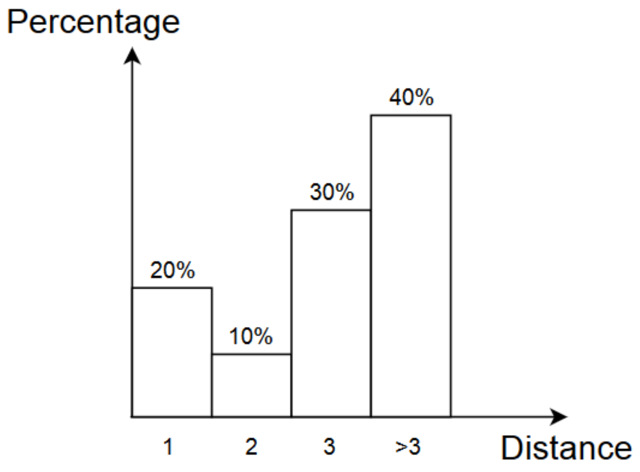


Figure 2.1: Shortest distance histogram of node v

As the precondition of likelihood ranking, density indexing records the discrete distance distribution from a node to all other nodes within its θ- neighborhood. Suppose the diameter constraint is set to 3 for a 100-node graph. If a node v can reach 20% of nodes in 1 hop, 10% of nodes in 2 hops, and 30% of nodes in 3 hops, the distance-histogram of v will be like *Figure 2.1.* The idea of building such index is that when we can reach more nodes with less hops from a content node, we have higher chance to cover all query keywords with smaller diameter. As a consequence, we can examine nodes in the order of their likelihood ranking, which is computed from the density index.

The GDensity framework consists of five components: 1. A histogram-like profile for each content node is created to record the distribution of shortest distances from the node to other nodes in its θ-neighborhood. 2. The list of seed nodes that contain the least frequent query keyword is extracted when a query comes. 3. The density index is used to do the likelihood ranking for all seed nodes, which will later be examined based on this ranking. 4. In each iteration of the progressive search, the search space centered by a seed node is increased by 1 to find qualified node sets. 5. A buffer of size k is maintained for storing answers, and the process will terminate once the buffer is full.

For large graphs, partial materialization and representative nodes are used to reduce the indexing cost. By applying random sampling, partial materialization allows the shortest distance distribution from a content node to a portion of its θ-neighbors instead of all nodes within its θ-neighborhood to be indexed to represent the approximate histogram. Representative nodes further speed up the indexing procedure by skipping the density index for a node if we have already done it for another node with similar local 1-hop topological structure. It means that if the percentage of common neighbors of two nodes exceeds a threshold, we can use the density of one node to represent that of the other node. However, even if GDensity requires small indexing time and index size for recording the shortest distance distribution, its querying part still has to load the whole graph into memory to calculate the actual shortest distances online.

2.3 Hop Doubling Label Indexing

Hop Doubling Label Indexing is a new indexing method for querying the shortest distance between two nodes in a graph. Due to large computation time and index size, most state-of-the-art in-memory indexing algorithms [3, 4, 5, 13, 25, 27, 28] can only handle small graphs. IS-Label indexing [8] and PLL scheme [1] that apply 2-hop labeling method [5] are able to deal with relatively larger graphs, but they still neither guarantee a small label size nor provide reasonable bounded complexity on the computation time. Hop Doubling Label Indexing algorithm, on the other hand, is an efficient indexing algorithm that provides bounded computation time and index size even when the given large graph and the index cannot fit in main memory.

The Hop Doubling Label Indexing algorithm constructs an incoming label list $L_{in}(v)$ and an outgoing label list $L_{out}(v)$ for every node v in a given weighted and directed graph $G = (V, E)$. Each label entry of v is a pair (p, d) where $p \in V$ and p is called a *pivot*, and d indicates the shortest distance between v and p. Let $dist(v, u)$ be the shortest distance from v to u. If $dist(v, u) \neq \infty$, we can find a label entry $(p, d1)$ from $L_{out}(v)$ and a label entry $(p, d2)$ from $L_{in}(u)$, with the same pivot p, such that $d1 + d2 = dist(v, u)$. Also, we cannot find another pivot p' such that $(p', d1') \in L_{out}(v)$, $(p', d2') \in L_{in}(u)$, and $d1' + d2' < d1 + d2$. Therefore, the shortest distance from v to u can be found by simply looking up the (p, d) label entries in v's outgoing label list $L_{out}(v)$ and u's incoming label list $L_{in}(u)$ to find a pivot p with the smallest sum of d. The label lists of all nodes in a graph form a 2-hop cover of the graph, meaning that the shortest distance between any two nodes can be calculated by two hops through an intermediate node, which is the pivot p. The 2-hop cover is minimal, so if we delete any label entry, even the trivial $(v, 0)$ label entry, we cannot answer all the distance queries.

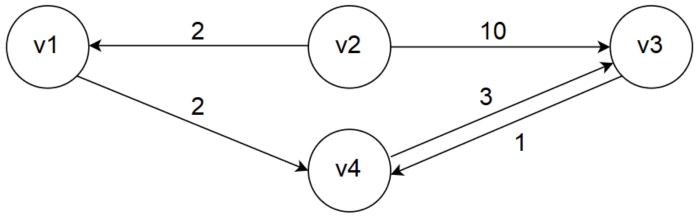


Figure 2.2: A sample weighted directed graph G

Figure 2.2 shows a sample weighted and directed graph G, and *Table 2.1* illustrates the corresponding 2-hop label lists of G computed using the Hop Doubling Label Indexing

Incoming List	Label Entries
L_{in}(v1)	(v1, 0) (v2, 2)
L_{in}(v2)	(v2, 0)
L_{in}(v3)	(v3, 0) (v2, 10) (v4, 3)
L_{in}(v4)	(v4, 0)
Outgoing List	Label Entries
L_{out}(v1)	(v1, 0) (v4, 2)
L_{out}(v2)	(v2, 0) (v4, 4)
L_{out}(v3)	(v3, 0) (v4, 1)
L_{out}(v4)	(v4, 0)

Table 2.1: Resulting 2-hop label lists of G

method. Suppose we want to find the shortest distance from v2 to v3. With L_{out}(v2)={(v2, 0) (v4, 4)} and L_{in}(v3)={(v3, 0) (v2, 10) (v4, 3)}, we can find two pivots v2 and v4 that exist in both label entry lists. For the label entries (v2, 0) and (v2, 10) with the same pivot v2, we have the sum of distance $0 + 10 = 10$. Similarly, the sum of distance $4 + 3 = 7$ corresponds to the pivot v4. Since 7<10, the shortest distance from v2 to v3 is 7, which is calculated through the intermediate node v4 on the related shortest path. Note that when we apply the Hop Doubling Label Indexing on undirected graphs, each node v has only one label list $L(v)$, where $L(v) = L_{in}(v) = L_{out}(v)$.

Hop Doubling Label Indexing algorithm was designed for point-to-point distance query in the way of fast distance computation at query time. It provides a precondition to solve all the problems of RClique and GDensity algorithms on large an/or dense graphs. Hop Doubling Label Indexing algorithm only takes the edge information of a graph as input and generates a full index with a limited number of label entries for each node. By indexing based on such result, our KeyLabel algorithm requires no limitation on the range of neighborhood to be predefined, so it's able to answer keyword search queries with any distance constraint. As proven in [12], for an unweighted graph, the computation cost of Hop Doubling Label Indexing algorithm is $O(|V|logM(|V|/M + log|V|))$, the I/O cost is $O(|V|log|V|/M \times|V|/B)$, and the index size is $O(h|V|)$, where h is a small constant, M is the memory size and B is the disk block size. It allows our KeyLabel algorithm to do indexing with reasonable memory space usage, indexing time and index size for large graphs and achieve very small memory space usage and querying time when answering a keyword search query.

