QUERY PROCESSING ON ATTRIBUTED GRAPHS

by

Ka Wai (Duncan) Yung

BEng in Computer Science, University of Hong Kong, 2009

MPhil in Computing, Hong Kong Polytechnic University, 2012

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Ka Wai (Duncan) Yung

It was defended on

Aug 11, 2017

and approved by

Dr. Shi-Kuo Chang, Department of Computer Science

Dr. Alexandros Labrinidis, Department of Computer Science

Dr. Daniel Ahn, Department of Computer Science

Dr. Konstantinos Pelechrinis, School of Information Science

Dissertation Director: Dr. Shi-Kuo Chang, Department of Computer Science

QUERY PROCESSING ON ATTRIBUTED GRAPHS

Ka Wai (Duncan) Yung, PhD

University of Pittsburgh, 2017

An attributed graph is a powerful tool for modeling a variety of information networks. It is not only able to represent relationships between objects easily, but it also allows every vertex and edge to have its attributes. Hence, a lot of data, such as the web, sensor networks, biological networks, economic graphs, and social networks, are modeled as attributed graphs. Due to the popularity of attributed graphs, the study of attributed graphs has caught attentions of researchers. For example, there are studies of attributed graph OLAP, query engine, clustering, summary, constrained pattern matching query, and graph visualization, etc. However, to the best of our knowledge, the studies of topological and attribute relationships between vertices on attributed graphs have not drawn much attentions of researchers. Given the high expressive power and popularity of attributed graph, in this thesis, we define and study the processing of three new attributed graph queries, which would help users to understand the topological and attribute relationships between entities in attributed graphs. For example, a reachability query on a social network can tell whether two persons can be connected given certain attribute constraints; a reachability query on a biological network can tell whether a compound can be transformed to another compound under given chemical reaction conditions; a How-to-Reach query can tell why the answers of the above two reachability query are negative; a visualizable path summary query can offer an overall picture of topological and attribute relationship between any two vertices in attributed graphs. Except for the proposed query types in this thesis, we believe that there is still penalty of meaningful attributed graph query types that have not been proposed and studied by the database and data mining community since an attributed graph is a very rich source of infor-

mation. Through this thesis, we hope to draw people's attentions on attributed graph query processing so that more hidden information contained in attributed graphs can be queried and discovered.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Shi-Kuo Chang, for the guidance during my study. He is always available to offer advice and support on any date and under any circumstance. I am very thankful that I can have Prof. Chang as my advisor.

It is my pleasure that I can have Prof. Alexandros Labrinidis, Prof. Daniel Ahn, and Prof. Konstantinos Pelechrinis serving on my Ph.D. committee. Thank you for giving me valuable advice on various research projects. All their advice are important to the completion of this work.

Love, understanding, and support from my family while I am away from home are invaluable. Without their understanding and support, I believe that I would not be able to complete this study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1.0 INTRODUCTION

1.1 MOTIVATION

Graph is a popular data structure that can efficiently represent relationships between objects. For example, in social networks, a user is represented by a vertex and social relationship such as friendship between two users is represented by an edge. However, unlike a tuple in a relational table which has attributes, a vertex or an edge does not contain information about the object or relationship that they are representing. Fortunately, the emergence of attributed graph [1] remedies this drawback. In an attributed graph, every vertex and edge are associated with a multi-dimensional attribute. The existence of attribute on every vertex and edge makes the graph capable of efficiently representing the relationship between objects as well as containing information of objects and relationships. That makes the expressive power of attributed graph to be very attractive for modeling a variety of information networks [2, 1], such as the web, sensor networks, biological networks, economic graphs, and social networks.

Due to the popularity of attributed graphs, the study of attributed graphs has caught attentions of researchers. For example, there are studies of attributed graph OLAP [1], attributed graph query engine [2], attributed graph clustering [3, 4], attributed graph summary [5, 6, 7, 8, 9, 10, 11, 12, 13, 14] , constrained pattern matching query on attributed graph [15, 16], and graph visualization [17, 18]. However, to the best of our knowledge, the studies of topological and attribute relationships between vertices on attributed graphs have not drawn much attentions of researchers.

Answers of attributed graph path queries offer insight for understanding relationships between vertices since attributed graphs are rich in topological and attribute relationship information. For example, a social network can tell whether two persons can be connected

given certain attribute constraints; a biological network can tell whether a compound can be transformed to another compound under given chemical reaction conditions; an economic graph can tell whether an employee can be connected to an employee in rival company given certain constraint in employment history. Furthermore, by understanding the structure and attribute information between 2 entities on the attributed graph, it is also possible to tell why there is no such connection. Given that the relationships between entities on attributed graphs can be so meaningful, we believe that the studies of attributed graph path query would be an important piece in attributed graph research and building of attributed graph database systems.

1.2 MAIN CONTRIBUTIONS

In this thesis, we defined and proposed algorithms for answering three types of new attributed graph queries - attributed graph reachability query, the How-to-Reach query, and the visualizable path summary query, which offer insights for users to understand topological and attribute relationships between vertices.

C1: Efficient Reachability Processing on Attributed Graph [19] The first contribution of this thesis is an approach for effectively processing reachability query on attributed graphs.

- We introduced and defined the reachability query on attributed graph problem. Based on this definition, we developed our approach in a 2-level storage framework, which stores graph topology in primary storage (i.e. faster and smaller capacity, e.g. DRAM, PCM, local storage in distributed system) and attributes in the secondary storage (i.e. slower but larger capacity, e.g. magnetic disk, SSD, remote storage in distributed system).
- We proposed a new constraint verification approach which takes the advantage of a 'perfect' hash function [20, 21] for compressing a multi-dimensional attribute into a unique hash value. Such a compressed hash value always only requires a constant primary storage space (no matter of attribute dimension and domain size) and is sufficient to represent a multi-dimensional attribute such that prevalent graph traversal al-

gorithms (BFS, DFS, A^* with Landmark [22]) is not required to access a distinct multi-dimensional attribute from the secondary storage more than once and ultimately, the primary storage space and expected number of secondary storage access can be theoretically bounded. Furthermore, the new approach allows constant CPU time and number of secondary storage access index maintenance, which would not impose a heavy burden on the overall performance of our approach for dynamic attributed graphs. This new constraint verification approach (hash index) can be used not only for our problem, but it can also be used by any graph traversal approach that involves attribute constraint verification.

- We proved theorems that can ensure the correctness of our new approach and we analyze the expected number of secondary storage access. We conclude that the expected number of secondary storage access for point attribute constraint query(Definition 15) is $O(A)$ and for set attribute constraint query (Definition 16) is $O(A_{diff} \times A)$, where A is the size of a multi-dimensional attribute and A_{diff} is the number of different multi-dimensional attributes visited during graph traversal.
- We developed a heuristic search technique that takes into account graph structure as well as attribute distribution during graph traversal so as to reduce A_{diff}. The idea of the heuristic search is to offer a direct passage that goes across graph regions that are likely to satisfy attribute constraints from source to destination. Also, the correctness of the heuristic search is theoretically proven. We emphasize that prevalent graph traversal techniques (e.g. BFS, DFS, A^* with Landmark [22]) can all be used for our heuristic search techniques. Furthermore, the techniques that we proposed can be used for both directed and undirected graphs.
- We proposed two optimization techniques for further improving the efficiency of our new constraint verification approach. The first one is the batch attribute retrieval strategy which wisely retrieves verification material in batches. As a result, the batch attribute retrieval strategy can guarantee another worst case number of secondary storage access. The other one is a guidance for attribute selection strategy when extra primary storage is available for storing more hash values so that valuable primary storage space can be better utilized.

C2: How-to-Reach Query on Attributed Graph [23] The second contribution of this thesis is an approach for efficiently finding high quality sub-optimal for answering How-to-Reach query on attributed graphs.

- We introduced and defined a new How-to-Reach query which can be implemented in attributed graph database systems for improving databases' usability. Since there may exist many attribute constraints that would allow source and destination to be connected, we further proposed a metric that would reflect the importance of attribute values for computing answer quality.
- We proposed a simple trick that 1)does not require heavy modification of existing implementations in graph database systems and 2)is proven to be able to allow existing shortest path algorithms to return optimal answers for How-to-Reach queries.
- Although after applying our trick, Dijkstra's algorithm can return optimal answer for How-to-Reach queries, we observed that 1)the computation time of Dijkstra's algorithm is unacceptable for inpatient users (i.e. $\approx$50sec) and 2)the hop distance of the optimal $s-t$ path tends to be meaningless (i.e. $\approx$1500 hops). Hence, to use such query in big graphs, we proposed the station index, which is a time and space efficient non-traversal based index that returns high-quality approximate answers with reasonable hop distances.

C3: Visualizable Path Summary on Attributed Graph [24] The third contribution of this thesis is an approach for effectively computing visualizable path summary on attributed graphs.

- We introduced and defined the visualizable path summary query on attributed graph problem. We defined attributed path summary to be groups of vertices that contain users' intuition as well as satisfy some path properties. The users' intuition is expressed as hints for computing the path summary. Users can offer whatever attribute values that they consider as the hint. These summaries offer insight to users about the attribute values and connection between the given source and destination vertices.
- We proposed an efficient and effective approach for finding attributed path summary. Our proposed approach consist of three phrases. The first phrase efficiently finds all key vertices that have attribute values belonging to the hint offered by the user. Then, based

on those key vertices, a novel stitching algorithm is proposed to connect the source, the destination, and key vertices together to form a relatively small key vertex graph. After that, high-quality candidate paths between the source and the destination are found on that small key vertex graph efficiently. Finally, candidate paths are inflated to vertex groups by greedily including adjacent vertices.

1.3 RESEARCH STATEMENT

By defining basic attributed graph queries and developing techniques for efficiently and effectively answering basic attributed graph queries, meaningful applications that are related to topological and attribute relationship between vertices on attributed graph can be developed and answered effectively and efficiently.

1.4 THESIS ORGANIZATION

The rest of this thesis is organized as follow: Chapter 2 introduces background and definitions for this thesis. Chapter 3 offers a literature review of all prior works that are related to this thesis. Chapter 4 presents all the details of our approach for efficiently answering attributed graph reachability queries. Chapter 5 presents all the details of our approach for efficiently and effectively answering attributed graph How-to-Reach queries. Chapter 6 presents all the details of our approach for effectively computing visualizable path summary for attributed graphs. Finally, Chapter 7 concludes this thesis.

2.0 BACKGROUND AND DEFINITIONS

In this chapter, we first define necessary notations and definitions used throughout the remainder of this thesis. After that, we introduce the primary-secondary hybrid storage framework utilized in this thesis.

2.1 DEFINITION OF ATTRIBUTED GRAPH AND ATTRIBUTE CONSTRAINT

Definition 1. [**Attributed Graph**] *An attributed graph [1] G, is an undirected graph denoted as $G = (V, E, A_v, A_e)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $A_v = (A_{v1}, A_{v2}, ..., A_{vd_v})$ is a set of d_v vertex-specific attributes, i.e. $\forall v \in V$, there is a multidimensional tuple $A_v(v)$ denoted as $A_v(v) = (A_{v1}(v), A_{v2}(v), ..., A_{vd_v}(v))$, and $A_e = (A_{e1}, A_{e2}, ..., A_{ed_e})$ is a set of d_e edge-specific attributes, i.e. $\forall e \in E$, there is a multidimensional tuple $A_e(e)$ denoted as $A_e(e) = (A_{e1}(e), A_{e2}(e), ..., A_{ed_e}(e))$.*

Figure 2.1 is an example of attributed graph. It has a topology (V, E) and two attribute tables to store vertex attributes $(A_v, d_v = 3)$ and edge attributes $(A_e, d_e = 2)$. A super-graph, which will be introduced in Chapter 4.5.1, is also shown in Figure 2.1.

Definition 2. [**Vertex Attribute Value**] *$A_{vi}(v)$ is the value of the i^{th} attribute of vertex v and is in the domain D_{vi} of attribute A_{vi}.*

$$A_{vi}(v) \in D_{vi} \ \ where \ \ D_{vi} \ \ is \ \ the \ \ domain \ \ of \ \ A_{vi}$$

Definition 3. [**Edge Attribute Value**] *$A_{ei}(e)$ is the value of the i^{th} attribute of edge e and is in the domain D_{ei} of attribute A_{ei}.*

$$A_{ei}(e) \in D_{ei} \ \ where \ \ D_{ei} \ \ is \ \ the \ \ domain \ \ of \ \ A_{ei}$$

For example, $A_{v\_country}(v2)$=C_A,$A_{e\_type}((s,v2))$=$friend$.

Definition 4. [**Vertex Attribute Value Set**] *S_{vi} is a set of attribute value of A_{vi}.*

$$S_{vi} \subseteq D_{vi}$$

Definition 5. [**Edge Attribute Value Set**] *S_{ei} is a set of attribute value of A_{ei}.*

$$S_{ei} \subseteq D_{ei}$$

For example, in Figure 2.1, $D_{v\_country} = \{C_A, C_B, C_C, C_D\}$ and $S_{v\_country}$ can be any possible subset of $D_{v\_country}$ e.g. $S_{v\_country} = \{C_A, C_B\}$.

Definition 6. [**Vertex Attribute Constraint**] *C_v is a set of vertex attribute value set S_{vi}.*

$$C_v = \{S_{v1}, S_{v2}, ..., Sv_{d_v}\}$$

Definition 7. [**Edge Attribute Constraint**] *C_e is a set of edge attribute value set S_{ei}.*

$$C_e = \{S_{e1}, S_{e2}, ..., Se_{d_e}\}$$

For example, a vertex attribute constraint can be $C_v = \{\{C_A\}, \{Eng., IT, Fin.\}, \{R_1\}\}$.

Definition 8. [**Vertex Attribute Constraint Satisfy**] *Vertices in path p $A_v(p)$ satisfies vertex attribute constraint C_v if and only if for all vertex v in p exclude s and t, every attribute $A_{vi}(v)$ of v belongs to S_{vi}, where $S_{vi} \in C_v$.*

$$A_v(p) \ sat. \ C_v \ iff \ \forall v \in p \setminus \{s,t\}, \forall i = 1...d_v, A_{vi}(v) \in S_{vi},$$

$$where \ \ S_{vi} \in C_v$$

Definition 9. [**Edge Attribute Constraint Satisfy**] *Edges in path p $A_e(p)$ satisfies edge attribute constraint C_e if and only if for all edge e in path p, every attribute $A_{ei}(e)$ of e belongs to S_{ei}, where $S_{ei} \in C_e$.*

$$A_e(p) \ sat. \ C_e \ \ iff \ \forall e \in p, \forall i = 1...d_e, A_{ei}(e) \in S_{ei},$$

$$where \ \ S_{ei} \in C_e$$

For example, in Figure 2.1, vertex $s, v2, v3, v4, v5, v12, t$ satisfy the vertex attribute constraint $C_v = \{\{C_A\}, \{Eng., IT, Fin.\}, \{R_1\}\}$.

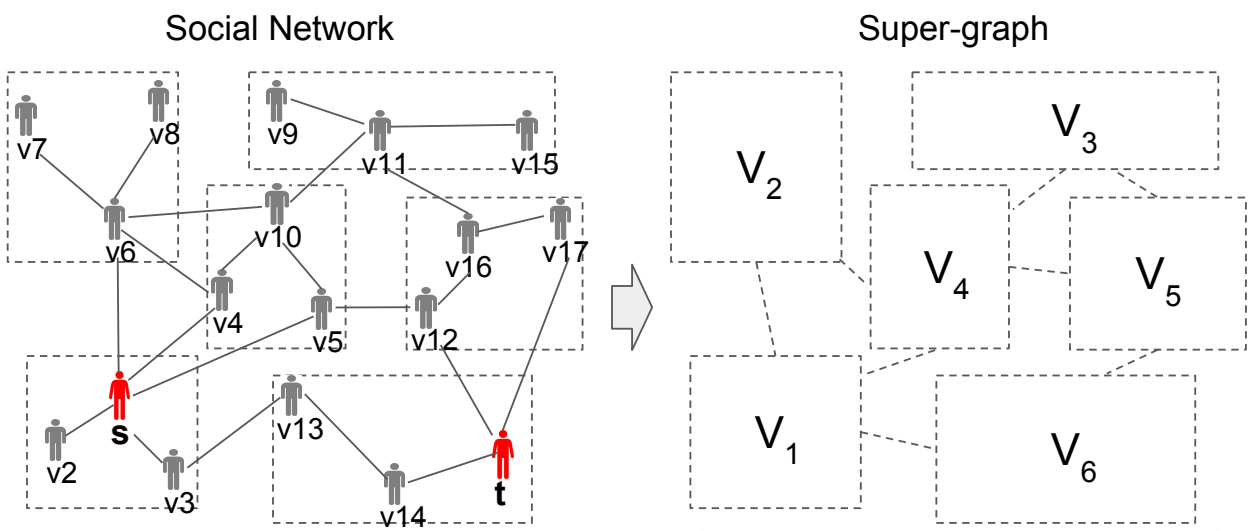
2.2 PRIMARY-SECONDARY HYBRID STORAGE FRAMEWORK

A pure in-memory solution does not scale well for graph query with attribute constraints. That is because attributed graphs consist of topology as well as attributes for every vertex and edge. For a large graph with 1 billion vertices, 3 billion edges, and 10 attributes for each vertex and edge, the size of all attributes is around 223 gigabytes (4 bytes per integer), which usually does not fit in primary storage (e.g. memory). Hence, it is unrealistic to assume that both topology and attributes of a big attributed graph can be always fit in primary storage.

In this thesis, a primary-secondary hybrid framework [2] is adopted for efficient attributed graph query processing. Figure 2.2 is the framework. In this framework, the entire graph topology is stored in the primary storage (e.g. memory or local storage) while vertex and edge attributes are stored in the secondary storage (e.g. disk or remote storage). This framework can also be viewed as a two-level storage hierarchy. The first level has faster read/write speed but smaller storage capacity; the second level has slower read/write speed but larger storage capacity.