Chapter 3

Problem Definition

For convenience of presentation, we focus on undirected and unweighted graph, but it's easy to modify current KeyLabel algorithm to deal with directed and/or weighted graphs. *Table 3.1* shows the frequently used notations in the following definitions.

Symbol	Description
$G = (V, E, W)$	G a node-attributed graph, where V is the node set, E is the edge set, W is the keyword set, and $W[v]$ is the set of keywords that node v contains
$Q = (QW, Dia, Tpk)$	a keyword query, where QW is the set of keywords, Dia is the diameter constraint, and Tpk is the top-k constraint
$dist(u, v)$	the shortest distance between node u and node v
$L_{2hop}[v]$	the label entries of node v in $(p, dist)$ format
$vf[w]$	the nodes of keyword w, where vf is the keyword-to-nodes inverted file generated from G

Table 3.1: Frequently used notations

Definition 1. (Diameter). Given a graph $G = (V, E, W)$ and a node set $V_i \subseteq V$, the diameter of V_i is the maximum distance among the shortest distances of all node pairs in V_i.

Definition 2. (Pairwise Total Distance). Given a graph $G = (V, E, W)$ and a node set $V_i \subseteq V$, the pairwise total distance of V_i is the sum of shortest distances of all pairs of nodes in V_i. Suppose the shortest distance between two nodes v_x and v_y is $dist(v_x, v_y)$, the pairwise total distance of a node set $V_i = \{v_1, v_2, ..., v_n\}$ is $\sum_{x=2}^{n} \sum_{y=1}^{x-1} dist(v_x, v_y)$.

Definition 3. (Minimal Cover). Given a graph $G = (V, E, W)$, a node set $V_i \subseteq V$ and a keyword set $W_i \subseteq W$, V_i is a cover of W_i if $W_i \subseteq \bigcup_{v \in V_i} W(v)$. If V_i is a cover of W_i, and no subset of V_i is a cover of W_i, then V_i is a minimal cover of W_i.

Problem 1. (Keyword Search with Diameter Ranking). Given a graph $G = (V, E, W)$ and a query $Q = (QW, Dia, Tpk)$, the diameter ranking top-k keyword search finds Tpk node sets $\{V_1, V_2, ..., V_{Tpk}\}$ with smallest diameters such that each set V_i has a diameter no larger than Dia and is a minimal cover of QW.

Problem 2. (Keyword Search with Pairwise Total Distance Ranking). Given a graph $G = (V, E, W)$ and a query $Q = (QW, Dia, Tpk)$, the pairwise total distance ranking top-k keyword search finds Tpk node sets $\{V_1, V_2, ..., V_{Tpk}\}$ with smallest pairwise total distances such that each set V_i has a diameter no larger than Dia and is a minimal cover of QW.

For top-k keyword search in graphs, the ranking of answers is mainly based on the tightness of relationships among member nodes in the answer. Diameter ranking of *Problem 1* orders matching groups by the largest distance between any two members in each group, while pairwise total distance ranking of *Problem 2* favorites groups where members have closer or tighter overall relationship to each other. Notice that if there are less than Tpk minimal covers with diameter no larger than Dia, all such minimal covers will be returned as the answer set. Sometimes, we can also apply other ranking methods, such as edge density and minimal spanning tree, or even the combination of two or more ranking mechanisms to meet specific ranking requirements. With different ranking methods, the resulting answer sets of top-k keyword search could be various.

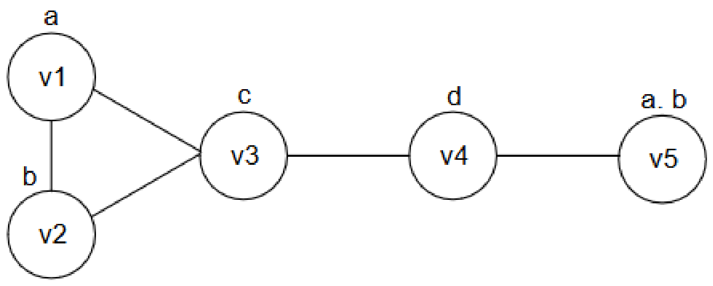


Figure 3.1: Another sample unweighted graph

Taking the graph shown in *Figure 3.1* for example where letter outsides nodes are keywords, if we have a query $Q = (\{a, b, c\}, 2, 2)$ for *Problem 1* with diameter ranking, we need to find 2 node sets having their diameters not exceeding 2. The node sets should cover the three keywords a, b and c. Candidate answers such as {v1, v3, v5} and {v2, v3, v5} fail both diameter constraint and minimal cover checks. The set {v1, v2, v3} with diameter 1 and pairwise total distance 3 and the set {v3, v5} with diameter 2 and pairwise total distance 2 will be returned as answers. If the query is used for *Problem 2*, the ranks of these two answers will exchange. Our KeyLabel algorithm even allows ranking answers based on the combination of both criteria. The ranking priority of diameter and pairwise total distance could be switched. In the following two chapters, we'll introduce the detailed mechanism of indexing and querying of KeyLabel algorithm separately.

Chapter 4

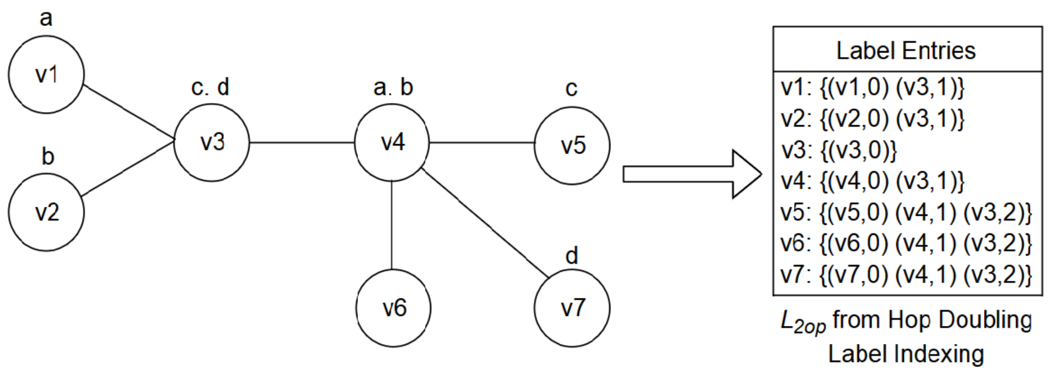
KeyLabel-Indexing

KeyLabel-Indexing, which is the offline step performed only once, combines the label entry result of Hop Doubling Label Indexing with the inverted file vf to build a new index that locates all label entries for the nodes containing a keyword w and within a given distance. With this index, at query time we can efficiently retrieve the relevant label entries for a given query. KeyLabel-Indexing procedure contains two stages, in which the graph is first indexed into label entries of nodes as described in Section 4.1, then together with inverted node lists, the labels are assigned to keywords in a new format as explained in Section 4.2. Same procedure of KeyLabel-Indexing is needed as the preparation for diameter ranking keyword search as *Problem 1*, pairwise total distance ranking as *Problem 2* and the even combined ranking discussed in Chapter 2.

4.1 Hop Doubling Label Indexing

As discussed in Section 2.1, Hop Doubling Label Indexing algorithm [12] is an indexing technique for point-to-point distance querying on weighted and directed graphs. The method constructs incoming and outgoing label lists for each node so that the shortest distance between any pair of nodes can be found later by simply looking up the two nodes' label lists. For the convenience of presentation, we focus on undirected graphs, so every node has only one label list. The label of a node v, denoted by $L_{2hop}[v]$, is a set of label entries in the format $(p, dist)$, where p is a pivot node that acts as an intermediate node for answering the part of distance queries involving v in 2 hops, and $dist$ is the shortest distance between v and p.

With Hop Doubling Label Indexing algorithm, the shortest distance between nodes v_x and v_y can be found by searching their label entry lists to see whether there are two label entries, one from each list, having the same pivot node. Instead of dealing with incoming and outgoing label entry lists on directed graphs, we find the shortest distance $dist(v_x, v_y)$ between nodes v_x and v_y by searching $L_{2hop}[v_x]$ and $L_{2hop}[v_y]$ for a pivot p with minimum *dist* sum on undirected graphs. Making the entries $(p, dist)$ in $L_{2hop}[v]$ sorted by the ID of pivot node p will speed up finding such common p in both lists in $O(|L_{2hop}[v_x]|+|L_{2hop}[v_y]|)$.



a
v1
b
v2
c. d
v3
a. b
v4
c
v5
v6
d
v7
Label Entries
v1: {(v1,0) (v3,1)}
v2: {(v2,0) (v3,1)}
v3: {(v3,0)}
v4: {(v4,0) (v3,1)}
v5: {(v5,0) (v4,1) (v3,2)}
v6: {(v6,0) (v4,1) (v3,2)}
v7: {(v7,0) (v4,1) (v3,2)}
L2op from Hop Doubling Label Indexing

Figure 4.1: (Offline) Hop Doubling Label Indexing

Figure 4.1 shows an example of applying Hop Doubling Label Indexing algorithm on a sample graph, which will be used for explaining the indexing and querying parts of our KeyLabel algorithm. The L_{2hop} label entry lists of all nodes allow us to compute the shortest distance between any pair of nodes. The average number of label entries for each node is only 2.28. With RClique algorithm, if we want to apply full index of this graph with both distance and path information, the average numbers of node-distance and node-node pairs for each node are both 6. For large and dense graphs, the difference between the index sizes of both algorithms could be huge.

Recall that both RClique and GDensity algorithms use node-based indexing. For each node v, RClique-Indexing calculates and records the distances from v to all nodes within its predefined neighborhood. When the range of neighborhood is large, or if the graph is dense, we can expect huge indexing time and index size. GDensity-Indexing only maintains distribution within v's predefined neighborhood, which results in very small index size, but it has to load the whole graph into memory before indexing. If the predefined range is large or even not available in advance, GDensity-Indexing becomes impractical, and RClique-Indexing might need to index up to $O(|V|^2)$ distances.

4.2 Keyword Label Indexing

As the second step of the offline indexing step, the Keyword Label Indexing combines the 2-hop index file lf generated by the Hop Doubling Label Indexing algorithm with the keyword-to-nodes inverted file vf so that all the label entries of nodes containing each keyword will be bounded to the keyword and recorded in the output keylabel file kf. With such keyword-based indexing, we'll be able to quickly retrieve all necessary label entries of the nodes that contain query keywords when a query arrives. The retrieved label entries usually have very small size and are sufficient for top-k keyword search queries with any diameter constraint.

To associate the label entries of a node v to a keyword k that v contains, we change the label entries of v from $(p, dist)$ to $(v, p, dist)$ so that all $(p, dist)$ label entries of each node can be easily regrouped later. We then sort all the label entries associated with the keyword k in ascending order of $dist$ before they are written into the keylabel file kf. The sorting allows us to retrieve label entries with $dist$ up to a certain diameter constraint without examining the whole label entry list.

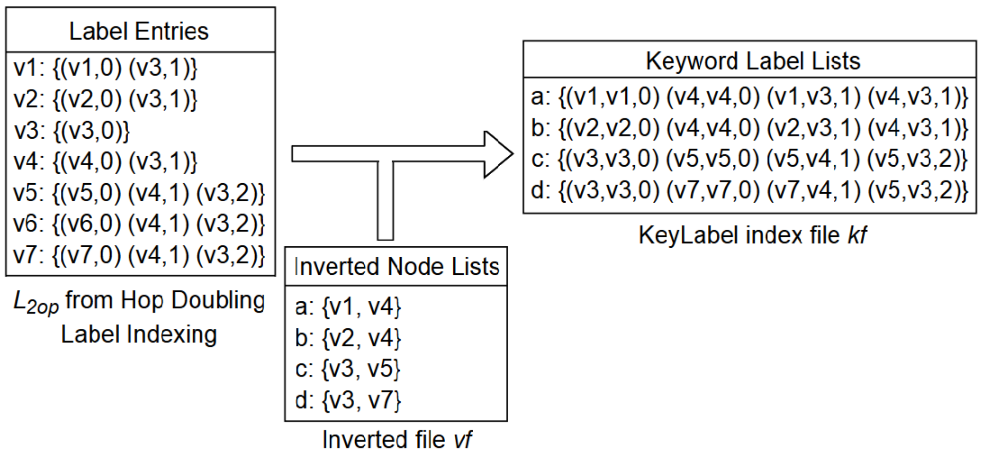

Label Entries
v1: {(v1,0) (v3,1)}
v2: {(v2,0) (v3,1)}
v3: {(v3,0)}
v4: {(v4,0) (v3,1)}
v5: {(v5,0) (v4,1) (v3,2)}
v6: {(v6,0) (v4,1) (v3,2)}
v7: {(v7,0) (v4,1) (v3,2)}
L2op from Hop Doubling Label Indexing
Inverted Node Lists
a: {v1, v4}
b: {v2, v4}
c: {v3, v5}
d: {v3, v7}
Inverted file vf
Keyword Label Lists
a: {(v1,v1,0) (v4,v4,0) (v1,v3,1) (v4,v3,1)}
b: {(v2,v2,0) (v4,v4,0) (v2,v3,1) (v4,v3,1)}
c: {(v3,v3,0) (v5,v5,0) (v5,v4,1) (v5,v3,2)}
d: {(v3,v3,0) (v7,v7,0) (v7,v4,1) (v5,v3,2)}
KeyLabel index file kf

Figure 4.2: (Offline) KeyLabel Label Indexing

Figure 4.2 illustrates the KeyLabel Label Indexing procedure on the same graph as shown in *Figure 4.1*. Taking keyword b for example, we first read the nodes {v2, v4} from the inverted file vf. Then we read label entries {(v2, 0), (v3, 1)} of node v2 and label entries {(v4, 0), (v3, 1)} of node v4 from the 2-hop label index file lf. Then we reconstruct label entries from $(p, dist)$ to $(v, p, dist)$ format to indicate which node the label entry belongs to. After we sort the new label entries based on $dist$, we'll have {(v2, v2, 0), (v4, v4, 0), (v2, v3, 1), (v4, v3, 1)} as the label list of keyword b.