2.3 QUERY PROCESSING SYSTEM FRAMEWORK

Figure 2.3 shows a potential query processing system framework that includes our contributions (red components) in this thesis. The system contains traversal based index and non-traversal based index for answering attributed graph queries. Both indexing approach consist of the query handler component and the index component. The query handler component controls how the query is answered and how the index is used; the index component offers services and information for the query handler component to compute the query answer. For traversal based indexing approach, there is the hash index component. This component offers indexing information for reducing number of secondary sotrage access of attributed graph traversal.



vID	Country	Job	Religion	hash
s	C_A	Eng.	R_1	39
$v2$	C_A	IT	R_1	48
$v3$	C_A	IT	R_1	48
$v4$	C_A	IT	R_1	48
$v5$	C_A	IT	R_1	48
$v6$	C_D	Fin.	R_3	120
...	...	...	...	...
$v10$	C_B	Fin	R_1	60
$v12$	C_A	Fin.	R_1	57
$v13$	C_B	IT	R_2	78
$v14$	C_C	Eng.	R_2	72
t	C_A	IT	R_1	48

Edge	Type	Since	hash
$(s, v2)$	friend	2010	30
$(s, v3)$	relative	2010	33
$(s, v4)$	relative	2011	42
$(s, v5)$	relative	2009	24
$(s, v6)$	friend	2016	84
$(v3, v13)$	relative	2015	78
$(v4, v6)$	relative	2015	78
$(v5, v10)$	relative	2015	78
$(v4, v10)$	relative	2015	78
...	...	...	...
$(v12, v16)$	friend	2016	84
$(v13, v14)$	friend	2010	30
$(v5, v12)$	friend	2010	30
$(v14, t)$	relative	2009	24
$(v12, t)$	relative	2010	33
$(v17, t)$	friend	2016	84

Figure 2.1: Attributed Graph and Super-graph

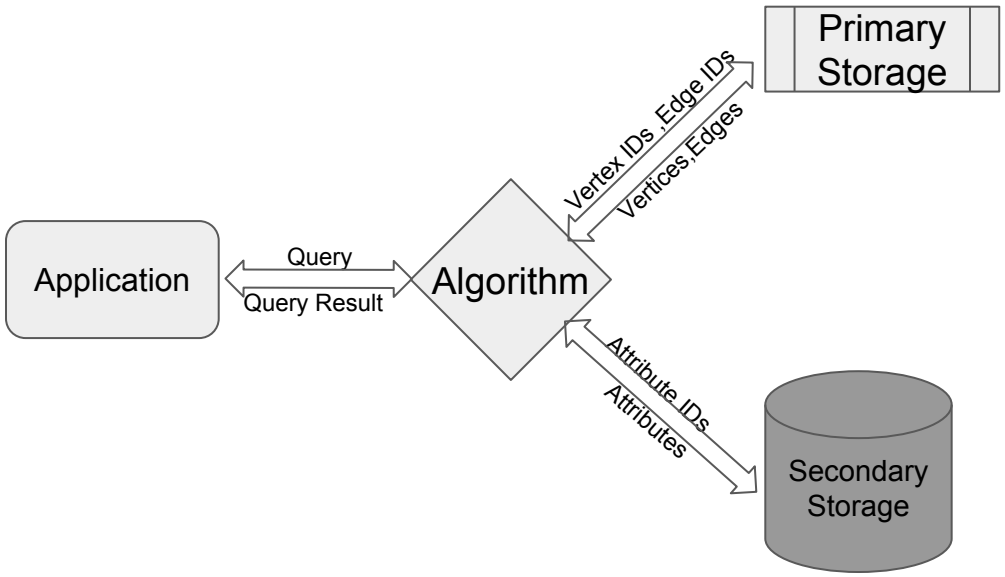


Figure 2.2: Primary-secondary Hybrid Storage Framework

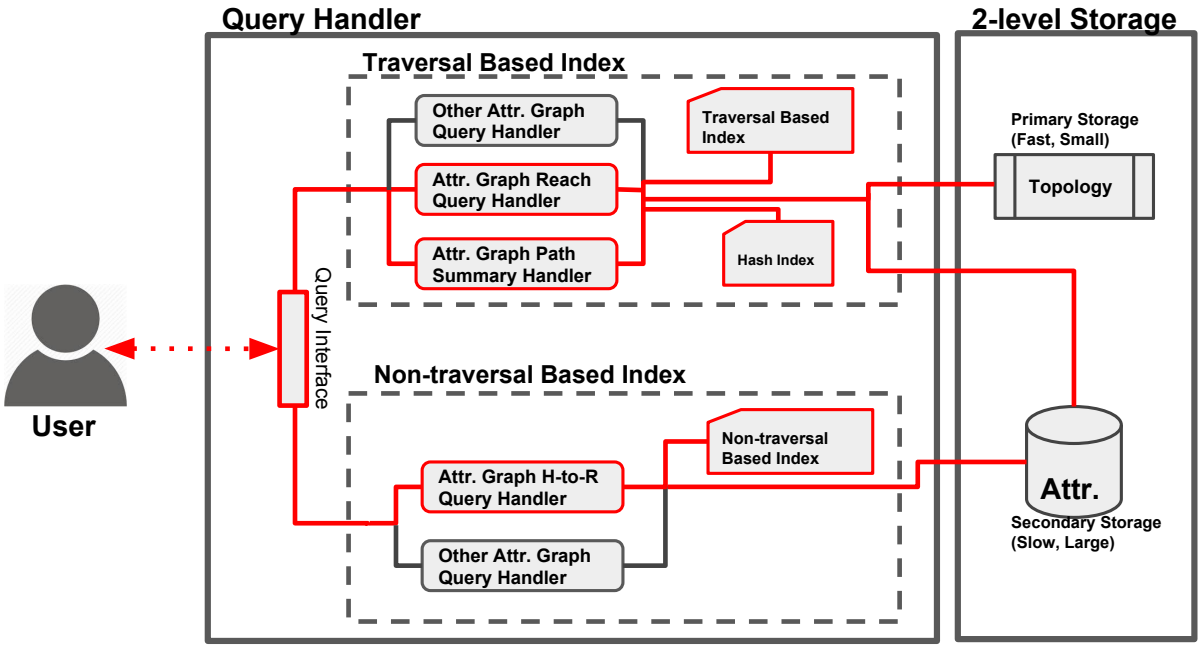


Figure 2.3: Potential Query Processing System Framework

3.0 LITERATURE REVIEW

In this Chapter, we talk about existing literature and techniques that are related to this thesis.

3.1 ATTRIBUTED GRAPH RESEARCH

3.1.1 Attributed Graph Query Engine

Sakr et al. [2] proposed a query engine that can support a combination of different types of queries over large attributed graphs. Their contributions include proposing an SPARQL-like language, called $G - SPARQL$, for querying attributed graphs, presenting an efficient hybrid memory/disk representation of large attributed graphs, and describing an execution mechanism for their proposed query language so as to optimize the query performance.

3.1.2 Attributed Graph OLAP

Wang et al. [1] proposed a new graph OLAP system, *Pagrol*, to provide efficient decision making query support for large attributed graphs. Their contributions include proposing a new conceptual graph cube model, *Hyper Graph Cube*, to support decision making queries and proposing various Map-reduce optimization techniques to efficiently compute graph cube.

3.1.3 Attributed Graph Pattern Matching

Tong et al. [16] proposed a framework and a method for efficiently finding best-effort subgraphs in an attributed graph with single attribute on each vertex. Roy et al. [15] indexing technique and query processing algorithm for efficiently processing graph pattern matching queries on weighted attribute graphs.

3.1.4 Attributed Graph Summarization

Graph summarization has been extensively studied [5, 6, 7, 8, 9, 10, 11, 12, 13, 14], and various ways of summarizing graphs have been proposed. Grouping-based summarization methods [5, 6, 6, 7, 8] takes into account both graph structure and attribute distributions for aggregating vertices into supernode and superedges; compression-based summarization methods [9, 10, 11] exploit the MDL principle to guide the grouping of vertices or the discovery of frequent sub-graphs to form a graph summary; influence-based summarization methods [12] leverage both graph structure and vertex attribute value similarities in the problem formualtion so as to summarize the influence process in a network; pattern-mining-based summarization methods [13, 14] identify frequent graph structural patterns for aggreagate into supernodes so as to reduce the size of the input graph and as a result, improving query efficiency. These techniques focus on computing summary for the whole graph. On the other hand, our techniques focus on computing visualizable path summary between two vertices that users are interested in.

3.1.5 Attributed Graph Clustering

Zhou et al [25] proposed $SACluster$, which is an attributed graph clustering algorithm based on both graph structural and attribute similarities through a unified distance measure. Zhou et al [25] proposed first to partition a large graph associated with attributes into k clusters so that each cluster contains a densely connected subgraph with homogeneous attribute values. Then, an effective method is used to automatically learn the degree of contributions of structural similarity and attribute similarity. Zhou et al [26] further improve the effi-

ciency and scalability of *SACluster* [25] by proposing an efficient algorithm *IncCluster* to incrementally update the random walk distances given the edge weight increments.

One fundamental difference between summarization and clustering is that former finds coherent sets of vertices with similar connectivity patterns to the rest of the graphs, while clustering aims at discovering coherent densely-connected groups of vertices [27]. Similar to graph summary, graph clustering only computes a summary of the whole graph while our techniques focus on a summary of paths between two vertices.

3.2 GRAPH QUERY PROCESSING RESEARCH

3.2.1 Reachability Query

Generally, an index is constructed offline so as to speed up online reachability query processing. An index is usually a set of labels, which contains reachability information for each vertex. Reachability query can be answered either by Label-Only approach or Label+Graph approach [28]. Label-only approach [29, 30, 31, 32, 33] answers reachability query by using the label of the source and the destination vertex. On the other hand, Label+Graph approach [34, 35, 36, 37, 38] access the graph when solely using label of source and destination vertex is not sufficient to answer the reachability query.

The reachability query has been extensively studied in graph, yielding a large number of algorithms [29, 30, 31, 32, 33, 34, 35, 36, 37, 38], with different query time, index size, and index construction time. The two extreme approaches are computing full transitive closure (TC) with $O(1)$ query time and $O(|V|^2)$ index space and using DFS/BFS with $O(|V|+|E|)$ query time and $O(1)$ index size. For all existing reachability query indexes, the query time and index size are in between the query time and index size of the two extreme approaches while the time complexity for index construction depends on the complexity of the index. A survey that summarizes reachability query indexing can be found in [39]. A table that summarizes the three main costs (query time, index size, and index construction time) for the existing works can be found in Table 1 in [28].

The reachability query with attribute constraints cannot be answered using traditional reachability query index. Existing works on graph reachability typically construct indexes which can be used to answer the Yes/No question of reachability, but they do not maintain any path information or how the source and destination vertex is connected. Although auxiliary path information can be added, it is unclear what and how information should be added and maintained since attribute constraints are given during query time.

3.2.2 Reachability Query with Constraint

Zou et al. [3] and Jin et al. [4] studied the edge label constraint reachability (LCR) query. In their problem settings, every edge in the directed acyclic graph is associated with a single label, and the two vertices are reachable if there exists a path that satisfies the given edge label constraint between the two vertices. Different from traditional reachability queries, LCR query needs to consider the edge labels along the path. In [3], Zou et al. transform an edge-label directed graph into an augmented direct acyclic graph and propose to use a partition-based framework which computes local path-label transitive closure for each partition. In [4], Jin et al. use a spanning tree and some local transitive closures to support LCR queries.

LCR query is equivalent to reachability query with attribute constraints when there is only one attribute on every edge which implies that LCR query is a special case of reachability query with attribute constraint. Hence, solution for LCR query cannot be directly used for efficiently processing reachability query with attribute constraint. Although it is possible to map a multi-dimensional attribute to a label, a very larger domain is needed for such mapping. Even though we assume that a multi-dimensional attribute can be mapped to a label, approaches in [3, 4] still cannot be directly applied for our problem setting as: 1) in the problem setting of [6,7], there is only one attribute on edge (no vertex attribute), 2) their approaches assume everything is in memory, and 3) the time and space complexity are too high for disk-based index construction.

In [40], Fan et al. defined a reachability query with regular expressions constraint on a graph with vertex attribute and single edge label. The definition is different from reachability

with attribute constraint since 1) the definition only constraints attribute of the source and destination vertex, 2) there is only a label for every edge and 3) the edge label constraint is not an or-condition (e.g. $fa^{\leq 2}fn$ in [40] Fig.1).

3.2.3 Shortest Path Query

Table 3.1: Summary of Shortest Path Index Issue

Approach	**Issue**
Label-based [41, 42]	cannot precompute 2-hop cover
Tree Decomposition-based [43, 44]	cannot precompute shortest paths
Search-based [45, 46]	huge space consumption on attr. graphs

3.2.3.1 Labeling-based Exact Methods Distance-aware 2-hop cover indexing approaches [41, 42] rely on finding a set of vertices $C(u)$ for each vertex u such that for every pair of vertices, there exists at least 1 vertex $w \in C(u) \cap C(v)$ on a shortest path between u and v. However, for our problem, we do not know such w in advance as the penalty/cost is related to a query parameter - attribute constraint.

3.2.3.2 Tree Decomposition-based Exact Methods Tree decomposition-based methods [43, 44] require to pre-compute all pair shortest path distance within every tree nodes. Since C_0 is given during query time, we cannot pre-compute such shortest path distances. Pre-computing all possible paths within a tree node is possible, but it is very impractical.

3.2.3.3 Search-based Exact/Approximate Methods Landmark [45, 46] is a popular technique that has been widely adopted for computing lower and upper bound distance between vertices so as to perform pruning and speed up graph traversal. In landmark-based methods, a set of vertices is chosen as landmarks, and single-source shortest paths are computed from each landmark. Landmark technique can also be used as an approximate method, which returns the distance estimations between sources and destinations. Without knowing attribute constraints, multiple paths has to be pre-computed. However, computing multiple

paths distances between every landmark to all vertices is not a space efficient approach when attribute information has to be computed and stored. In evaluation section, we compared the approximate landmark-based method with our approach.

3.3 WHY-NOT QUERY PROCESSING RESEARCH

'Why-Not' has been widely studied since it was first proposed by Chapman et al [47] in 2009. Existing 'Why-Noy' approaches can be rougly classified into three categories: (a) manipulation identification [48, 47], (b) database modification [49, 50, 51, 52], and (c) query refinement [53, 54, 55, 56]. To the best of our knowledge, we are the first to address why-not questions on attribute constrained reachability queries.

3.4 GRAPH DATA MANAGEMENT SYSTEM RESEARCH

3.4.1 Semi-structured Data Managment System

Semi-structured data management systems, such as [2, 57], offer query languages such as SPARQL to query RDF data, and XML query languages to query XML documents. SPARQL-based systems mainly target on graph pattern matching query [58]. Furthermore, converting an attributed graph to RDF makes the number of triples to be a few times larger than the number of vertices and/or edge [59]. XML querying techniques [57] is mainly used for managing tree-structured data instead of graphs.

3.4.2 Graph Management System

Centralized graph platforms, such as Grace [60] and GraphChi [61], and distributed graph platforms, such as Pregel [62], Giraph [63], Trinity [64] and PowerGraph [65], mainly focus on query optimization and system design which is different from the focus of this thesis - optimization of algorithm complexity and efficiency.

3.5 ADVANCED GRAPH TRAVERSAL TECHNIQUE

3.5.1 BFS/DFS

BFS and DFS are two well-known graph search techniques. Both of them start from a source vertex and visit other vertices during graph traversal. However, BFS and DFS do not consider any attribute information and the location of destination vertex. On the contrary, our proposed heuristic search is a guided search that takes into account both factors.

3.5.2 A* Search and Landmark Technique

A* algorithm [66] is a well know best-first heuristic search algorithm that has been widely used for different applications and several extensions have been proposed to improve its performance [67]. A* algorithm involves an estimation of the cost of the cheapest path from the current vertex to the destination during graph traversal. For an attributed graph, it is non-trivial to accurately estimate such cost since there are a lot of possible paths from the current vertex to the destination. Although the offline construction of landmark in [22] can be used for estimating such cost, landmark technique does not consider online attribute constraints. Online attribute constraints can make any path from the current vertex to the destination vertex change or disappear. As a result, estimated cost from current vertex to destination vertex is highly affected by online attribute constraints.

3.6 GRAPH VISUALIZATION TECHNIQUE

The size of the graph to view is a key issue in graph visualization [17]. To deal with this, researchers proposed a lot of techniques in graph drawing [17], such as H-tree layout, radial view, balloon view, tree-map, spanning tree, cone tree, hyperbolic view, as well as methods for reducing visual complexity [18], such as clustering, sampling, filtering, partitioning. We argue that simply applying those graph drawing technique cannot handle big attributed graphs with million of vertices and edges as these methods are too general. For existing

visual complexity reduction methods, how to effectively applying them to our problem needs further investigation.

4.0 FAST REACHABILITY COMPUTATION

In this Chapter, we study the problem of efficient processing of reachability query on attributed graphs [19].

4.1 MOTIVATION APPLICATION

Attributed graph is widely used for modeling a variety of information networks [2, 1], such as the web, sensor networks, biological networks, economic graphs, and social networks. Given the high popularity of attributed graph, the efficient processing of attributed graph query becomes an important issue for different attributed graph applications. Unfortunately, the study of efficient query processing on attributed graphs did not catch much attention. To the best of our knowledge, there is only two literature studying attributed graph pattern matching query [15, 16]. To contribute to the study of efficient processing of attributed graph query, in this chapter, we study one of the most fundamental graph query type - the reachability query with attribute constraint.

In many real applications, both topological structures in addition to attributes of vertices and edges are important [2].

Social Network: In a social network, each person is represented as a vertex, and two persons are linked by an edge if they are related. Vertex attributes can be the profile of a person; Edge attributes can be details of relationships between two persons. A reachability query on social networks discovers whether person A relates to person B under given path attribute constraints. For example, for investigation purpose, a police officer can ask whether there is a path from person A to a leader in a terrorist group such that all persons and

relationships on the path satisfy the vertex attribute constraint:{country=country A and religious view=religion X} and edge attribute constraint:{year=2010}.

Economic Graph: In an economic graph [68](e.g. LinkedIn), each user or company can be represented by a vertex and two related users or companies are connected by an edge. Vertex attributes can be job profile of a person; edge attributes can be details of relationships between two users or companies. A recruitment agent can ask whether applicant A relates to a board member in the company such that all users on the path working in company A ({employer=company A}) and are connected since 2012 ({year=2012}). During commercial crime investigation, a detective can ask whether user A relates to company B such that all users on the path worked in company B ({previous employer=company B}).

Metabolic Network: In metabolic networks, each vertex is a compound, and an edge between two compounds indicates that one compound can be transformed into another one through a certain chemical reaction. Vertex attributes can be profile of the compound; edge attributes can be details of a chemical reaction between two compounds. A reachability query on metabolic networks discovers whether compound A can be transformed to compound B under given path attribute constraints. For example, a scientist can ask whether compound A can be transformed to compound B such that all compounds and chemical reactions on the path satisfy the vertex attribute constraint:{state=solid or liquid} and edge attribute constraint:{cost-to-trigger-reaction$\leq$100}.

Other than the above applications, reachability query with attribute constraint can also be applied to other attributed graphs such as chemical reaction networks, gene regulatory networks, protein-protein interaction networks, signal transition networks, communication networks, attributed road networks, etc.

4.2 CHALLENGES AND TECHNICAL CONTRIBUTIONS

4.2.1 Chellenges

The inclusion of attribute constraints in reachability query makes the index design more complicated than ordinary reachability query. That is mainly because of the following reasons:

Indexing an attributed graph involves not only indexing the graph topology but also attributes of vertices and edges. Hence, we need an index that takes into account both graph topology and attributes.

Unlike ordinary reachability query, we do not know the graph structure that satisfies attribute constraints in advance as attribute constraints are given at query time. It is possible that any subgraph can satisfy the given attribute constraints. Furthermore, Jin et al. [4] mentioned that since traditional reachability query index does not include attribute information, it cannot be easily extended to answer reachability query with edge label constraint, which is a special case of our problem.

A pure in-memory solution does not scale well for reachability query with attribute constraints on large attributed graphs. That is because, for a large graph (e.g. Facebook social network, Linkedin economic graph, Twitter social network) with 1 billion vertices, 3 billion edges, and five attributes for each vertex and edge, the size of all attributes is around 150 gigabytes (assume using eight characters to represent an attribute), which usually does not fit in memory.

4.2.2 Technical Contributions

Our first contribution is to introduce and define the reachability query on attributed graph problem. We observe that it is unrealistic to assume that both graph topology and attributes can be fit in memory for large attributed graphs. Therefore, we adopted the approach in [2] which store graph topology in the memory and attributes in the disk. Based on this approach, we propose a memory-disk hybrid approach for efficiently answering reachability query on attributed graphs. The techniques that we proposed can be used for both directed and

undirected graphs. In this chapter, we will present these techniques using undirected graphs as an example.

Our second contribution is to propose a hashing scheme for reducing and bounding expected I/O. We proved a theorem that can ensure the correctness of our hashing scheme and we analyzed the expected number of I/O. We conclude that the expected I/O for point attribute constraint query(Definition 15) is $O(\frac{A}{B})$ and for set attribute constraint query (Definition 16) is $O(A_{diff} \times \frac{A}{B})$, where B is the size of a disk block, A is the size of a multi-dimensional attribute, and A_{diff} is the number of different multi-dimensional attributes visited during graph traversal. Besides, this hashing scheme is not limited to reachability query. It can be adopted by any query that requires attributed graph traversal.

Since the number of I/O is related to A_{diff}, our third contribution is to propose a heuristic search technique so as to reduce A_{diff}. The idea of the heuristic search is to traverse regions that are likely to pass through and close to the destination.

Finally, we conduct extensive experiments on both real and synthetic datasets. We found that the hashing scheme and heuristic search technique effectively reduce the number of I/O as well as in-memory computation time.

4.3 PROBLEM DEFINITION

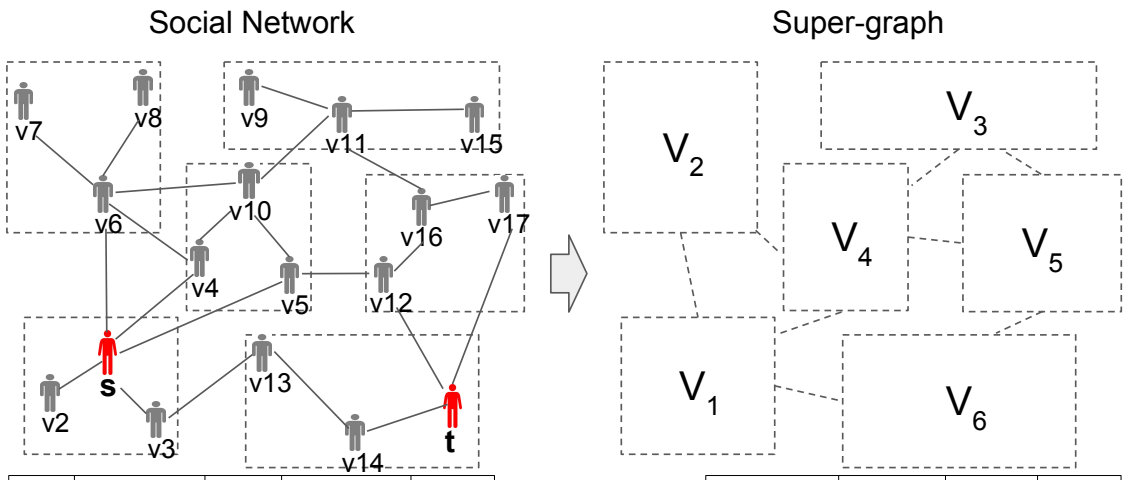
In this section, we will present the preliminary definitions and the problem statement. Table 4.1 shows frequently used symbols in this chapter.

Definition 10. [**Attributed Graph**] *An attributed graph [1] G, is an undirected graph denoted as $G = (V, E, A_v, A_e)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $A_v = \{A(v)\}$ is a set of d_v vertex-specific attributes, i.e. $\forall v \in V$, there is a multidimensional tuple $A(v)$ denoted as $A(v) = (A_1(v), A_2(v), ..., A_{d_v}(v))$, and $A_e = \{A(e)\}$ is a set of d_e edge-specific attributes, i.e. $\forall e \in E$, there is a multidimensional tuple $A(e)$ denoted as $A(e) = (A_1(e), A_2(e), ..., A_{d_e}(e))$.*

Figure 4.1 is an example of an attributed graph.

Table 4.1: Summary of Frequently Used Symbols

Symbol	**Meaning**
$G = (V, E, A_v, A_e)$	an Attributed Graph
V	a Set of Vertex
E	a Set of Edge
A_v	a Set of $A(v)$
A_e	a Set of $A(e)$
d_v	Dimension of Vertex Attribute
d_e	Dimension of Edge Attribute
v	a Vertex
$e_{(u,v)}$	an Edge with vertex u, v
A	Size of $A(v)$
B	Sec. Storage Device Block Size
$A(v)$	All Attributes of v
$A(e)$	All Attributes of e
$A_i(v)$	i^{th} Attribute of v
$A_i(e)$	i^{th} Attribute of e
D_i	Domain of $A_i(v)$
S_i	a subset of D_i
C_v	Vertex Attribute Constraint
G_h	G with Attribute Hash Value



vID	Country	Job	Religion	hash
s	C_A	Eng.	R_1	39
$v2$	C_A	IT	R_1	48
$v3$	C_A	IT	R_1	48
$v4$	C_A	IT	R_1	48
$v5$	C_A	IT	R_1	48
$v6$	C_D	Fin.	R_3	120
...	...	...	...	...
$v12$	C_A	Fin.	R_1	57
$v13$	C_B	IT	R_2	78
$v14$	C_C	Eng.	R_2	72
t	C_A	IT	R_1	48

Edge	Type	Since	hash
$(s, v2)$	friend	2010	30
$(s, v3)$	relative	2010	33
$(s, v4)$	relative	2011	42
$(s, v5)$	relative	2009	24
$(s, v6)$	friend	2016	84
$(v3, v13)$	relative	2015	78
...	...	...	...
$(v13, v14)$	friend	2010	30
$(v5, v12)$	friend	2010	30
$(v14, t)$	relative	2009	24
$(v12, t)$	relative	2010	33

Figure 4.1: Attributed Graph and Super-graph

In this chapter, we assume that both vertex and edge attribute domains are discrete. Hereinafter, we only use vertex attributes for illustration purpose.

Definition 11. [Vertex Attribute Value] *$A_i(v)$ is the value of the i^{th} attribute of vertex v and is in the domain D_i of attribute i.*

$$A_i(v) \in D_i, \ \ where \ \ D_i \ \ is \ \ the \ \ domain \ \ of \ \ attribute \ \ i$$

Definition 12. [**Vertex Attribute Constraint**] *C_v is a set of vertex attribute value set S_i.*

$$C_v = \{S_1, S_2, ..., S_{d_v}\}, where \;\; S_i \subseteq \;\; D_i$$

Definition 13. [**Vertex Attribute Constraint Satisfy**] *A path p satisfies vertex attribute constraint C_v if and only if for all vertices v in p excluding s and t, every attribute $A_i(v)$ of v belongs to S_i, where $S_i \in C_v$.*

$$p \; sat. \; C_v \;\; iff \; \forall v \in p \setminus \{s, t\}, \forall i = 1...d_v, A_i(v) \in S_i,$$

$$where \;\; S_i \in C_v$$

Definition 14. [**Reachability on Attributed Graph**] *Given an attributed graph G, a source vertex s, a destination vertex t, vertex constraint C_v, we say that s can reach t ($s \rightsquigarrow t$) under vertex constraint C_v if and only if there exists a path p from s to t where p satisfies C_v.*

$$s \rightsquigarrow t \;\; iff \;\; \exists p \;\; s.t. \;\; p \; sat. \; C_v$$

Problem Statement [**Attributed Grrph Reachability Query**] Given an attributed graph G, a source vertex s, a destination vertex t, vertex constraint C_v, and edge constraint C_e, reachability query on attributed graph verifies whether s can reach t under vertex and edge constraint C_v, C_e.

4.4 A NEW APPROACH FOR CONSTRAINT VERIFICATION

Obviously, DFS/BFS can be used to traverse the attributed graph and retrieve the corresponding attribute when encountering a vertex. By doing that the worst case secondary storage access bound is $O(|V| + |E|)$. In this section, we will introduce a new approach that exploits hash values to bound the expected number of secondary storage access for point and set attribute constraint query processing using merely graph traversal.

Definition 15. [**Point Attribute Constraint**] *C_v^p is a vertex attribute constraint such that the size of all S_i is 1.*

$$C_v^p = \{S_1, S_2, ..., S_{d_v}\}$$

$$where \;\; \forall i = 1...d_v, \forall j = 1...d_e, s.t. \; |S_i| = 1$$

Definition 16. [Set Attribute Constraint] *C_v^r is a vertex attribute constraint such that there exists S_i with size bigger than 1.*

$$C_v^r = \{S_1, S_2, ..., S_{d_v}\}$$

$$where \ \exists i \in 1...d_v, s.t. \ |S_i| > 1$$

The basic idea of this new approach is to compress every multi-dimensional attribute to a hash value and store the hash values in primary storage for answering queries. During query time, attribute constraints are verified against hash values. If certain conditions are satisfied, we can verify the satisfiability of attributes by just comparing their hash values (without retrieving attributes from disk). We start the presentation with index construction.

4.4.1 Index Construction

Suppose the hash value of every attribute is computed offline and stored in primary storage. During query time, if we have to check the satisfiability of an attribute, we compare its hash value with the hash value of the point constraint. However, even though the two hash values are the same, we cannot conclude that the attribute satisfies the point attribute constraint as hash collision may happen. To make the hash value comparison to be valid, we have to ensure certain conditions are satisfied. Theorem 1 states those conditions.

THEOREM 1. **[Hash Value Condition]** *Attributes $A(v)$ of vertex v satisfies a point vertex constraint C_v^p if all three conditions below are true.*

$$A(v) \ sat. \ C_v^p \ \ if \ \ con.1 \wedge con.2 \wedge con.3 = true$$

1. *Hash value $hash(A(v))$ equals to hash value $hash(C_v^p)$. (i.e. $hash(A(v)) = hash(C_v^p)$)*
2. *There exists an attribute in the attributed graph G the same as C_v^p. (i.e. $\exists A(v_i) \in G$ s.t. $A(v_i) = C_v^p$)*
3. *There does not exist vertices $v_i, v_j \in G$ with different attributes $A(v_i), A(v_j)$ have hash value $hash(A(v_i)) = hash(A(v_j))$. (i.e. $\nexists v_i, v_j \in G, s.t. \ hash(A(v_i)) = hash(A(v_j)) \wedge A(v_i) \neq A(v_j)$)*

Proof. We prove this theorem by contradiction. Assume if $con.1 \wedge con.2 \wedge con.3 = true$, $A(v)$ *not sat.* c_v. As $A(v)$ *not sat.* C_v^p, we know that $A(v) \neq C_v^p$. Given that $A(v) \neq C_v^p$, it is possible that $hash(A(v)) \neq hash(C_v^p)$ or $hash(A(v)) = hash(C_v^p)$ (If $A(v) = C_v^p$, it is certain that $hash(A(v)) = hash(C_v^p)$ as hash function is deterministic.). If $hash(A(v)) \neq hash(C_v^p)$, it contradicts with condition 1 (♣). If $hash(A(v)) = hash(C_v^p)$, there exists 2 different vertex attributes can be mapped to the same hash value. Suppose $A(v') = C_v^p$, $hash(A(v')) = hash(A(v)) = hash(C_v^p)$, and $A(v') \neq A(v)$. $A(v')$ can be in G or not in G. If $A(v')$ is in G, (as $A(v)$ is in G,) both $A(v)$ and $A(v')$ are in G, $hash(A(v)) = hash(A(v')) = hash(C_v^p)$, and $A(v) \neq A(v')$ contradict with condition 3 (♠). If $A(v')$ is not in G, condition 2 (★) is violated. Because of (♣), (♠), and (★), the proof is complete. □

The index construction algorithm (Algorithm 1) is designed based on Theorem 1.

Algorithm 1 Index Construction

```
1: procedure ConHashIndex(G_h, iFile)
2:     for all v ∈ G_h do
3:         h ← GetAttrHash(v, G_h)
4:         a ← IOAttr(v, G_h)
5:         if hashAddr[h] = φ then
6:             addr ← IOInfo_w(iFile, a, 1)
7:             hashAddr[h] = addr
8:         else
9:             addr ← IOInfo_w(iFile,',' +a, 1) ▷ append to addr in iFile and will
   update count of that hash value
10:            hashAddr[h] = addr
11:        end if
12:    end for
13: end procedure
```

Figure 4.2 is an example of the index based on the attributed graph in Figure 4.1. The number of different attributes that have the same hash value is in the *Count* column. For example, the count of 48 is 1 as only C_A, IT, R_1 has hash value equal to 48.

v Attr. Hash	Attr.	Count
39	$C_A, Eng., R_1$	1
48	C_A, IT, R_1	1
57	$C_A, Fin., R_1$	1
72	$C_C, Eng., R_2$	1
78	C_B, IT, R_2	1
120	$C_D, Fin., R_3$	1

e Attr. Hash	Attr.	Count
24	$relative, 2009$	1
30	$friend, 2010$	1
33	$relative, 2010$	1
42	$relative, 2011$	1
78	$relative, 2015$	1
84	$friend, 2016$	1

Figure 4.2: Index Structure

4.4.2 Query Algorithm

Given the index structure (in Section 4.4.1), which is stored in $iFile$, any prevalent graph traversal algorithm (e.g. BFS, DFS, A^* with Landmark [22]) can adopt our new approach for verifying satisfiability of attributes before traversing to adjacent edges and vertices. More details for constraint verification (Algorithm 2) will be presented below.

4.4.2.1 Point Attribute Constraint Verification Algorithm 2 is the pseudo code. Algorithm 2 first gets the hash value h of a vertex v from G_h (line 2) and compute the hash value h_c of the point attribute constraint C_v (line 4). Then, it compares h with h_c (line 5). If $h \neq h_c$, $false$ is returned (line 6). After that, it looks up the satisfiability of v from $satTable$ (line 7), which is a global variable in primary storage. If sat is not empty, sat is returned (line 24). Otherwise, entry $(attr, count)$ of h is retrieved from $iFile$ (line 9). If $count = 1$, $attr$ is compared with C_v and $true/false$ is inserted into $satTable$ and returned (line 11-16); otherwise, attribute of v has to be retrieved from attribute file and compared with C_v (line 18-22).

4.4.2.2 Set Attribute Constraint Verification Algorithm 2 can also be used for set attribute constraint verification. The only difference is that line 4-6 is ignored, which result in secondary storage access (line 9) is needed for every hash value that is not in $satTable$.