For huge graphs, we might not be able to hold the whole 2-hop index file lf and/or the whole inverted file vf in memory. In this case, we need to partition files and use vf inner loop and lf as outer loop to produce the keylabel index file kf. Each time, we load the node lists of a number of keywords from vf into memory and scan lf once. When a node v contains any currently examined keyword, its label entries read from lf will be reconstructed into $(v, p, dist)$ format and added to the label list of that keyword. After scanning lf, each label list of currently examined keywords will be sorted in ascending order of $dist$ and appended into kf. We repeat the procedure until the label list of all keywords are sorted and appended into kf. We'll apply such partition and inner/outer loops in the scalability test later to cooperate with large data set.

Chapter 5

KeyLabel-Querying

The querying part of KeyLabel algorithm, KeyLabel-Querying, processes fast top-k keyword search given by a query $Q = (QW, Dia, Tpk)$ by utilizing the keyword label list index kf introduced in the previous section. KeyLabel-Querying first loads partial label entries of each query keyword from kf, reforms and regroups label entries back to the original format, and reassigns them to nodes as described in Section 5.1. Then a Depth-Frist Search for candidate answers is performed on an imagined tree structure as explained in Section 5.2. Same procedure described in Section 5.1 is needed for diameter ranking, pairwise total distance ranking and the even combined ranking described in Section 2, but in Section 5.2, we only introduce DFS answer search for *Problem 1* with diameter ranking, which can be easily modified, as later discussed at the end of the section, for *Problem 2* with pairwise total distance ranking and other problems with combined ranking criteria.

5.1 Label Retrieval

Algorithm 1: Label Retrieval

```
Data: query $Q = (QW, Dia, Tpk)$ and keylabel file $kf$
Result: 2-hop label entry lists $L'_{2hop}$
for each keyword $w$ in $QW$ do
    read a label entry $(v, p, dist)$ of $w$ from $kf$;
    while $dist$ is no more than $Dia$ do
        insert $(p, dist)$ into set $L'_{2hop}[v]$ ordered by $p$;
        read a new label entry $(v, p, dist)$ from $kf$;
    end
end
```

For a given query $Q = (QW, Dia, Tpk)$, the label retrieval stage of KeyLabel-Querying, as shown in Algorithm 1, first loads the label entries of query keywords from the indexing file kf. Since label entries have already been sorted on $dist$, we can use a sequential scan to extract only the prefix of $kf[w]$ for every keyword w in QW cut off by Dia. This is because when we later try to find shortest distance between two nodes by looking for the smallest sum of $dist$s of two label entries, one from each node's label entry list, with same pivot p, the result must exceed Dia if the $dist$ of either label entry already exceeds Dia. Unlike RClique algorithm that needs to randomly access the indexed information of all nodes containing any query keyword, and unlike GDensity algorithm that has to load the whole graph into memory, our KeyLabel algorithm only reads very limited label entry subsets of only query keywords. Typically, the memory is large enough to hold all such prefixes for a query because the number of keywords in a query is small and each content node has a limited number of label entries with $dist$ no larger than Dia.

After obtaining the list of label entries in the $(v, p, dist)$ format from kf, we change them back to the original $(p, dist)$ form to construct a new 2-hop label entry list $L'_{2hop}[v]$ of node v, and sort the list in ascending order of p. Note that $L'_{2hop}[v]$ produced in this step is the subset of label entries $(p, dist)$ for node v such that $dist <= Dia$, and v contains at least one query keyword. The sorting allows us to calculate shortest distance between two nodes by checking the intersection of pivot nodes in a linear time of their total label size. This arrangement will save huge processing time because we need to frequently perform calculation of shortest distance between two nodes in later DFS answer search.

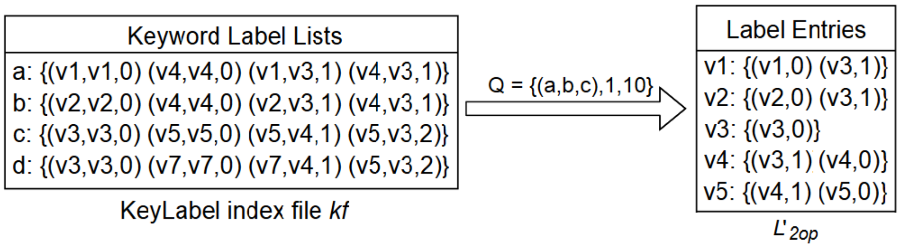

Keyword Label Lists
a: {(v1,v1,0) (v4,v4,0) (v1,v3,1) (v4,v3,1)}
b: {(v2,v2,0) (v4,v4,0) (v2,v3,1) (v4,v3,1)}
c: {(v3,v3,0) (v5,v5,0) (v5,v4,1) (v5,v3,2)}
d: {(v3,v3,0) (v7,v7,0) (v7,v4,1) (v5,v3,2)}
KeyLabel index file kf
Q = {(a,b,c),1,10}
Label Entries
v1: {(v1,0) (v3,1)}
v2: {(v2,0) (v3,1)}
v3: {(v3,0)}
v4: {(v3,1) (v4,0)}
v5: {(v4,1) (v5,0)}
L'2op

Figure 5.1: (Online) Label Retrieval

Figure 5.1 shows the Label Retrieval procedure with the same example we used in Chapter 4. $Q = \{(a, b, c), 1, 10\}$ represents a top-10 keyword search query that requests for 10 best node sets, which are minimal covers of 3 keywords (a, b, c) with the distance between any pair of nodes in the node set being no larger than 1. We first retrieve the keyword label lists of keywords (a, b, c) from the KeyLabel index file kf. Note that the retrieval of the label list of keyword c terminates before (v5, v3, 2) because any interaction with further label entries will lead to a distance larger than 1. Then, the retrieved $(v, p, dist)$ label entries will be regrouped based on v and formatted back to $(p, dist)$ form. The resulting L'_{2hop} do not contain label entry lists of v6 and v7 because none of the query keywords is contained in these two nodes.

5.2 Depth-First Answer Search

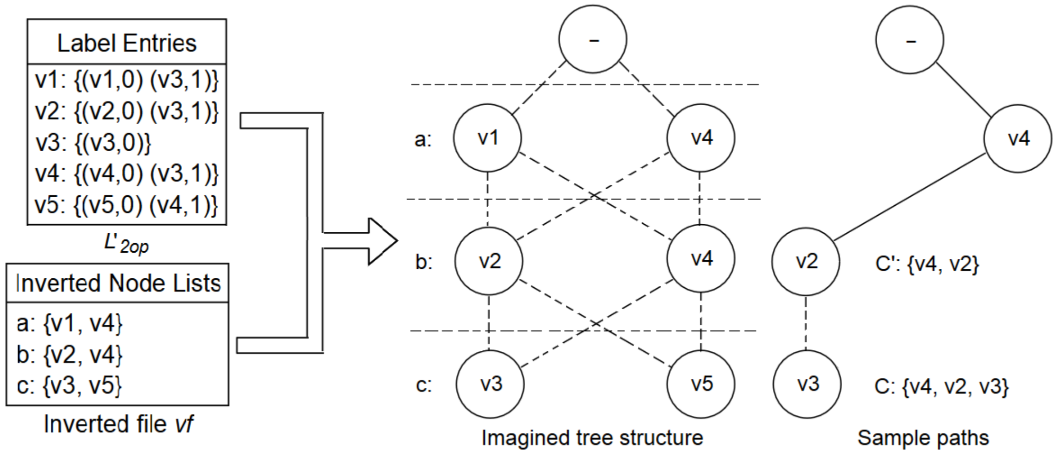


Figure 5.2: (Online) Depth-First Answer Search

Now, we can find the Tpk answers by a depth-first search for the given query $Q = (QW, Dia, Tpk)$. The node list of all the query keywords in QW read from the inverted file vf will be loaded into memory and viewed as $|QW|$ levels in an imagined tree structure with an empty root. The node list of each query keyword represents one level of the imagined tree for DFS traversal. *Figure 5.2* shows the DFS Answer Search procedure on the same example as used in Section 5.1. The imagined tree structure has an empty root and the node lists of query keywords (a, b, c) as three other levels.