Algorithm 2 Check Constraint

```
1: procedure CheckConstraint(v, C_v, G_h)
2:     h ← GetAttrHash(v, G_h)                          ▷ pri. storage operation
3:     if isPointConstraint(C) then
4:         h_c ← ConmputeHash(C_v)
5:         if h ≠ h_c then                               ▷ check condition 1
6:             return false
7:         end if
8:     end if
9:     sat ← satTable[h]                                 ▷ return φ if not exist
10:    if sat = φ then
11:        (attr, count) ← IOInfo_r(iFile, hashAddr[h])
12:        if count = 1 then
13:            if CheckAttr(attr, C_v) = true then
14:                satTable.insert(h, true)
15:                return true
16:            else if CheckAttr(attr, C_v) = false then
17:                satTable.insert(h, false)
18:                return false
19:            end if
20:        else
21:            attr ← IOAttr(v, G_h)                     ▷ get Attr. of v from disk
22:            if CheckAttr(attr, C_v) = true then
23:                return true
24:            else
25:                return false
26:            end if
27:        end if
28:    else
29:        return sat
30:    end if
31: end procedure
```

The effect of that to the number of secondary storage access is analyzed in below.

4.4.3 Bounding Expected Number of Secondary Storage Access

In this section, we will first analyze the expected number of secondary storage access for point attribute constraint query. Then, based on that result, we will devise the expected number of secondary storage access for set attribute constraint query.

4.4.3.1 Point Attribute Constraint Query

Lemma 1. [Probability of no Collision] *The probability of no collision for a hash value is* $(\frac{D-1}{D})^n$*, where* D *is the hash domain size and* n *is the number of data point.*

Proof. This is a well-known result for uniform hash function. □

For example, when a 64-bit integer is used as the hash domain D and there are 10 billion data points, $(\frac{D-1}{D})^n \approx 0.99999999994579$.

THEOREM 2. **[Secondary Storage Access Bound for Point Attr. Query]** *The expected number of secondary storage access for a point attribute constraint query when worst case happens is approximately* $O(A)$*, where* A *is size of* $A(v)$*.*

Proof. The verification of condition 1 is supported by Algorithm 2 line 5. Line 5 is an in-memory operation, so it does not cost any I/O.

The verification of condition 2 and 3 is supported by in Algorithm 2 line 9-24. Based on the pseudo code, if $sat \neq \phi$, no I/O is incurred; if $sat = \phi$, $1 + \frac{A \times count}{B}$ I/O is needed to retrieve an entry (1 I/O for *count* and $\frac{A \times count}{B}$ I/O for attribute(s)) from $iFile$ (line 9). After that, if $count = 1$, *sat* of h in $satTable$ is set to either $true$ or $false$ and it will never be ϕ again; if $count > 1$, $\frac{A}{B}$ I/O is used to get *attr* (line 18) and *sat* will always be ϕ until every vertex is visited in the worst case. Therefore, we can devise the expected number of I/O by:

$$E(IO) = Prob(count = 1) \times IO_{count=1} + Prob(count > 1) \times IO_{count>1}$$

$$IO_{count=1} = 1 + \frac{A \times count}{B} = 1 + \frac{A}{B}$$

$$IO_{count>1} = |V| \times (1 + \frac{A \times count}{B} + \frac{A}{B})$$

Based on Lemma 1, $Prob(count = 1) = (\frac{D-1}{D})^n \approx 1; Prob(count > 1) = 1 - (\frac{D-1}{D})^n \approx 0$, when 64-bit integer is used as D and there are 10 billion data points.

$$E(IO) \approx IO_{count=1} = 1 + \frac{A}{B}$$

We can conclude that the expected number of I/O when worst case happens is approximately $O(\frac{A}{B})$. □

4.4.3.2 Set Attribute Constraint Query

THEOREM 3. **[Secondary Storage Access Bound for Set Attr. Query]** *The expected number of secondary storage access for a set attribute constraint query when worst case happens is approximately $O(A_{diff} \times A)$, where A_{diff} is the number of different $A(v)$ visited, and A is size of $A(v)$.*

Proof. Since this is a set attribute constraint query, line 4-6 in Algorithm 2 is ignored. Therefore, during the BFS, for each A_v that has never appeared before (first-time attribute), $1 + \frac{A}{B}$ I/O is needed to retrieve an entry from $iFile$. For every first time attribute, based on the pseudo code, if $sat \neq \phi$, no I/O is incurred; if $sat = \phi$, $1 + \frac{A \times count}{B}$ I/O is needed to retrieve an entry (1 I/O for $count$ and $\frac{A \times count}{B}$ I/O for attribute(s)) from $iFile$ (line 8). After that if $count = 1$, sat of h in $satTable$ is set to either $true$ or $false$ and it will never be ϕ again; if $count > 1$, $\frac{A}{B}$ I/O is used to get $attr$ (line 18) and sat will always be ϕ until every vertex is visited in the worst case. Therefore, we can devise the expected number of I/O by:

$$E(IO) = A_{diff} \times \left[Prob(count=1) \times IO_{count=1} + Prob(count>1) \times IO_{count>1}\right]$$

$$IO_{count=1} = 1 + \frac{A \times count}{B} = 1 + \frac{A}{B}$$

$$IO_{count>1} = |V| \times (1 + \frac{A \times count}{B} + \frac{A}{B})$$

Based on Lemma 1, $Prob(count = 1) = (\frac{D-1}{D})^n \approx 1; Prob(count > 1) = 1 - (\frac{D-1}{D})^n \approx 0$, when 64-bit integer is used as D and there are 10 billion data points.

$$E(IO) \approx A_{diff} \times IO_{count=1} = A_{diff} \times \left[1 + \frac{A}{B}\right]$$

We can conclude that the expected number of I/O when worst case happen is approximately $O(A_{diff} \times \frac{A}{B})$. □

4.4.4 Discussion

4.4.4.1 Hash Function The proposed new approach requires a hash function with very few collisions so as to perform well. Murmur hash [20] and Spooky hash [21] are two non-cryptographic hash functions that can be used. We examined these two hash functions by using 1 billion 20-dimension attributes, and we discovered that both hash functions do not have any hash value collision.

4.4.4.2 Index Maintenance The update of index structure can be done using $O(1)$ CPU time and number of secondary storage access when there is an attribute update. Suppose attributes of v_2 (in Figure 4.1) is updated from $\{C_a, IT, R_1\}$ to $\{C_a, Eng., R_1\}$. First, the entry of hash value 48 is looked up in Table 4.2. Since there is more than one vertex with attribute $\{C_a, IT, R_1\}$, the entry of hash value 48 is not deleted from Table 4.2. The number of vertices with attribute $\{C_a, IT, R_1\}$ can be computed during index construction. Then, entry of hash value 39 is looked up in Table 4.2. Since the entry has attribute $\{C_a, Eng., R_1\}$, the count is not changed. The number of vertices with attribute $\{C_a, Eng., R_1\}$ is incremented by 1.

4.5 HEURISTIC SEARCH TECHNIQUE

In the above section, the expected number of secondary storage access for set attribute constraint query is $O(A_{diff} \times \frac{A}{B})$. In order to reduce the number of secondary storage access, we propose a heuristic search technique to reduce A_{diff}.

The intuition of this technique is to offer a direct passage that goes across graph regions that are likely to satisfy attribute constraints from s to t so that t can be reached faster. For example, in Figure 4.3, G is partitioned into 6 regions $(V_1, V_2, ..., V_6)$ and suppose s and t are in V_1 and V_6. If there are a lot of paths that satisfy attribute constraint from s to t, we do not want to:

- visit regions (e.g. V_2) that are far away from V_6 (case 1),

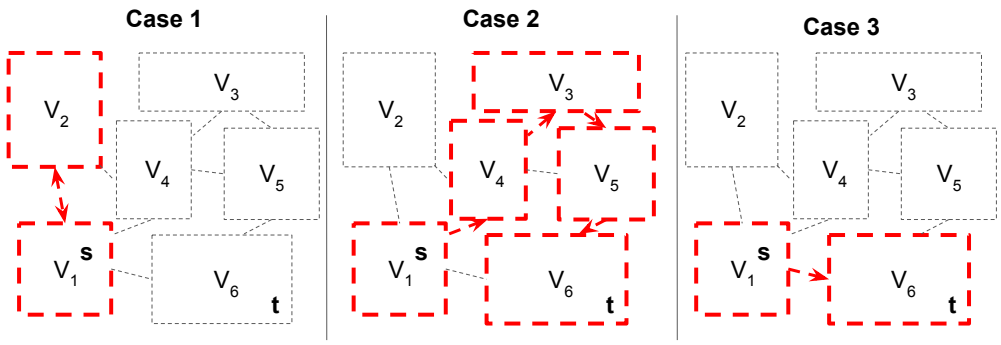


Figure 4.3: Choice of Path from s to t

- choose a long path to reach t such as a path that passes through $V_1 \rightarrow V_4 \rightarrow V_3 \rightarrow V_5 \rightarrow V_6$ (case 2);

instead, we want to find a short path from s to t (e.g. case 3 $V_1 \rightarrow V_6$). Our proposed heuristic search technique is designed based on this observation. More details are presented below.

4.5.1 Index Construction for Heuristic Search

We partition the graph into clusters and build a structure called - super-graph.

Definition 17. [**Super-graph**] *G_s is an undirected graph with super-vertex and super-edge, and for every super-vertex and super-edge, there is a synopsis that represents the distribution of attributes in the super-vertex or super-edge.*

Definition 18. [**Super-vertex**] *V_i is a vertex in G_s such that for every vertex v in G, v belongs to only one V_i in G_s.*

$$\forall v \in G, v \in V_i \wedge v \notin V_j \;\; if \;\; i \neq j, where \;\; V_i, V_j \in G_s$$

Definition 19. [**Super-edge**] *E_i is an edge in G_s such that if there exists an edge $e_{(u,v)}$ between vertex u, v in G and u, v belong to two different super-vertices V_i, V_j in G_s, $e_{(u,v)}$ belongs to $E_{(V_i,V_j)}$ in G_s.*

$$\forall e_{(u,v)} \in G, if \; u \in V_i \wedge v \in V_j, e_{(u,v)} \in E_{(V_i,V_j)}, where \; E_{(V_i,V_j)} \in G_s$$

For example, in Figure 1, the social network is partitioned into 6 clusters/super-vertices. Dotted rectangles are super-vertices. There is a super-edge between V_1 and V_2 as there is an edge from s to $v6$.

When building the super-graph, any clustering algorithm that fits dataset property can be used. For each vertex V_i and edge $E_{(V_i,V_j)}$ in G_s, we build synopsis, which represents the distribution of attributes in the super-vertex or super-edge. A simple synopsis can be a set of sample attributes drawn from vertices in the super-vertex.

4.5.2 Efficient Query Algorithm

Algorithm 3 is the pseudo code for the query algorithm. Given s, t, C_v, G_h, and G_s (line 1), the query algorithm answers the attribute constraint reachability query. A queue q is maintained in the algorithm (line 2). In the beginning, s is put into the queue q (line 3). When the graph traversal starts, the first element v is pop from q (line 5). Then, the algorithm finds a super-graph shortest path SP_s from super-vertex $SN[v.first]$ that contains v to super-vertex $SN[t]$ that contains t (line 9). A guided BFS_g is performed starting at vertex v (line 10). If a vertex v_{out} that is not in SP_s is visited, the vertex is put into q and adjacent vertices of v_{out} are not visited in this guided BFS_g. During graph traversal, synopsis in super-vertex and super-edge are updated. If t is not found in BFS_g, after BFS_g, another element is pop from q and the same steps as above are performed until q is empty (line 4). Figure 4.4 is an example.

4.5.2.1 Super-graph Shortest Path Given G_s and synopsis of every super-vertex and super-edge in G_s, the query algorithm tries to find a shortest path in G_s from super-vertex $SN[s]$ or $SN[h]$ that contains s or any h to super-vertex $SN[t]$ that contains t based on the sum of estimated pass-through probability SP_{cost}, which can be estimated by using the attribute constraints and the synopsis.

4.5.2.2 Guided Search Basically, any graph traversal algorithm can be adopted (e.g. A^*, BFS, DFS). For simplicity, we use BFS as an example. Guided BFS_g is a modified

Algorithm 3 Efficient Query Algorithm

1: **procedure** ReachabilityQuery(s, t, C_v, G_h, G_s)
2: $Queue\ q$
3: $q.put(pair(s, \phi))$ ▷ No need to check constraint of s
4: **while** $q.empty() = false$ **do**
5: $v = q.pop()$
6: **if** $visited[v.first] = true$ **then**
7: $continue$ ▷ Other BFS_g may set $visited$ to $true$
8: **end if**
9: $visited[v.first] \leftarrow true$
10: $SP_s \leftarrow SGraphSP(SN[v.first], v.second, SN[t], G.G_s, C_v)$
11: $(R, q) \leftarrow BFS_g(v.first, t, G_h, C_v, SP_s, q)$
12: **if** $R = true$ **then**
13: **return** yes
14: **end if**
15: **end while**
16: **return** no
17: **end procedure**

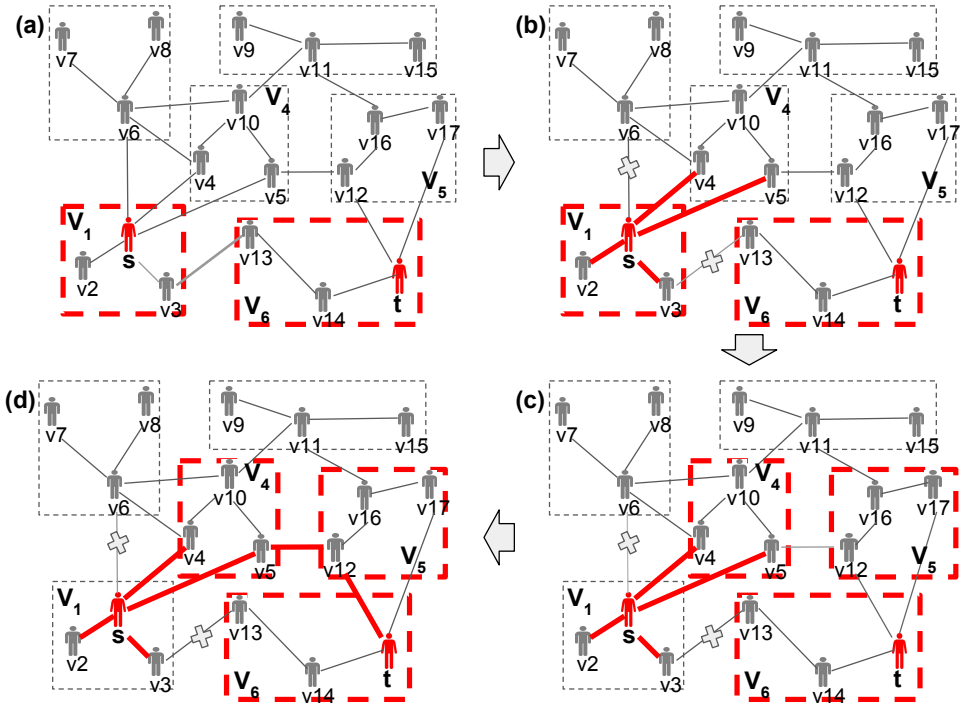


Figure 4.4: Example for Efficient Query Algorithm

version of typical BFS. Algorithm 4 is the pseudo code. The difference between guided BFS_g and typical BFS is that:

- vertex constraint is verified and synopsis is updated,
- vertices that are not in SP_s are not traversed (line 10) and are put into a queue q_G offered by the caller of the guided BFS (line 16), and
- $E^s_{(SN[cur],SN[v'])}$, which is a super-edge that can be ignored in $SGraphSP()$ when $SN[v']$ is used as source, is put into q_G together with v'.

Algorithm 4 Guided BFS

1: **procedure** $BFS_g(cur, t, G_h, C_v, SP_s, q_G)$
2: $Queue\ q$
3: $q.put(cur)$
4: **while** $q.empty() = false$ **do**
5: $v = q.pop()$
6: **if** $v = t$ **then**
7: **return** $(true, q_G)$
8: **end if**
9: **if** $visited[v] = true$ **then**
10: $continue$
11: **end if**
12: **if** $SN[v] \in SP_s$ **then**
13: $visited[v] \leftarrow true$
14: **for all** $v' \in v.adjList$ **do**
15: **if** $CheckConstraint_{BFS_g}(v', G_h, C_v) = true$ **then**
16: $q.put(v')$
17: **end if**
18: **end for**
19: **else** ▷ ignore $E_{(SN[cur],SN[v'])}$ in $SGraphSP()$
20: $q_G.put(pair(v, E_{(SN[cur],SN[v'])}))$
21: **end if**
22: **end while**
23: **return** $(false, q_G)$
24: **end procedure**

4.5.2.3 Synopsis Update Whenever a vertex is visited, we know the attribute of that vertex. Hence, the estimated pass-through probability of a super-vertex can be updated.

4.5.3 Proof of Correctness

We prove the correctness in this section.

Definition 20. *An attribute constraint reachability algorithm is correct if, in the worst case, all reachable vertices can be reached.*

Definition 21. [**Super-vertex Strongly Connected Component** $sSCC$] *is a strongly connected component such that:*

1. *all vertices in $sSCC$ are within the same super-vertex and*
2. *there exists a path between any vertex in $sSCC$ completely in the super-vertex.*

THEOREM 4. [**Correctness**] *Algorithm 3 can answer reachability query with attribute constraints correctly.*

Proof. Without loss of generality, we assume that all reachable vertices are condensed into $sSCC$s.

We prove the correctness by first defining the loop invariants and then prove the loop invariant is true for all iterations.

Loop Invariant: 1. At iteration i, DFS_g visits all $sSCC$ on SP_s and puts all unvisited $sSCC$ that are adjacent to $sSCC$ on SP_s into q. 2. All $sSCC$ in q will be visited before iteration $i + |q| + 1$, where $|q|$ is the current size of q.

Initialization: Prior to the first iteration, no $sSCC$ is visited.

Maintenance: Assume the invariant is true at iteration n. At iteration $n + 1$, the invariant is also true. That is because:

1. DFS_g visits all $sSCC$ on SP_s and puts all unvisited $sSCC$ that are adjacent to $sSCC$ on SP_s into q,
2. all $sSCC$ will be popped from q before iteration $i + |q| + 1$, and
3. whenever an $sSCC$ is popped from q, the $sSCC$ must be in SP_s and will be visited.

Termination: When the algorithm terminates, q is empty, and all reachable $sSCC$ are visited. That is because:

1. if q is not empty, iteration would continue and

2. if there are unvisited $sSCC$s that are adjacent to visited $sSCC$s, unvisited $sSCC$s would be put into q by DFS_g.

□

4.6 OPTIMIZATION: BATCH ATTRIBUTE RETRIEVAL TECHNIQUE

We observed that a disk block for a block-based device is usually big enough to store many attributes. For example, a 4096-byte disk block can store attributes of 102 10-attribute vertices (4 bytes per attribute). Therefore, storing attribute values of only a 10-attribute vertex in a 4096-byte disk block and retrieving it using an I/O is very inefficient.

Motivated by this, in this section, we will present a *batch attribute retrieval* technique to achieve such optimization. Then, we will analyze the worst case I/O complexity for BFS algorithm with this technique. Finally, a summary of time and space complexity is provided. In this section, we assume that a block-based device is used (i.e. we can only retrieve a fix size block from the device for every I/O.).

4.6.1 Index Construction

We propose to pack attributes and their hash value counts of adjacent vertices and edges into a disk block. For example, in Figure 4.1, vertex/edge ids, attributes, and hash value counts of s, v_2, v_3, v_4, v_5, v_6, $e_{(v_2,s)}$, $e_{(v_3,s)}$, $e_{(v_4,s)}$, $e_{(v_5,s)}$, $e_{(v_6,s)}$, are stored in a 4096-byte disk block (assume 4 bytes per attribute dimension and 10 attribute dimensions).

Algorithm 5 is the pseudo code for index construction. For every vertex v, the algorithm first checks whether its contents have been stored (line 4-5). If not, the contents of v is packed into block b (line 6) and $isStored[v]$ is set to $true$ (line 7). Then, for every adjacent vertex vi of v, if content of v' is not stored, it is put into b (line 9-10); if content of $e_{(v,v')}$ is not stored, it is put into b also (line 11-12) and $isStored[e_{(v,v')}]$ is set to $true$ (line 13). After that, b is written to disk (line 14) and $addr$ of it is stored in $AttrAddr[v]$ (line 15). For every adjacent vertex v' of v, if they were not stored before, disk address of contents of

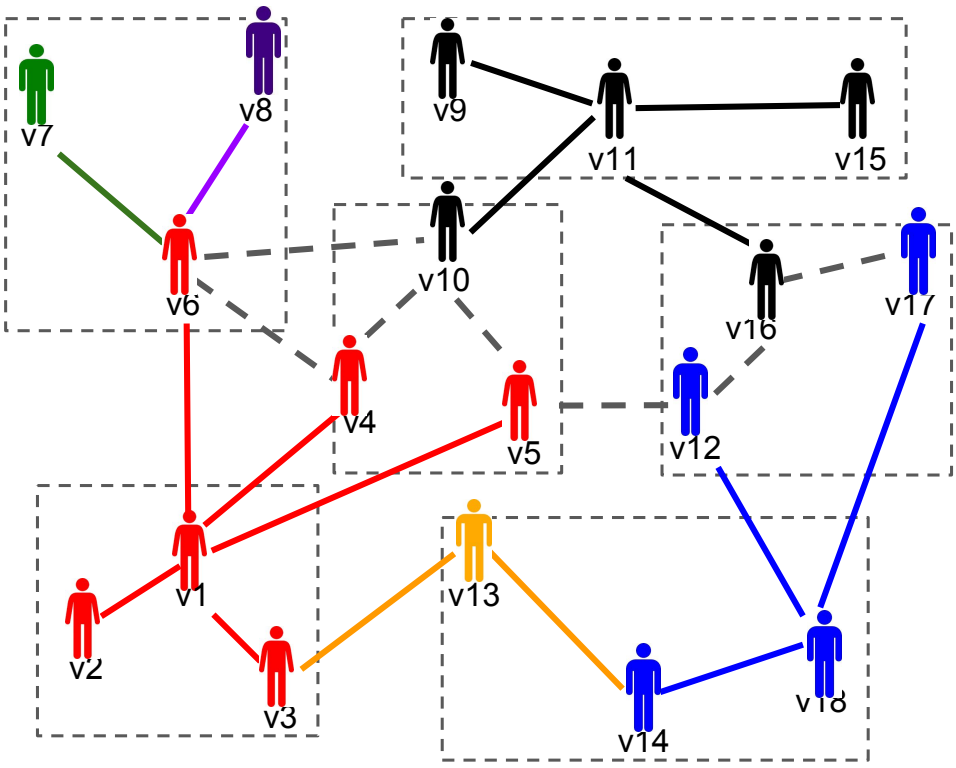


Figure 4.5: Batch Attribute Retrieval

v' ($AttrAddr[v']$) is set to $addr$ (line 16-19).

Finally, for every edge e, if it was not stored before (line 22) i.e. dotted line edges, content of it is put into b, $isStored[e]$ is set to $true$, b is written to disk, and the address $addr$ is put into $AttrAddr[e]$.

Figure 4.5 shows the effect of Algorithm 5. In Figure 4.5, contents of v_1 and its adjacent edges and vertices are put into a disk block. The same happens for v_{11} and v_{18}. For v_7, since contents of its adjacent vertex (i.e. v_6) were stored in other disk block before, only contents of v_7 and $e_{(v_7,v_6)}$ are stored in the same disk block. The same happens for v_8 and v_{13}. Finally, for $e_{(v_4,v_6)}$, $e_{(v_4,v_{10})}$, $e_{(v_6,v_{10})}$, $e_{(v_5,v_{10})}$, $e_{(v_5,v_{12})}$, $e_{(v_{12},v_{16})}$, and $e_{(v_{16},v_{17})}$, since contents of both of their left and right vertices were stored in other disk blocks, contents of each of them are stored in a different disk block. In total, 13 disk blocks are used to store contents of all vertices and edges.

Algorithm 5 Index Construction for Batch Attribute Retrieval

```
1: procedure ConBatchAttrIndex(G_h, bFile)
2:     for all v ∈ G_h do
3:         block b;  address addr
4:         if isStored[v] = true then
5:             continue
6:         end if
7:         b ← PackAttrnCount(b, v, G_h)
8:         isStored[v] ← true
9:         for all v' ∈ v.adjList do
10:            if isStored[v'] ≠ true then
11:                b ← PackAttrnCount(b, v', G_h)
12:            end if
13:            if isStored[e_(v,v')] ≠ true then
14:                b ← PackAttrnCount(b, e_(v,v'), G_h)
15:                isStored[e_(v,v')] ← true
16:            end if
17:        end for
18:        addr ← IOInfo_w(b, bFile)
19:        AttrAddr[v] ← addr
20:        for all v' ∈ v.adjList do
21:            if isStored[v'] ≠ true then
22:                AttrAddr[v'] ← addr
23:                isStored[v'] ← true
24:            end if
25:        end for
26:    end for
27:    for all e ∈ G_h do
28:        b.clear()
29:        if isStored[e] ≠ true then
30:            b ← PackAttrnCount(b, e, G_h)
31:            isStored[e] ← true
32:            addr ← IOInfo_w(b, bFile)
33:            AttrAddr[e] ← addr
34:        end if
35:    end for
36: end procedure
```

4.6.2 Query Algorithm

Given $AttrAddr$ (line 1), BFS can be used to answer set attribute constraint reachability queries. Algorithm 6 is used to replace Algorithm 2 in Section 4.4. The differences between Algorithm 6 and Algorithm 2 is that if an attribute is not verified (line 4), based on the address in $AttrAddr$, a block is retrieved from disks (line 6), and all attributes contained in that block are verified (line 6) and corresponding entries in $satTable$ are updated (line 9,12). Hence, 1 I/O can be used to verify multiple vertex and/or edge attributes that can be fit in the same disk block.

Algorithm 6 Check Constraint using Batch Attribute Retrieval

```
1:  procedure CheckConstraint_B(v, C_v, G_h, AttrAddr)                ▷ same for e
2:      h ← GetAttrHash(v, G_h)                                      ▷ in-memory operation
3:      sat ← satTable[h]                                            ▷ return φ if not exist
4:      if sat = φ then
5:          block b ← IOInfo_r(bFile, AttrAddr[v])
6:          for all (v, attr, count) ∈ b do
7:              if count = 1 then
8:                  if CheckAttr(attr, C_v) = true then
9:                      satTable.insert(h, true)
10:                     return true
11:                 else if CheckAttr(attr, C_v) = false then
12:                     satTable.insert(h, false)
13:                     return false
14:                 end if
15:             else
16:                 attr ← IOAttr(v, G_h)                            ▷ get Attr. of v from disk
17:                 if CheckAttr(attr, C_v) = true then
18:                     return true
19:                 else
20:                     return false
21:                 end if
22:             end if
23:         end for
24:     else
25:         return sat
26:     end if
27: end procedure
```

4.6.3 Bounding I/O

(analyze the worst case and expected I/O bound using block size, attribute size, attribute domain size)

THEOREM 5. **[Worst Case I/O Bound for Set Attr. Query]** *The worst case I/O complexity is* $O(\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil + \left\lceil \frac{r}{B} \right\rceil |V|)$*, where* r *is the size of vertex id, attribute and count.*

Proof. In the worst case, the whole graph has to be traversed. At the same time, every hash value is not the same and has to be verified. Therefore, the worst case I/O is the number of disk block used to store the vertex/edge ids, attributes, and counts of all hash values i.e. the number of I/O used by Algorithm 5.

In Algorithm 5 (line 2 to 19), in the worst case, every vertex v_i puts vertex/edge ids, attributes, and hash value counts (size is r) of itself and all adjacent vertices and edges into $\left\lceil \frac{r(2d_i+1)}{B} \right\rceil$ disk block(s), where d_i is the degree of v_i. Hence, the worst case number of I/O is the sum of I/O for every vertex, which is $\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil$.

In Algorithm 5 (line 20 to 26), if edge id, attributes, and hash value count of an edge is not stored before (in line 2 to 19), they are put into $\left\lceil \frac{r}{B} \right\rceil |V|$ disk blocks. Although the for-loop (line 20) has size $|E|$, the number of I/O (line 25) is always smaller than or equal to $|V|$. That is because edge id, attributes, and hash value count of an edge is not stored only when vertex ids, attributes, and hash value count of both vertices of the edge is stored. Therefore, the number of such edges is smaller than or equal to $|V|$.

Since the two for-loop (line 2 to 19 and line 20 to 26) is in parallel, the number of I/O of Algorithm 5 is the sum of $\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil$ and $\left\lceil \frac{r}{B} \right\rceil |V|$, which is $O(\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil + \left\lceil \frac{r}{B} \right\rceil |V|)$. □

Comparing to BFS with $O(\left\lceil \frac{r}{B} \right\rceil |V| + \left\lceil \frac{r}{B} \right\rceil |E|)$ I/O, batch attribute retrieval is better in term of worst case complexity when:

$$\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil + \left\lceil \frac{r}{B} \right\rceil |V| < \left\lceil \frac{r}{B} \right\rceil |V| + \left\lceil \frac{r}{B} \right\rceil |E|$$

$$\sum_{i=1}^{|V|} \left\lceil \frac{r(2d_i+1)}{B} \right\rceil < \sum_{i=1}^{|E|} \left\lceil \frac{r}{B} \right\rceil$$

Table 4.2 is a summary of index construction and query CPU time and space worst case complexity. For index construction, the CPU time complexity is $O(|V|d_{max} + |E|)$ as the nested for-loop (Algorithm 5, line 2 to 19) executes $|V|d_{max}$ times and the single for-loop (line 20 to 26) executes $|E|$ times; the space complexity is $O(|G|)$ as the whole graph is only stored once. For set attribute constraint query, the CPU time complexity is $O(\left\lceil \frac{B}{r} \right\rceil |V|+|E|)$ as in the worst case, every vertex and edge has to be visited and when visiting a vertex, for every block retrieved from disk, $O(\left\lceil \frac{B}{r} \right\rceil)$ steps are needed to parse the contents (ids, attribute, and hash value count); the space complexity is $O(|G|)$ as only the graph topology, hash value, $isStored$, and $AttrAddr$, which have the same size as the graph topology, are needed to be in memory.

Table 4.2: Worst Case Time and Space Complexity

Operation	**CPU Time**	**Disk Space**
Index Construction (offline)	$O(\|V\|d_{max} + \|E\|)$	$O(\|G\|)$
Operation	**CPU Time**	**Memory Space**
Set Attr. Constraint Query	$O(\left\lceil \frac{B}{r} \right\rceil \|V\| + \|E\|)$	$O(\|G\|)$

4.7 OPTIMIZATION: BETTER UTILIZATION OF PRIMARY STORAGE

Distribution of attribute constraints in queries is usually not uniform. Some attribute constraints appear more often while some seldom appear. A similar situation happens for query execution time. Queries with some attribute constraints take longer time too execute while queries with some other attribute constraints finish quickly. Therefore, we can see that some attribute constraints are actually more important than some others, in term of efficiency. In order to speed up execution time of more important attribute constraints, we propose a workload-aware hash value approach to fully utilize all available memory. The idea of

this approach is to compute extra hash values using a subset of attributes and put those hash values into memory. The corresponding index structure of those hash values is also constructed and stored in the disk. We assume that a set of historical query workloads W that contains query attribute constraints and a total number of I/O are given.

Definition 22. [**Historical Query Workloads**] *W is a set of tuples, which consists of a subset of $A(v)$ and number of I/O, and we assume that the number of I/O is measured when $A(v)$ is used for computing hash values and an index.*

4.7.1 Use of Extra Hash Values

When only a subset of $A(v)$ is involved in an attribute constraint query, hash values and an index that are constructed based on only the subset of $A(v)$ can be used to answer the query. For example, if an attribute constraint query only involves $Country$ and Job, hash values h_{ex} and an index in_{ex} that are constructed based on $Country$ and Job can be used to answer the query. Similar to the index in Figure 4.2, in_{ex} would have also 3 columns - h_{ex}, attributes, and count. However, h_{ex} is computed using only $Country$ and Job; attribute column only consists of $Country$ and Job; and the count is based on h_{ex}. in_{ex} is still stored in the disk, and all h_{ex} are stored in memory.

Given the same attributed graph, A_{diff} is smaller when fewer attributes are taken into account. For example, A_{diff} of an attributed graph with attributes $Country, Job, Religion$ is bigger than A_{diff} of the same attributed graph with attributes $Country, Job$. Therefore, if the attribute constraint query only involves $Country$ and Job, the I/O complexity is better when hash values and index are computed based on less number of attributes. Hence, if more hash values and indexes using different subsets of $A(v)$ are computed, appropriate hash values and indexes can be used for answer queries so that better I/O cost can be achieved. Since memory budget for storing extra hash values is limited, in the next section, we will present strategies for choosing the subset of $A(v)$ for computing extra hash values.

4.7.2 Attribute Selection Strategy

In this section, we will present 2 strategies for choosing subsets of $A(v)$ for computing extra hash values. We also proposed another strategy (strategy 3) for consideration. Let m be the number of the set of hash values that can store in memory. In a set of hash values, all hash values are computed using the same subset of $A(v)$ and the size of the set is $|V|$. Since we have to guarantee that any query has to be answered, we must have hash values that are computed using $A(v)$. Hence, we can only have $(m-1)$ extra sets of hash values in memory.

The 2 strategies are proposed based on Lemma 2.

Lemma 2. *Hash values and index constructed using S can achieve best number of I/O for attribute constraint queries that involve S as attribute constraints when hashing scheme is used (Section 4.4), where S is a subset of $A(v)$.*

Proof. Hash values and index constructed using proper subset of S cannot be used for answering attribute constraint queries with S as attribute constraints-(♣). Attributed graph with attributes which are a proper superset of S has larger A_{diff} than attributed graph with S as attributes-(♠). Because of (♣) and (♠), Lemma 2 is true. □

4.7.2.1 Strategy1: Most Frequent The first strategy is to pick subsets of $A(v)$ that are top-$(m-1)$ most frequently exist in historical query workload. Figure 4.6 is an example. Suppose m is 4 and there are 3 attributes (A_1, A_2, A_3) for every vertex. In Figure 4.6, the number of queries in W that have attribute constraint $A_1, A_2, A_3, A_1A_2, A_1A_3, A_2A_3$, and $A_1A_2A_3$ are 10, 20, 4, 5, 9, 2, and 0 respectively. Since $A_1A_2A_3$ must be picked, we can only pick three more subsets of attributes. The three most frequently exist subsets of attributes, which are A_1, A_2, and A_1A_3, are picked.

Theorem 6 is the theoretical guarantee of strategy 1.