In the imagined tree structure, all nodes in level t are children of every node in level t-1. Note that the linkage information between nodes in the imagined tree structure is not maintained, so we just need a little memory space for keeping the nodes and the most recently visited path. Suppose we have a current DFS visit on node v2 in level 2 and a previous DFS visit on node v4 in level 1. The path that contains most recently visited nodes in each level from level 1 to level 2 form a node set C' = (v4, v2). The score of C', i.e., the diameter and/or the pairwise total distance, is a lower bound on the score of any node set that is a super set of C'. The score is monotonic because adding a node to a node set does not decrease its score.

Suppose we have a next DFS visit on node v3 in level 3, which leads to a node set C = (v4, v2, v3). Let $DB(C)$ denote the diameter score of C, and let $PWB(C)$ denote the pairwise total distance score of C. The scores of C can be computed from that of C' with additional calculation of distance information between node v3 and any other node in C as following: $DB(C) = max\{DB(C'), dist(\text{v3, v4}), dist(\text{v3, v2})\}$ and $PWB(C) = PWB(C') + dist(\text{v3, v4}) + dist(\text{v3, v2})$. Note that we need to remove duplication of nodes from the node set before calculating the scores.

Due to the monotonicity of node set's scores, if $DB(C') > Dia$, all DFS visits that result in a super set of C' can be skipped for candidate answer examination. When a path reaches a leaf node in the imagined tree structure and the related node set passes the diameter check, we consider the node set to be a candidate answer that covers all query keywords. To check whether the node set is a minimal cover of query keywords, we remove one node each time from the set to see whether the remaining nodes can cover all the query keywords. If the candidate answer is a minimal cover of QW, it would be inserted into the top-k priority answer queue based on the diameter and/or pairwise total distance ranking.

Algorithm 2: Depth-first Answer Search

```
Data: $Q = (QW, Dia, Tpk)$, inverted file $vf$ and new label entry lists $L'_{2hop}$
Result: priority answer queue $AQ$ with diameter ranking
for every query keyword $w$ in $QW$ do
    read the node list of $w$ from $vf$;
end
imagine a tree structure with node lists of query keywords while true do
    visit a node $v$ in pre-order DFS traversal of the tree;
    if DFS traversal finished then
        break;
    end
    $C \leftarrow \emptyset$;
    add most recently visited nodes from level 1 to the level of $v$ to a node set $C$ and
    remove duplication;
    calculate $DB(C)$ using $L'_{2hop}$;
    if $DB(C) > Dia$ then
        skip all DFS visits that will lead to a super set of $C$;
        continue;
    end
    if $C$ includes a leaf node then
        if $C$ is a minimal cover of $QW$ then
            add $(C, DB(C))$ to $AQ$ ordered by $DB$;
            if $sizeof(AQ)$ is $Tpk + 1$ then
                remove the last answer in $AQ$;
            end
        end
    end
end
```

Algorithm 2 shows the workflow of the depth-first answer search of KeyLabel-Querying for *Problem 1*. It reads the node list of query keywords for DFS traversal on imagined tree structure and loads the 2-hop label list L'_{2hop} of the nodes that contain the query keywords for shortest distance calculation. A priority queue AQ of query answers with diameter ranking and up to Tpk size will be returned. At any moment of DFS search, only one node set, formed by most recently visited nodes from level 1 to current level, instead of the whole tree structure, is maintained in memory for candidate answer searching. If pairwise total distance ranking for *Problem 2* or even the combined ranking of both criteria is required, we can simply calculate both DB and PWB using L'_{2hop} and change the priority of the answer queue AQ.

5.3 Pruning Strategies

To skip more DFS visits in order to reduce the computation of shortest distances, our KeyLabel algorithm applies two pruning strategies: Infrequent First and Constraint Decreasing.

The Infrequent First strategy sorts query keywords in ascending order of the lengths of their node lists. The order is used to decide the level of the node list of each query keyword in the imagined tree structure. Suppose we have four query keywords with node list lengths 6, 9, 3 and 8. Let the imagined tree structure has 3, 6, 8 and 9 as node list lengths from level 1 to level 4. If a node set C resulted from a DFS visit on a node at level 2 already has $DB(C) > Dia$, $8 * 9 = 72$ DFS visits that would lead to a super set of C could be skipped. If we reverse the levels of query keywords to have 9, 8, 6 and 3 as lengths for the four levels, only $6 * 3 = 18$ DFS visits could be skipped in the same situation. Suppose each node visit of the DFS traversal in the imagined tree structure has the same chance to fail the diameter checking. When we apply Infrequent First to place larger node lists higher in the tree, the querying time could be reduced in that more DFS node visits are expected to be skipped and thus we can save the time of performing related computations.

The Constraint Decreasing strategy is used when we only apply the diameter constraint Dia for ranking. Suppose the priority answer queue is already full with Tpk candidate answers before we finish the DFS traversal. Let d be the diameter of the last answer in the priority answer queue. If we find a new candidate answer with a diameter no less than d, the answer will be removed right away since we just need to keep up to Tpk candidate answers. The top Tpk answers will change only when we find a new candidate answer with a diameter less than d. As a result, we can reset Dia to $d - 1$ when the priority answer queue reaches the size Tpk. The reduced diameter constraint will cause the remaining DFS visits to have more chances to violate the diameter constraint, and thus more DFS visits could be skipped to speed up the querying process.

Chapter 6

Experimental Results

In this section, we evaluate the performance on large graphs, in particular, the claims we made about the proposed method KeyLabel in Introduction. The experimental comparison of RClique, GDensity, and KeyLabel algorithms is done in Linux OS using a machine with an Intel(R) Core(TM) i5 2.67GHz CPU, 8GB RAM and 7200RPM SATA hard disk. The implementation is performed under a C++ environment. We keep all index files on disk after the offline indexing step and retrieve the related portion of the indices during the online querying step.

6.1 Tested Data Sets

We summarize the differences and purposes of tested data sets as: DBLP1 and DBLP2 represent graphs of varied degree density for comparison among three algorithms; DBLP2 data set in particular is also used in the experiment of effect of pruning strategies. BTC represents varied size graphs for evaluating scalability. RDF data set is used to evaluate the effectiveness and accuracy of constructing SPARQL queries from keyword search. GDensity currently does not support weighted or directed graphs, so all selected data sets are undirected and unweighted. *Table 6.1* summarizes the statistics about node number, edge number, average keyword per node and average degree of each data set.

Data set	Node	Edge	Avg Keyword	Avg Degree
DBLP1	317K	664K	0.344	4.19
DBLP2	317K	1M	2.27	6.62
BTC	168M	181M	0.3	2.2
RDF	894K	963K	14.68	2.15

Table 6.1: Information about data sets

DBLP1 The DBLP1 graph is modeled from the DBLP XML data (*http://dblp.uni-trier.de/xml/*). It contains the information about a collection of papers and authors as nodes and related citations and authorship as edges. Words in paper title and author names are viewed as keywords. We extract the same number of nodes as DBLP2 data set from the original XML data in order to compare the influence of average degree and average keyword on query search performance.

DBLP2 Similar to DBLP1, the SNAP DBLP2 data set is a collection of research papers in computer science field (*http://snap.stanford.edu/data/com-DBLP.html*), in which nodes represent authors, and edges represent co-authorship of two authors if they published some paper together. Authors belong to the same community if they published to the same journal or conference, which is used as a keyword. DBLP2 has a higher average degree and average keyword per node than DBLP1 data set, which increases the general difficulty of keyword search.

BTC The BTC semantic graph is converted from the Billion Triple Challenge 2009 data set, which contains web pages as nodes and links as edges that have been crawled from the Internet. The graph with 168M nodes and 361M edges has been tested in [12] for point-to-point distance querying, but it doesn't contain any keyword information. We synthetically generate a number of keywords, which is equal to 0.1% of the number of nodes in the graph. Then we sort nodes in increasing order of their degrees. The synthetically generated keywords are divided into three equally sized bins, marked as "easy", "medium" and "hard". For each keyword in the "easy" keyword bin, we randomly draw 100 nodes from the first 1/3 of the sorted nodes with lowest average degree and assign the keyword to these nodes. Similarly, we randomly draw 300 and 500 nodes from the second 1/3 and last 1/3 of the sorted nodes for each keyword in the "medium" and "hard" keyword bins. Keywords with larger average degree of nodes and higher frequency would induce larger search space and thus increase the difficulty of keyword search for all three algorithms.

RDF The RDF data provided by DBpedia organization defines a benchmark for information extracted from Wikipedia (*http://benchmark.dbpedia.org/benchmark_10.nt.bz2*) in RDF format. The original data set containing 15M tuples can be converted into 7M nodes and 12M edges. Each tuple contains subject, predicate and object. The subject is an entity node, while the object could be an entity node or a data value node. Predicates represent the relationship between subject and object. Data values are viewed as attributes/keywords of entity nodes, so only predicates that link two entity nodes are kept as edges in the graph. In order to perform the indexing of RClique algorithm, which is time consuming, within a reasonable time, we have to shrink the size and average degree of the graph by randomly removing 90% of edges and related nodes in the original RDF data set.

6.2 Query Design

We introduced two problems with different ranking methods in Chapter 2, but the experiment will focus on *Problem 1* that applies diameter ranking because pairwise total distance ranking is not supported by GDensity algorithm. Each query has four parameters, including diameter constraint, top-k constraint, query size and keyword difficulty.

Diameter (*Dia*) constraint is set according to the average diameter and largest diameter of each data set. Too large diameter will cause almost the whole graph to be searched, which is not practical. Too small diameter often leads to a small searching range and consequently results in very little search time, which is not good for comparison.

Top-k (*Tpk*) constraint limits the number of output answers of a query. With various top-k constraints, we are able to observe how the query search time varies when different numbers of answers are requested. We set top-k constraints to be 1, 10, and 100 so that the difference in search time will be obvious.

Query size (*Qsz*) is the number of keywords in a query. The query size could affect search time either ways because, on one hand, more keywords mean more combinations of candidate answers to verify, on the other hand, the frequency of a larger set of keywords decreases. We set the query size to be from 2 to 5.

Difficulty (*Dif*) of a keyword is determined by the total degree of all nodes containing that keyword. By sorting keywords based on their node lists' total degrees and evenly dividing keywords into three bins, we define three levels of difficulty, i.e., "easy", "medium", and "hard", as their difficulty. With the other parameters being equal, the query search time normally increases with the degrees of nodes because the search space is enlarged.

6.3 Indexing Comparison on DBLP Data

Dataset-Algorithm	Indexing Time (s)	Index Size (MB)	Memory Usage (MB)
DBLP1-RClique	18626	12595	176
DBLP1-GDensity	346	30	710
DBLP1-KeyLabel	600	685	1373
DBLP2-RClique	76162	48742	351
DBLP2-GDensity	503	10	782
DBLP2-KeyLabel	771	3174	2322

Table 6.2: Indexing time, index size and memory usage comparison

Table 6.2 shows indexing time, resulted index size, maximum memory space usage for offline indexing of RClique, GDensity and KeyLabel algorithms on DBLP1 and DBLP2 data sets discussed in Section 6.1. Both RClique-Indexing and GDensity-Indexing requires an upper bound θ for diameter constraint of all queries before indexing. For DBLP1 data, we set θ to 20 because the diameter of the graph is very large and small θ will later result in most queries having no answer, which is not good for comparison. For DBLP2 data, if we set θ more than 4, RClique-Indexing will take too much time to finish.

As shown in *Table 6.2*, RClique-Indexing needs more than 10 times longer time and larger disk space than the other two algorithms for indexing because it calculates and stores the linkage information from every node to all nodes within its θ-neighborhood. RClique-Indexing requires minimum memory space because it only examines the θ-neighborhood of one node each time. GDensity-Indexing has minimum indexing time and index size because it only calculates and stores the pairwise shortest distance distribution within every node's θ-neighborhood. However, When the graph is too large to be held in main memory, GDensity algorithm will become inapplicable. The indexing time of KeyLabel-Indexing is the total time used for hop doubling label indexing and keyword label list indexing. The label entries of all nodes containing each keyword will be stored in index file, so the final index size is relatively larger than that of GDensity-Indexing. However, only a very small portion of such label entries need to be retrieved at querying time.

6.4 Querying Comparison on DBLP Data

For the comparison of querying time of three algorithms, we draw three sets of curves showing their query speed with different setting for the four most important factors, including diameter constraint, top-k constraint, query size and keyword difficulty, that affect query search performance. For each data set, we test 3 *Dia*s (various for different data sets), 3 *Tpk*s, 4 *Qsz*s and 3 *Dif*s, resulting in 108 combinations of parameter setting in total. We create 10 queries for each setting. The average search time of queries that have same value of particular parameter would be used as performance criteria. For example, the average querying time for each of the "easy", "medium" and "hard" keyword difficulty is calculated from $3 * 3 * 4 * 10 = 360$ queries having same keyword difficulty but various settings for the other three parameters.

Dataset-Algorithm	Querying Memory Usage (MB)
DBLP1-RClique	3822
DBLP1-GDensity	447
DBLP1-KeyLabel	8
DBLP2-RClique	7054
DBLP2-GDensity	694
DBLP2-KeyLabel	56

Table 6.3: Querying memory usage comparison

The linkage information of nodes that contain query keywords will be loaded when query comes, so RClique-Querying has large maximum memory space usage, as shown in *Table 6.3*, for querying when query keywords have high frequency. GDensity-Querying has to keep the whole graph structure in memory when doing query, so the memory space used for query search is relatively large. KeyLabel-Querying only query keywords' label entries with distance no more than *Dia* for querying, so it requires much less memory space usage than the other two algorithms. Due to the complexity of queries, some query search might take very long time, so we setup a threshold of 100 seconds for each single query search. All query times exceed this threshold will be set to 100 seconds before averaging.