THEOREM 6. *Strategy 1 guarantees that most of the historical workloads can achieve the best I/O cost, given m as memory budget.*

Proof. Strategy 1 picks top-$(m-1)$ most frequent attribute constraints in W-(♢). Lemma 2 ensures that using hash values and indexes computed by those $m-1$ subset of $A(v)$ for

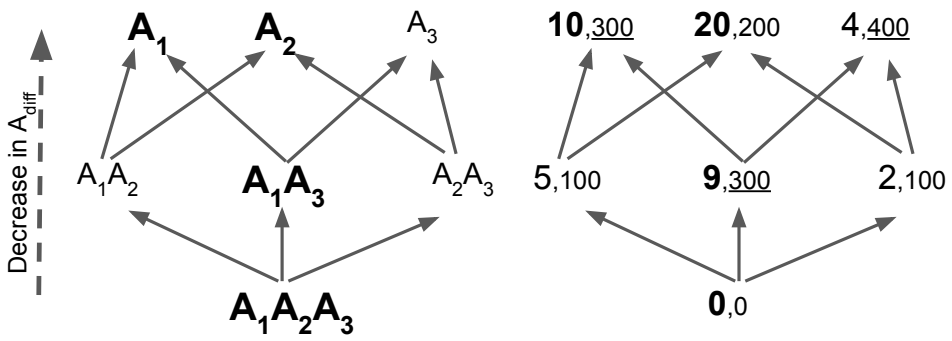


Figure 4.6: All Combinations of 3 Attributes

answering queries can achieve best number of I/O for attribute constraint queries that involve those $m-1$ subset of $A(v)$ as attribute constraints-($\heartsuit$). Because of ($\diamondsuit$) and ($\heartsuit$), we can conclude that this theorem is true. □

4.7.2.2 Strategy 2: Maximum I/O The second strategy is to pick subsets of $A(v)$ that are top-$(m-1)$ maximum number of I/O in historical query workloads. Figure 4.6 is an example. Suppose m is 4 and there are 3 attributes (A_1, A_2, A_3) for every vertex. In Figure 4.6, the maximum number of I/O of queries in W that have attribute constraint $A_1, A_2, A_3, A_1A_2, A_1A_3, A_2A_3$, and $A_1A_2A_3$ are 300, 200, 400, 100, 300, 100, and 0 respectively. Since $A_1A_2A_3$ must be picked, we can only pick three more subsets of attributes. The three maximum number of I/O subsets of attributes, which are A_1, A_3, and A_1A_3, are picked.

Theorem 7 is the theoretical guarantee of strategy 2.

THEOREM 7. *Strategy 2 picks the subsets of $A(v)$ that can minimize the maximum number of I/Os of all queries in W (IO^i_{max}), given m as memory budget constraint.*

$$IO_{max} = \ min\{IO^1_{max}, IO^2_{max}, ..., IO^{C^d_{m-1}}_{max}\}$$

$$where \ \ IO^i_{max} = max\{IO^i_1, ..., IO^i_{|W|} | Comb_i, m\},$$

$$Comb_i \ \ is \ a \ \ subset \ of \ A(v) \ with \ m-1 \ elements.$$

Proof. Without lose of generality, we assume that there are k different subsets of $A(v)$ in W and their I/O cost are $IO_1, ..., IO_k$. We further assume that $IO_1 \geq IO_2 \geq ... \geq IO_k$.

Let max_1 be the maximum number of I/O when the subsets of $A(x)$ picked by strategy 2 are used. Strategy 2 picks the first $m-1$ subsets and based on Lemma 2, queries in W with any of those $m-1$ subsets as attribute constraints can achieve best I/O cost $(IO'_1, .., IO'_{m-1})$. Therefore, max_1 can be defined as:

$$max_1 = max(IO'_1, ..., IO'_{m-1}, IO_m)$$

Assume that strategy 3 picks the same subsets of $A(v)$ as strategy 2, except that it replaces a subset of $A(v)$ (subset y) picked by strategy 2 with a subset of $A(v)$ (subset x) that is not picked by strategy 2. Let max_2 be the maximum number of I/O when the subsets of $A(x)$ picked by strategy 3 are used. Strategy 3 picks the first $m-1$ subsets and subset x, except subset xy and based on Lemma 2, queries in W with any of those $m-1$ subsets as attribute constraints can achieve best I/O cost $(IO'_1, ..., IO'_{x-1}, IO_x, IO'_{x+1}, ..., IO'_{m-1})$. Therefore, max_2 can be defined as:

$$max_2 = max(IO'_1, ..., IO'_{x-1}, IO_x, IO'_{x+1}, ..., IO'_{m-1}, IO_m)$$

We can divided the proof into below cases:

1. $max_1 > IO_m$,
   a. if $max_1 = IO'_x$, then $max_2 = IO_x \geq max_1 = IO'_x$. That is because $IO_x \geq IO'_x \geq IO'_1, ..., IO'_{m-1} > IO_m$.
   b. if $max_1 > IO'_x$, and if $max_2 = IO_x$, then $max_2 > max_1$. That is because $IO_x > IO'_1, ..., IO'_{x-1}, IO'_{x+1}, ..., IO'_{m-1}$ and $IO_x \geq IO'_x$.
   c. if $max_1 > IO'_x$, and if $max_2 > IO_x$, then $max_2 = max_1$. That is because $max_2 > IO_x$ implies that both max_2 and max_1 are $\in \{IO'_1, ..., IO'_{x-1}, IO'_{x+1}, ..., IO'_{m-1}\}$.
2. $max_1 = IO_m$,
   a. $max_2 = IO_m$, then $max_1 = max_2$.
   b. $max_2 > IO_m$, then $max_2 > max_1$. That is because $max_1 = IO_m$.

Therefore, max_1 is always less than or equal to max_2. Since IO_x can be any subset, we pick IO_x to be $IO'_x \geq IO'_m, ..., IO'_{x-1}, IO'_{x+1}, .., IO'_k$. Then, for strategy 3, no matter how many subsets are replaced, the above analysis is still true. Hence, we can conclude that this theorem is true. □

4.7.2.3 Strategy 3: Lowest Entropy First The third strategy is to pick subsets of $A(v)$ that have the lowest attribute value entropy. Since the attribute value entropy is low, the number of different hash values (i.e., V_{diff}) is also small. Hence, the extra hash values would be very beneficial to queries that only consider attributes which are subset of $A(v)$.

4.8 EXPERIMENTAL RESULT

In this section, we will first describe the experiment settings and present graph information for our experiments. Then, we will look into the performance of our techniques for both application queries and exhaustive parameter tuning.

4.8.1 Experiment Setup and Dataset

All experiments were performed using C++ implementations under a Linux machine with an Intel 4GHz CPU (4-core), 16 GB of memory, and 1 TB solid state drive with 512k block size. We used Murmur hash [20] as our hash function.

For all experiments, we build super-graphs with 50 super-vertex for $fb - bfs1$ and 1000 super-vertex for all other datasets by using a naive clustering approach. The naive clustering approach first picks 50 or 1000 random vertices and performs BFS with the 50 or 1000 vertices as the sources to form 50 or 1000 clusters. For real applications, other clustering techniques (e.g. [69]). We draw 1000 uniform samples from each super-vertex as the synopsis.

4.8.1.1 Baselines We study the efficiency of our new approach and heuristic search technique by applying them on a prevalent graph traversal algorithm. Due to the issues

of A^* mentioned in Section 3.5.2 and the popularity of BFS, we used BFS as our graph traversal algorithm in our experiments. We compare ordinary BFS, BFS with only our new constraint verification approach ($Hash + BFS$), BFS with our new constraint verification approach and heuristic search ($Hash+BFS_{Heur.}$), and another baseline solution (LCR_{single}).

LCR_{single} is a modification of the approach in [3]. LCR_{single} uses 0.5k blocks to build an index I_{LCR} for every dimension of the multi-dimensional attribute using the solution in [3] and stores all indexes in disk. During query time, LCR_{single} uses I_{LCR} to compute reachability query for attribute dimensions that are involved in the attribute constraint dimension-by-dimension until there is one attribute dimension cannot return a yes or all involved attribute dimensions return yes. LCR_{single} may offer incorrect answers.

4.8.1.2 Datasets Table 4.4 is a summary of our graph dataset. In order to control the number of attributes and attribute domain sizes, we generate attributes (Table 4.3) based on vertex and edge attribute in Facebook graph-API [70]. For graphs with 30 vertex attributes, we just repeat the 10 vertex attributes for 3 times.

Table 4.3: Attributes

Vertex Attribute	**Domain Size,Distribution (μ,σ)**
AgeGroup	*10, gau(5,2.5)*
Education	*5, gau(3,1.25)*
Gender	*2, uni.*
HomeCountry	*100, gau(50.25)*
Interested in	*3, uni.*
Languages	*50, gau(25,12.5)*
Relationship status	*2, uni.*
Religion	*20, gau(10,5)*
Work	*50, uni.*
Political	*10, gau(5,2.5)*
Edge Attribute	**Domain Size,Distribution**
isFamily	*2, uni.*
isFriend	*2, uni.*
isFriendRequest	*2, uni.*
isSubscribers	*2, uni.*
isSubscribedto	*2, uni.*

Table 4.4: Dataset Information

Real Graph	**Num of Vertex**	**Num of Edge**
dblp [71]	*0.31m*	*1.05m*
fb-bfs1 [72]	*1.18m*	*29.78m*
twitter-0.25 [73]	*52.58m*	*490.8m*
Synthetic Graph	**Num of Vertex**	**Num of Edge**
Small-World [71]	*200m*	*1b*
	100m	*500m*
	50m	*250m*
	10m	*50m*

Table 4.5: Parameter Setting

Parameter	**Value**
Num V Attr	**10**,*30*
Num E Attr	**5**
Num V Constraint	*1*, **5**,*10*,*15*,*20*
Num E Constraint	**1**,*2*,*3*,*4*, *5*
Num of Super-vertex	**15**,**50**,**1000**
Num of Sample per Super-vertex	**100**

4.8.2 Performance for Application Queries

Table 4.8 and 4.7 shows the performance of our techniques for 5 different application queries defined in Table 4.6 for different application scenarios. We can see that when the attribute constraint is very specific (e.g. Q4 and Q5), the performance of $Hash + BFS_{Heur.}$ and $Hash + BFS$ in the smaller graph ($fb - bfs1$) are basically the same. However, for other queries, $Hash + BFS_{Heur.}$ can always outperform other approaches.

Table 4.6: Application Queries

Query	**Application**
Q1. **Select** paths **where** all vertices along the path have HOMECOUNTRY= *'in N/S America'* **and** SRC=*'T'* **and** DEST=*'L'*	decides whether target person T is related to terrorist L.
Q2. **Select** paths **where** all vertices along the path have AGE$\leq$ 60 **and** (LANGUAGE=*'English'*, *'Chinese'* **or** *'German'*) **and** SRC=*'T'* **and** DEST=*'H'*	decides whether target person T is related to hacker H.
Q3. **Select** paths **where** all vertices along the path have WORK=*'All Engineering Related'* **and** all edges along the path have (isFamily=*'true'*, isFriend=*'true'* **or** isFriendRequest=*'true'*) **and** SRC=*'T'* **and** DEST=*'M'*	decides whether job candidate T is related to manager M in rival company.
Q4. **Select** paths **where** all vertices along the path have POLITICAL$\neq$*'democratic'* **and** HOMECOUNTRY=*'United States'* **and** SRC=*'T'* **and** DEST=*'P'*	decides whether target person T is related to party leader P.
Q5. **Select** paths **where** all vertices along the path have (HOMECOUNTRY= *'United State'* **or** *'Canada'*) **and** all edges along the path have isFamily=*'true'* **and** SRC=*'T'* **and** DEST=*'S'*	decides whether target person T is related to a suspect S.

Table 4.7: Results for Specific Queries ($twitter - 0.25$)

Q.	$Hash + BFS_{Heur.}$	$Hash + BFS$	BFS	LCR_{single}
1	3.57sec	14.46sec	35.91sec	30.22sec
2	6.83sec	25.90sec	84.74sec	84.30sec
3	4.05sec	16.63sec	58.16sec	39.19sec
4	0.46sec	0.71sec	1.18sec	1.13sec
5	1.42sec	2.69sec	6.09sec	5.11sec

Table 4.8: Results for Specific Queries ($fb-bfs1$)

Q.	$Hash+BFS_{Heur.}$	$Hash+BFS$	BFS	LCR_{single}
1	0.52sec	0.56sec	14.84sec	15.02sec
2	0.42sec	0.47sec	12.26sec	11.79sec
3	1.01sec	1.20sec	22.26sec	20.56sec
4	0.01sec	0.01sec	0.01sec	0.01sec
5	0.01sec	0.01sec	0.07sec	0.07sec

4.8.3 Performance on Real Graphs

In this section, we will present the experimental results of 2 real graphs - fb-$bfs1$ and $twitter-0.25$ with analysis.

4.8.3.1 Experiment Design In this experiment, we try to vary the number of vertex and edge attribute constraints so as to observe the change of overall running time and number of SSD I/O of our techniques. Parameter settings are summarized in Table 4.5 and bold words are the default setting.

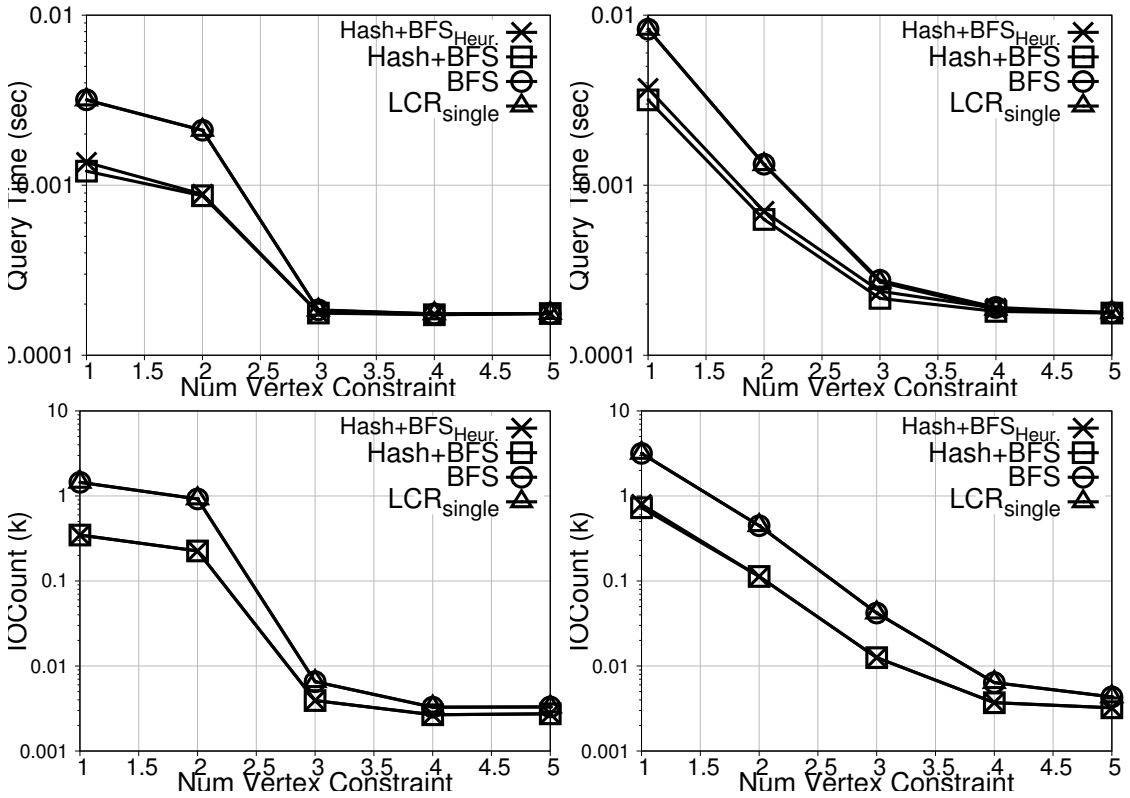
The number of vertex and edge attribute constraint in this experiment are defined as

$$\sum_{i=1}^{d_v}(|D_i^v|-|S_i^v|) \; and \; \sum_{i=1}^{d_e}(|D_i^e|-|S_i^e|) \tag{4.1}$$

(D_i^v, D_i^e, S_i^v, and S_i^e are the same as D_i, S_i in Definition 11,12). The default number of attributes is ten as usually attribute constraints do not involve more than ten attributes. Hence, we do not need to compute hash values using more than ten attributes. However, which 10 attributes to pick is query workload dependent, and we will leave that for future studies. We observe the behavior of different solutions by fixing the number of vertex and edge attribute to 10 and 30 with domain sizes and distributions as describe in Section 6.5.1 and varying the number of vertex or edge constraints (Figure 4.11, 4.13, Figure 4.15 and 4.17). We also repeat the experiment by doubling the domain sizes and mean and s.d. of distributions (Figure 4.12, 4.14, Figure 4.16 and 4.18).

4.8.3.2 Result Results for varying number of vertex/edge constraints using default setting are shown in Figure 4.7(a), 4.9(a), Figure 4.11(a), 4.13(a), Figure 4.15(a) and 4.17(a). The general trend of the average running time and the number of I/O when increasing the number of attribute constraint is decreasing for all settings. That is because in general, the larger the number of attribute constraints, the faster the graph traversal stop. For all settings, $Hash + BFS_{Heur.}$ always has a smaller average running time and number of I/O than $Hash+BFS$,LCR_{single}, and BFS, which is the contribution of our new constraint verification approach and efficiency heuristic search technique. The only exception is the *dblp* dataset. Since the *dblp* dataset is a relatively small graph, the performance differece between $Hash + BFS_{Heur.}$ and $Hash + BFS$ is not significant. The performance of LCR_{single} is close to BFS since local transitive closures cannot vastly reduce execution time. Damaged by the extremely inefficient transitive closure construction time ($O(|V|^3)$ [3]) and storage space of LCR_{single}, blocks for computing local transitive closure cannot be large (especially for high degree undirected graphs), which leads to relatively small hopping from vertex to vertex during graph traversal and as a result, a lot of disk-based graph traversals are needed. In addition, for an undirected graph with non-small attribute domain size, the transitive closure is rather bulky, which leads to costly disk retrieval of transitive closures for border vertices.

The increase of domain size and number of vertex attributes (Figure 4.7(b), 4.8, 4.9(b), 4.10, Figure 4.11(b), 4.12, 4.13(b), 4.14, Figure 4.15(b), 4.16, 4.17(b), and 4.18) do not contribute much to the increase in execution time of all approaches. Firstly, BFS is not really affected by the increase in number of attributes and domain size as it always does vertex-by-vertex and edge-by-edge constraint verification; similar situation happens to LCR_{single} as LCR_{single} only accesses index of attributes involved in attribute constraints. Secondly, for $Hash+BFS_{Heur.}$ and $Hash+BFS$, we can only see a slightly shift up of their curves. That is due to the fact that the number of edges is a lot larger than number of vertices, which makes the time for edge constraint verification occupies a certain portion of the total running time.