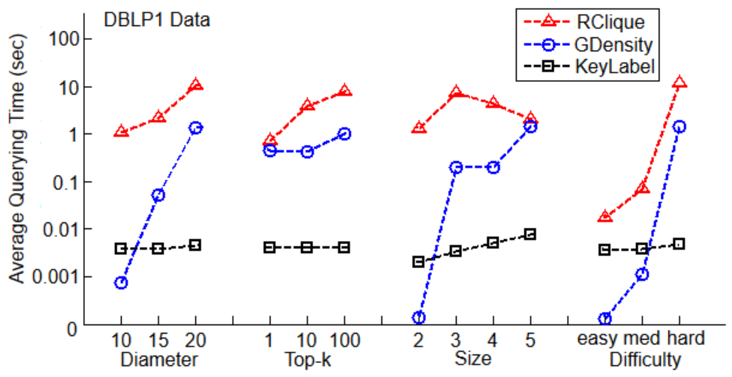


Figure 6.1: Querying performance on DBLP1

DBLP1 Data is tested with 1080 queries, among which 254 queries have non-empty answer sets and 826 queries have empty answer sets. The result is shown in *Figure 6.1*. 40 queries exceed 100 seconds with RClique-Querying, and 5 queries exceed 100 seconds with GDensity-Querying. KeyLabel-Querying can finish within 0.1 second for all the tested queries. Thus the actual querying time of RClique-Querying and GDensity-Querying is longer than shown here. The average degree and average keyword of DBLP1 data set are very low so RClique-Querying and GDensity-Querying normally have small *Dia*-neighborhoods to check and require little average query time. KeyLabel-Querying has almost same average query time for all cases because most of the time is spent on basic setup. RClique-Querying requires more querying time in all cases. The harder the parameter settings of the query are, the more obviously KeyLabel-Querying outperforms the other two.

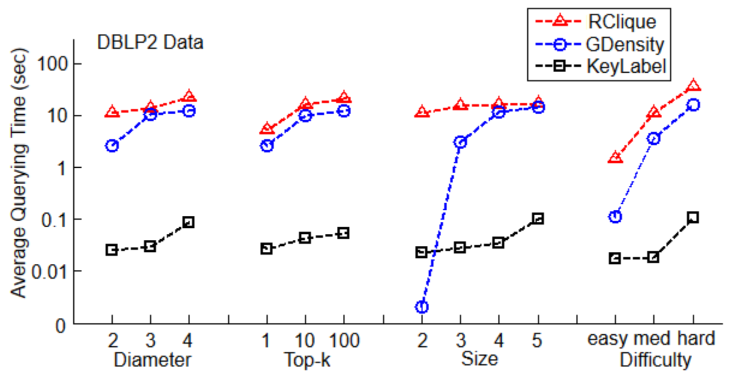


Figure 6.2: Querying performance on DBLP2

DBLP2 Data is also tested with 1080 queries, among which 673 queries have non-empty answer sets and 407 queries have empty answer sets. With RClique-Querying, 204 out of the

1080 queries take more than 100 seconds to finish. For GDensity-Querying, this number is 77. No query exceeds the threshold for KeyLabel-Querying, and the longest search time is less than 4 seconds. The result is shown in *Figure 6.2*. DBLP2 data set has higher average degree and average keyword per node than DBLP1 data set, which leads to longer search time for both RClique-Querying and GDensity-Querying. Our KeyLabel-Querying requires much less query time than GDensity-Querying except for the Qsz-2 case. It's because GDensity-Querying only needs to start from a node with one keyword and check whether any node contains the other keyword in its *Dia*-neighborhood. When *Dia* is very small, it can terminate quickly. Again, RClique-Querying requires more querying time in all cases.

6.5 Pruning Strategies Test on DBLP2 Data

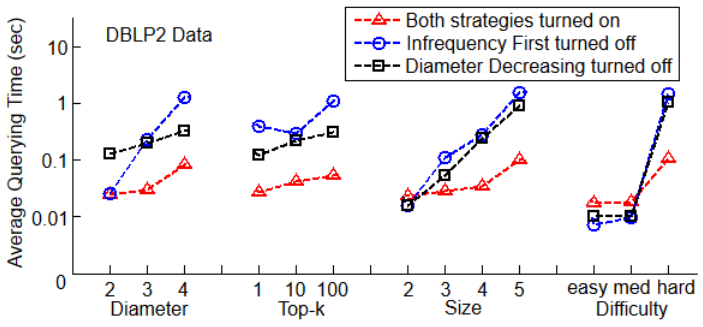


Figure 6.3: Pruning strategies test on DBLP2

To prune more candidate answers at an earlier stage, our KeyLabel-Querying applies two pruning strategies, Infrequent First and Diameter Shrinking, as described in Section IV-B. The average query time of our KeyLabel-Querying on DBLP1 data set is very small, and most of the time is spent on loading and rearranging label entries retrieved from index file, so the test result of DBLP1 data set cannot reflect the influence of pruning strategies. Figure 6.3 shows the querying performance of KeyLabel-Querying on DBLP2 data set with two pruning strategies turned on/off. Turning pruning strategies off might result in a little bit less query time than turning them on for easy settings, but as the diameter constraint, top-k constraint, query size and keyword difficulty increase, turning these two strategies on will cause a large drop in querying time.

6.6 Scalability Test on Large BTC 2009 Data

When the data set is large, RClique and GDensity might become impractical because both algorithms would take too much time for indexing or require large memory space for querying. We tested the scalability of our KeyLabel algorithm on the large BTC 2009 data set, which can not be indexed or queried by RClique and GDensity in our machine. When indexing such large data, we need to apply the partition approach of KeyLabel-Indexing discussed in Section 4.2. To verify the influence that the size of graphs has on our KeyLabel algorithm, we also test 50M-node and 100M-node data subsets extracted from the original 168M-node graph. The number of keywords for three data sets is set to 0.1% of the number of their nodes.

Node (M)	50	100	168
Indexing Time (s)	1661	12704	25601
Index Size (MB)	2867	6861	12288
Indexing Memory Usage (MB)	3934	4748	5722
Querying Memory Usage (MB)	487	950	1851

Table 6.4: Indexing and querying statistics on BTC

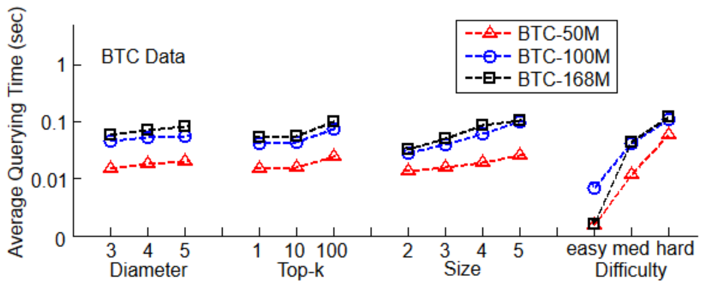


Figure 6.4: Scalability test of querying time on BTC

Table 6.4 shows the indexing time, index size, maximum indexing memory usage and maximum querying memory usage of KeyLabel algorithm on BTC data sets. Even for the largest 168M-node data set, our KeyLabel algorithm does not require too much memory space at querying time, and the indexing cost is acceptable. *Figure 6.4* shows the search time performance of KeyLabel algorithm on BTC data sets. All queries can finish within 1 second. The querying time increases almost linearly with the four factors of query setting for all three data sets, but the influence of keyword difficulty is obviously larger than that of other three factors.