(a)Num V Attr.=10 (b)Num V Attr.=30

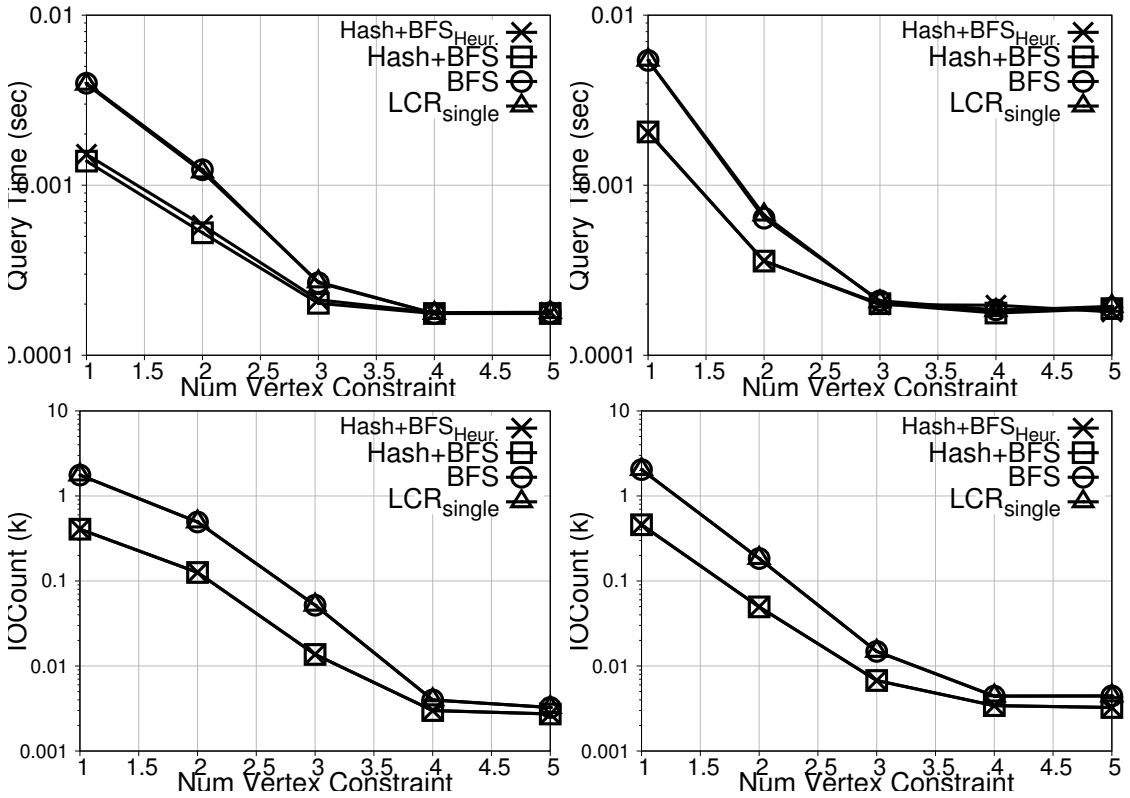
Figure 4.7: **[dblp]-**Vary # of V Const. with Org. Dom.

4.8.4 Performance on Synthetic Graphs

In this experiment, we used default setting and attributes in Table 4.5 and Table 4.3 for $Small - World$ graphs with different sizes. In Figure 4.19, we can see that in general, average running time and number of I/O increase with graph size and $Hash + BFS$ and $Hash + BFS_{Heur.}$ can successfully reduce the average running time and number of I/O.

4.8.5 Performance of Optimization Techniques

4.8.5.1 Batch Attribute Retrieval In this experiment, we evaluate the performance of the batch attribute retrieval technique using our default setting. Table 4.9 shows the changes of the average number of IO of three different approaches - $Hash + BFS_{Heur}$, $Hash + BFS$,



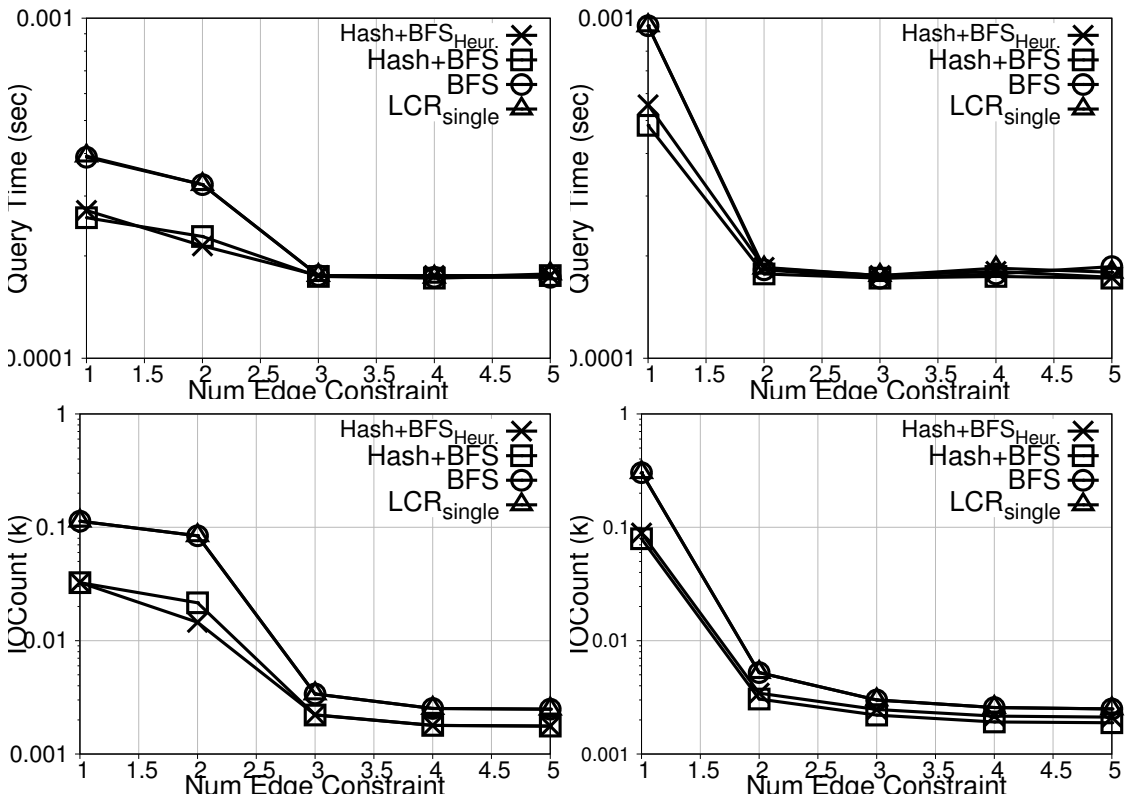
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.8: **[dblp]-**Vary # of V Const. with Double Dom.

and BFS. In Table 4.9, the value on the left of $\rightarrow$ is the number of IO before using batch retrieval while the value on the right of $\rightarrow$ is the number of IO after using batch retrieval. We can see that batch attribute can effectively reduce ethe number of IO.

Table 4.9: Batch Retrieval Comparison

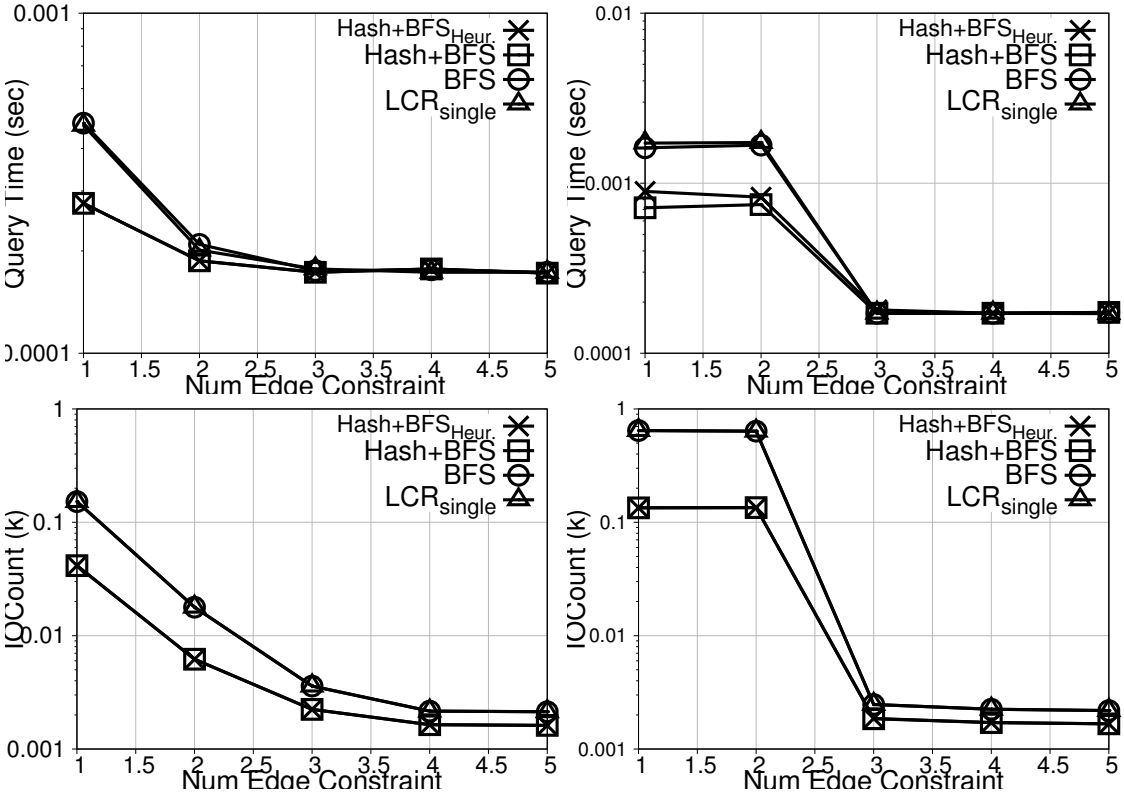
Graph	$Hash + BFS_{Heur}$	$Hash + BFS$	BFS
dblp [71]	*2.71→0.61*	*2.71→0.61*	*3.07→0.7*
fb-bfs1 [72]	*619k→79k*	*690k→87k*	*14579k→ 228k*
twitter-0.25 [73]	*9544k→865k*	*24658k→2274k*	*1068412k→ 11684k*



(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.9: **[dblp]-**Vary # of E Const. with Org. Dom.

4.8.5.2 Use of Extra Hash Values In this experiment, we compare the performance of extra values for answering queries using our default setting. We first build 30-attribute hash value index. Then, we build a 10-attribute hash value index as an extra set of hash value. We compare the query efficiency of using both indexes to answer queries that are generated under our default setting. In Table 4.10, we can see that the execution time is improved by the extra set of hash values.

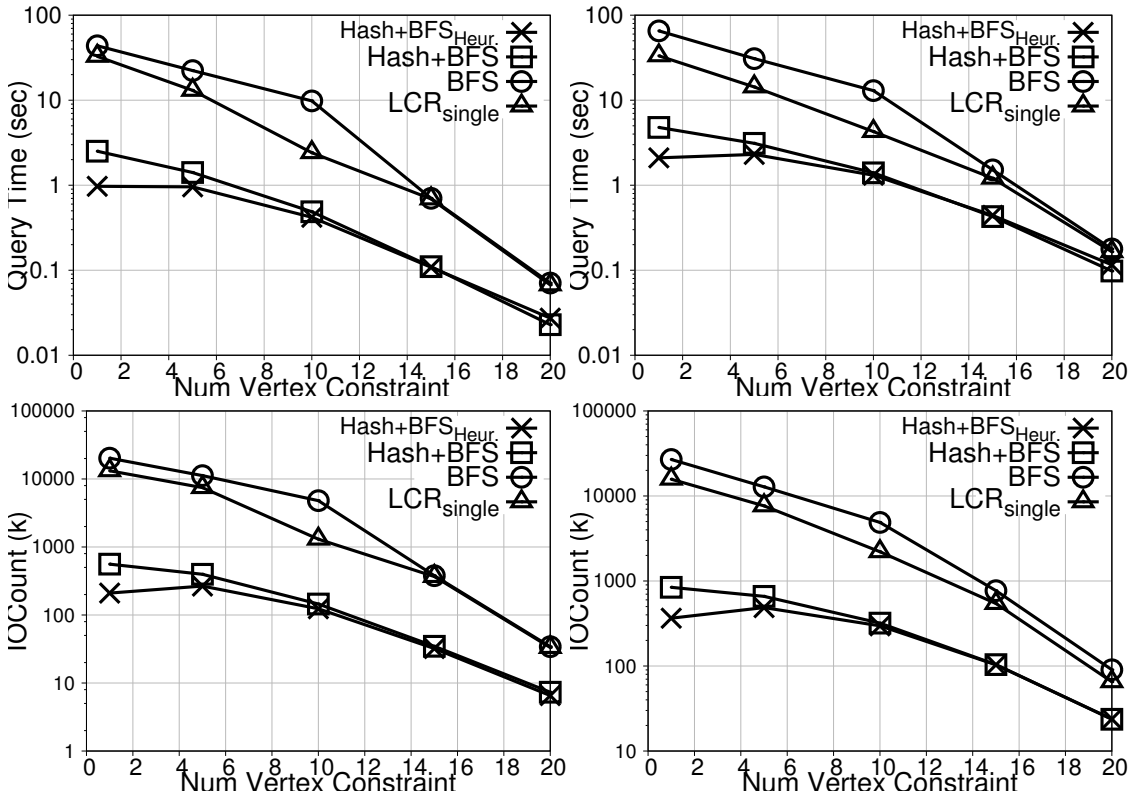


(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.10: **[dblp]-**Vary # of E Const. with Double Dom.

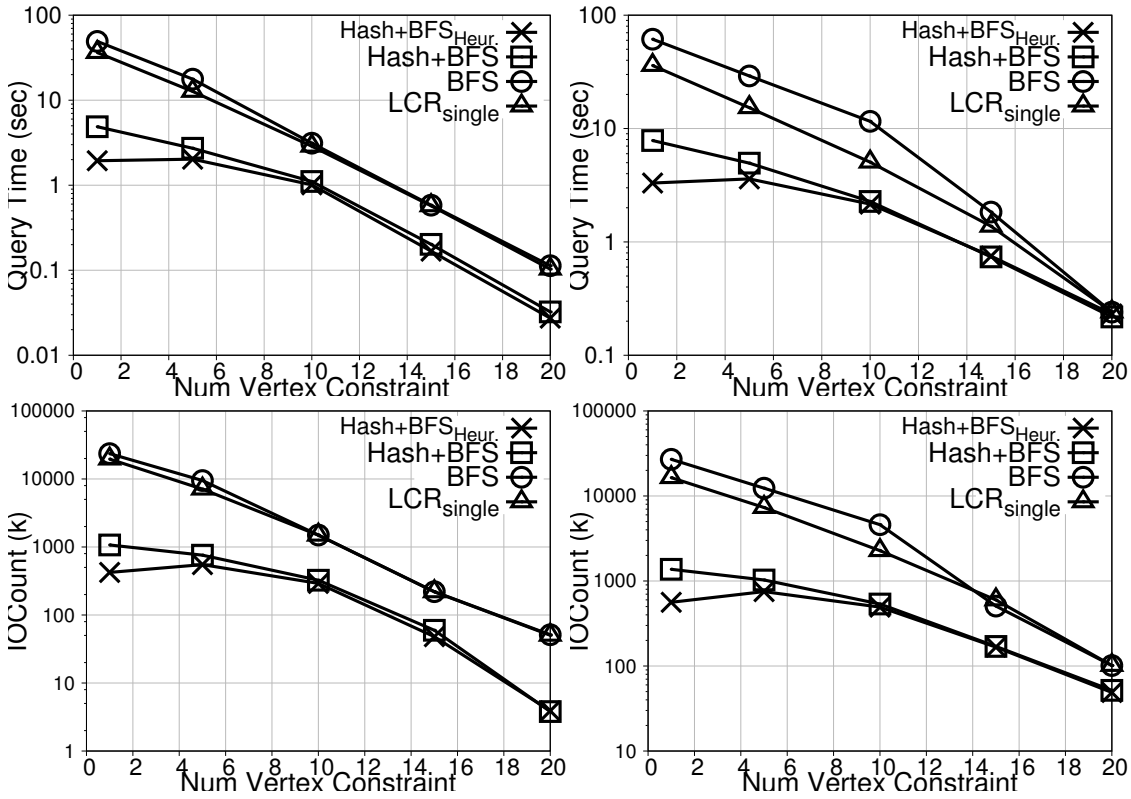
Table 4.10: Effect of Extra Hash Values

	$Hash + BFS_{Heur}$		$Hash + BFS$	
Graph	With Extra Hash	Without Extra Hash	With Extra Hash	Without Extra Hash
dblp [71]	*0.000702sec*	*0.000885sec*	*0.000629sec*	*0.000867sec*
fb-bfs1 [72]	*0.955sec*	*1.54sec*	*1.41sec*	*2.07sec*
twitter-0.25 [73]	*3.09sec*	*3.99sec*	*21.35sec*	*23.89sec*



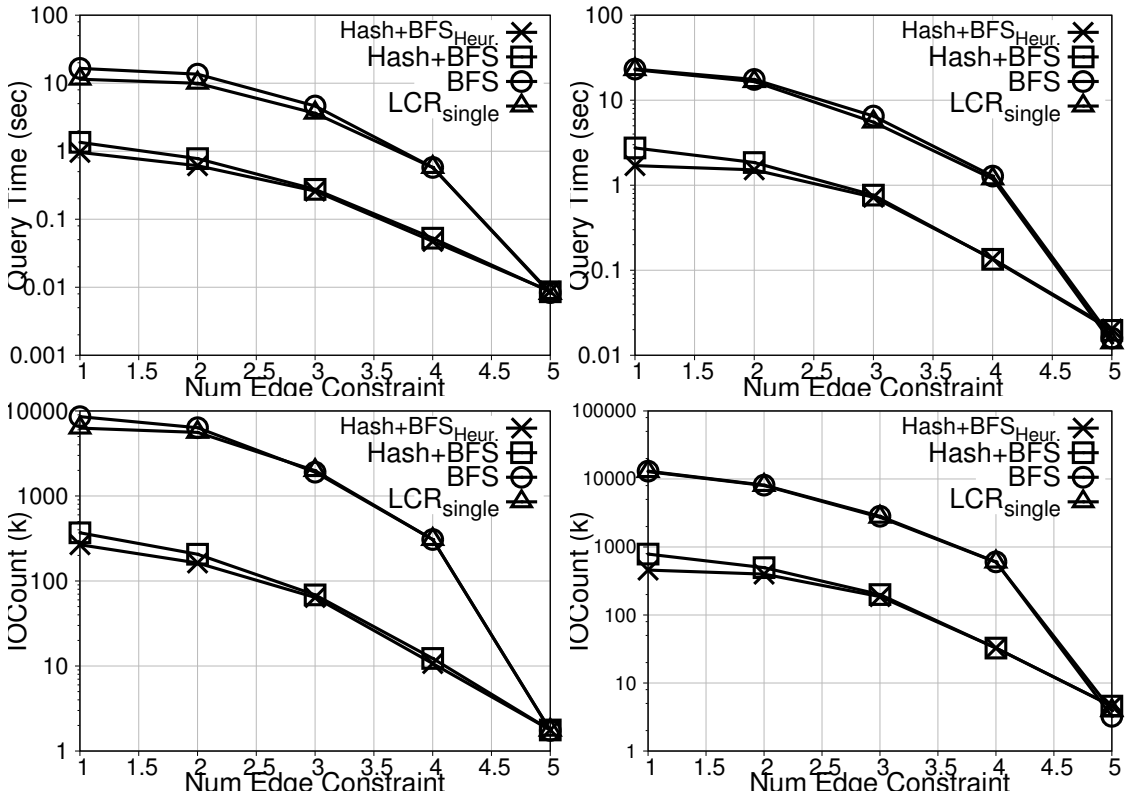
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.11: [**fb-bfs1**]-Vary # of V Const. with Org. Dom.