6.6.1 Accuracy Test on DBpedia RDF Data

As a concrete application, we evaluate our KeyLabel algorithm for constructing SPARQL queries for the RDF data set introduced in Section 6.1 by modeling RDF data as a graph. We compare the accuracy of computed queries with the result of RClique. Note that GDensity produces exactly same answers as our KeyLabel algorithm.

We randomly select 20 subgraphs with diameter 4 and construct the related SPARQL queries from the subgraphs as our “ground truth". For each ground truth SPARQL query Q_S, we create a keyword search query Q by picking 1 to 2 meaningful keywords from each node in the subgraph induced by Q_S to simulate users' partial knowledge about Q_S. Then we apply RClique and KeyLabel algorithms to produce top-10 answers for Q and rebuild SPARQL queries Q'_S by retrieving the linkage information for the nodes in each of top-10 answers. At last, we identify whether any rebuilt SPARQL query Q'_S matches the corresponding ground truth SPARQL query Q_S and compare the highest ranked match of both algorithms. The higher rank of a match, the more useful the graph keyword search for constructing SPARQL queries for RDF data.

Queries	KL Match	RC Match
Query1	1/10	2/10
Query2	2/10	6/10
Query3	1/10	1/10
Query4	1/10	-/10
Query5	1/7	1/10
Query6	1/10	1/10
Query7	1/10	1/10
Query8	1/10	4/10
Query9	2/10	-/10
Query10	6/10	-/10
Query11	2/6	2/6
Query12	1/10	1/10
Query13	1/10	1/10
Query14	1/10	1/10
Query15	1/2	1/8
Query16	1/10	1/10
Query17	1/10	1/10
Query18	1/10	1/10
Query19	2/10	1/10
Query20	1/10	1/10

Table 6.5: Accuracy test on RDF data

RClique takes 81,069 seconds for indexing and generates 36.9GB index with neighborhood constraint θ set to 4, while KeyLabel requires 214 seconds for indexing and results in 2.2GB index. The average time for running the 20 queries using KeyLabel is 0.14 seconds. For RClique, the average querying time reaches 38.49 seconds. *Table 6.5* shows the result of accuracy test on RDF data. Taking Query10 for example, 6 means that among the 10 answers returned by the KeyLabel algorithm, the SPARQL query rebuilt from the 6th ranked answer is the first match to the corresponding ground truth SPARQL query. -, on the other hand, indicates that none of the SPARQL queries rebuilt from the 10 answers returned by the RClique algorithm is a match to the corresponding ground truth SPARQL query. Except for Query19, KeyLabel algorithm has the same or higher ranked hit than RClique.

It's obvious that our KeyLabel algorithm outperforms the RClique algorithm a lot in indexing time, index size, querying speed and accuracy. The lower ranked hit of RClique is caused by the 2-approximation of answers returned, which returns many loose answers having a large diameter.

Chapter 7

Conclusion and Future Works

The graph keyword search uses both node content information and interaction information to search for relevant parts of graphs that meet certain content and closeness requirements. Many large graph search problems can be modeled as graph keyword search problems. We studied this problem for large graphs where previous approaches suffered from serious bottlenecks. The proposed KeyLabel algorithm takes the result of Hop Doubling Label Indexing algorithm, assigns reformatted label entries to keywords to generate an index based on both keywords and distance of nodes, and performs fast keyword search with small memory space usage using that index. KeyLabel algorithm shows good performance even for very large data sets. KeyLabel is proven to outperform RClique and GDensity in most cases. The experimental result also verifies the efficiency and accuracy of KeyLabel algorithm in mapping keyword based queries to SPARQL queries on RDF data sets.

Here are some directions for future work.

Directed Version In this paper, we model all data sets into undirected graphs for comparing the efficiency of our KeyLabel algorithm with the other two algorithms. Directed version of KeyLabel algorithm can be built with minor changes. First, we apply the Hop Doubling Label Indexing algorithm to generate incoming and outgoing label entry lists for each node. Then we create an incoming list and an outgoing list of label entries for each keyword by assigning the corresponding label entry lists of nodes to the keywords they contain. After sorting the lists in ascending order of *dist*, we save them into disk. When a query comes, both incoming and outgoing label entry lists of all query keywords cut off by the distance constraint are loaded, reformatted and reassigned to nodes. During the DFS traversal in the imagined tree structure, we need to calculate the shortest distances between two nodes twice with different directions by interacting the incoming label entry list of one node and the outgoing label entry list of the other. The score calculation of candidate answer sets and pruning strategies need to be modified too.

Path Tracking Our KeyLabel algorithm adopts the 2-hop label entry list result of the Hop Doubling Label Indexing algorithm, which breaks the actual shortest path between two nodes into 2 hops. When calculating the shortest distance between two nodes, we just search for the intersection of pivots in the label entries with smallest sum of *dist* without providing the shortest path information. If the detailed shortest paths between pairs of nodes in each answer are required, we can first recursively find some intermediate nodes on the path by using the 2-hop label entry lists. Suppose we want to reconstruct the path from v1 to v2. If v3 is the pivot node found in the calculation of the shortest distance from v1 to v2, we can continue finding the pivot nodes in the shortest paths from v1 to v3 and from v3 and v2. The basic case is that either of the two end points becomes the pivot node. By such recursive function, we can find many intermediate nodes on the path from v1 to v2. We could greatly reduce total path reconstruction cost by applying some existing path finding algorithms together with these intermediate nodes on large graphs.

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440. IEEE, 2002.

[3] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. The exact distance to destination in undirected world. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 21(6):869–888, 2012.

[4] Jiefeng Cheng and Jeffrey Xu Yu. On-line exact shortest distance query processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 481–492. ACM, 2009.

[5] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[6] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S Sudarshan. Keyword search on external memory data graphs. *VLDB*, 1(1):1189–1204, 2008.

[7] Bolin Ding, J Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE 2007*, pages 836–845. IEEE, 2007.

[8] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment*, 6(6):457–468, 2013.

[9] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *SIGMOD*, pages 16–27. ACM, 2003.

[10] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316. ACM, 2007.

[11] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378. IEEE, 2003.

[12] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, James Cheng, and Yanyan Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. 7(12), 2014.

[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 445–456. ACM, 2012.

[14] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516. VLDB Endowment, 2005.

[15] Mehdi Kargar and Aijun An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, pages 985–994. ACM, 2011.

[16] Mehdi Kargar and Aijun An. Keyword search in graphs: finding r-cliques. 4(10):681–692, 2011.

[17] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): concepts and abstract syntax. 2006.

[18] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476. ACM, 2009.

[19] Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972.

[20] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914. ACM, 2008.

[21] Nan Li, Xifeng Yan, Zhen Wen, and Arijit Khan. Density index and proximity search in large graphs. In *CIKM*, pages 235–244. ACM, 2012.

[22] Wen-Syan Li, K Selçuk Candan, Quoc Vu, and Divyakant Agrawal. Retrieving and organizing web pages by âĂIJinformation unitâĂİ. In *World Wide Web*, pages 230–244. ACM, 2001.

[23] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[24] Miao Qiao, Lu Qin, Hong Cheng, Jeffrey Xu Yu, and Wentao Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[25] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *Advances in Database Technology-EDBT 2004*, pages 237–255. Springer, 2004.

[26] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, pages 405–416. IEEE, 2009.

[27] Fang Wei. Tedi: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 99–110. ACM, 2010.

[28] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 493–504. ACM, 2009.

[29] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 527–538. ACM, 2005.