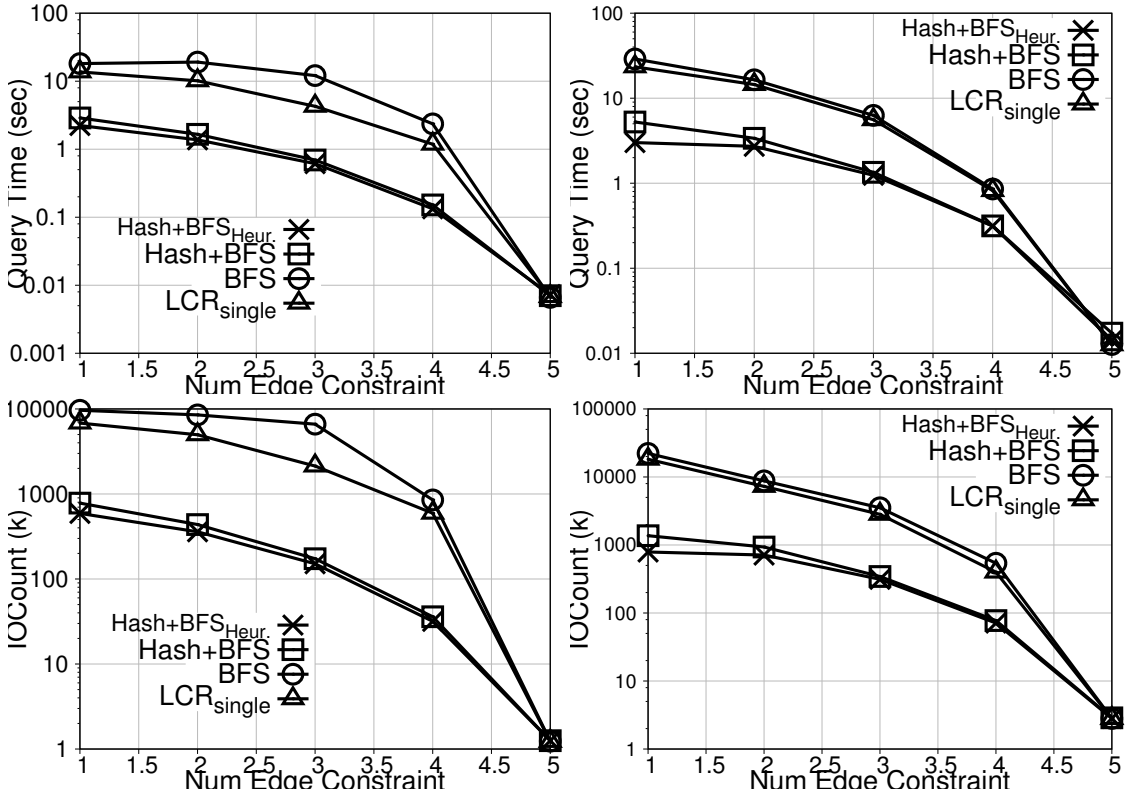
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.12: **[fb-bfs1]-**Vary # of V Const. with Double Dom.



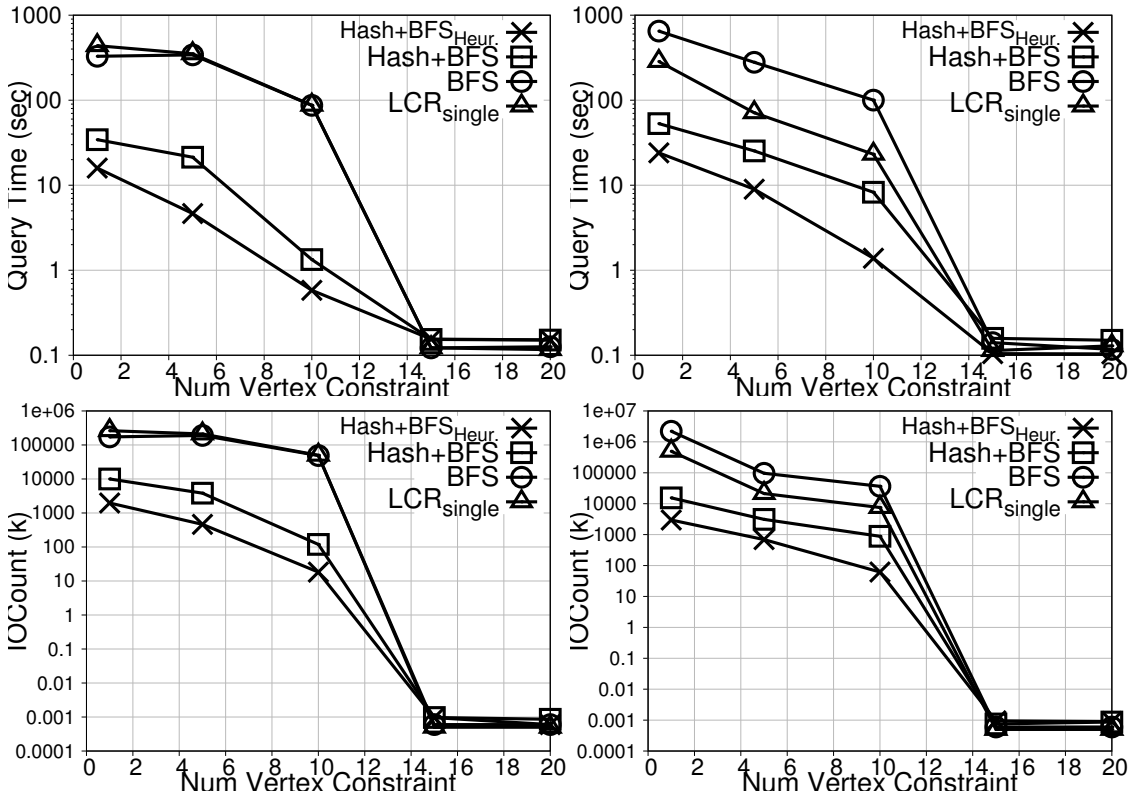
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.13: **[fb-bfs1]-**Vary # of E Const. with Org. Dom.



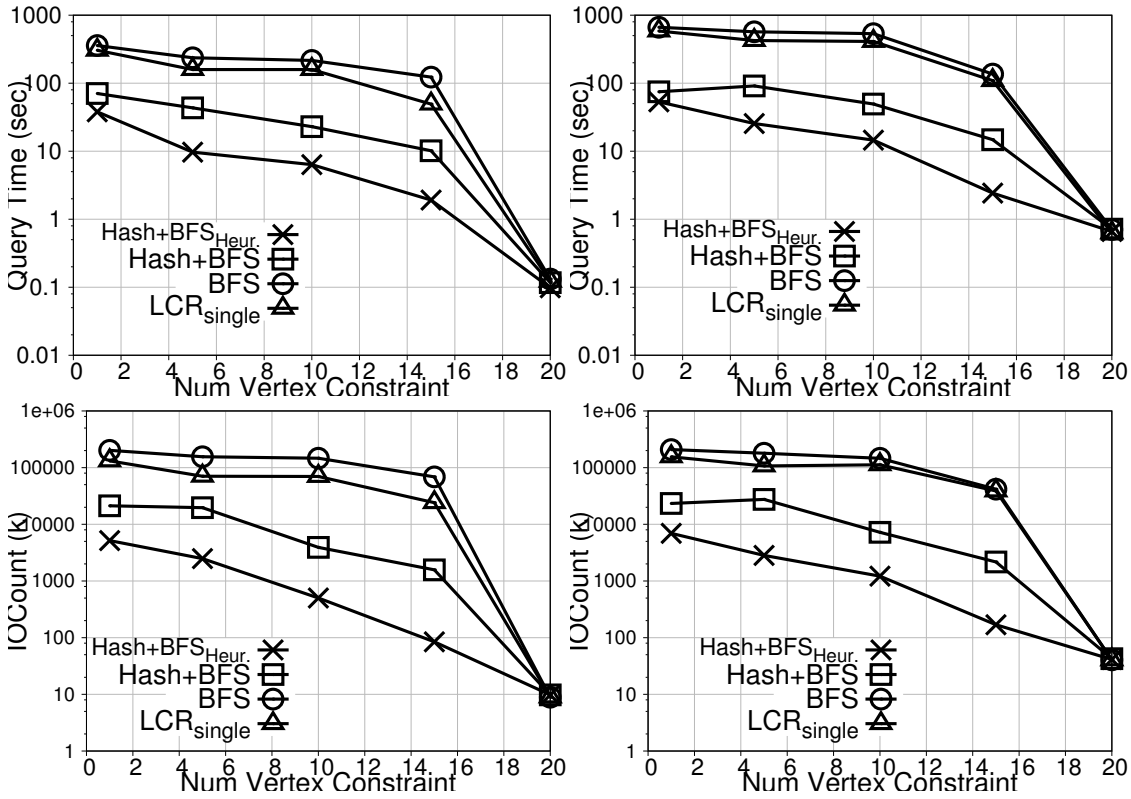
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.14: **[fb-bfs1]-**Vary # of E Const. with Double Dom.



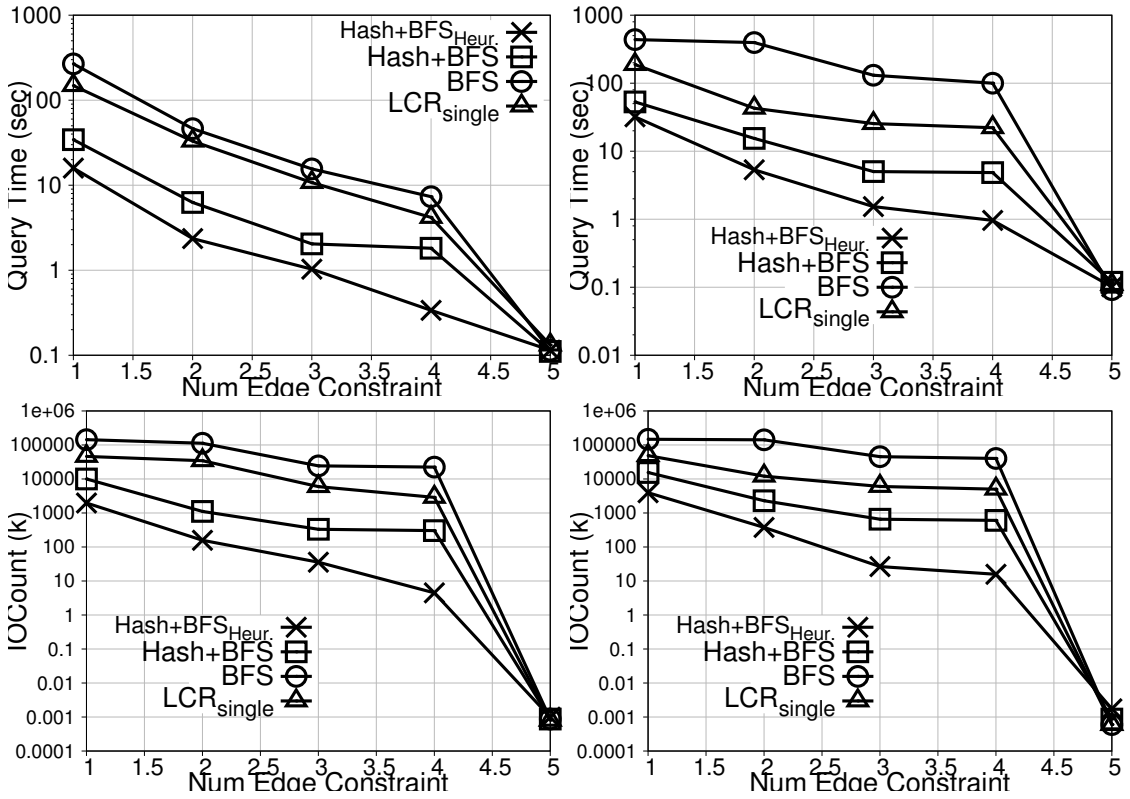
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.15: **[twitter-0.25]**-Vary # of V Const. with Org. Dom.



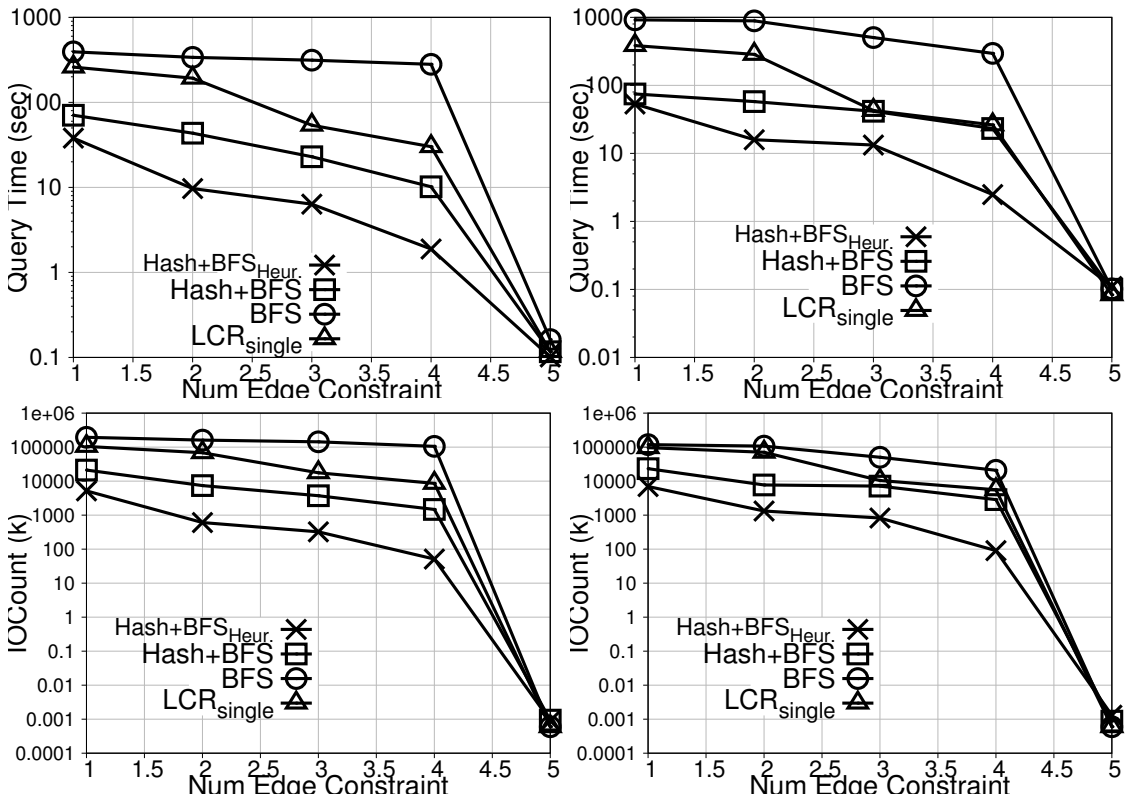
(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.16: **[twitter-0.25]-**Vary # of V Const. with Double Dom.



(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.17: **[twitter-0.25]-**Vary # of E Const. with Org. Dom.



(a)Num V Attr.=10 (b)Num V Attr.=30

Figure 4.18: [**twitter-0.25**]-Vary # of E Const. with Double Dom.

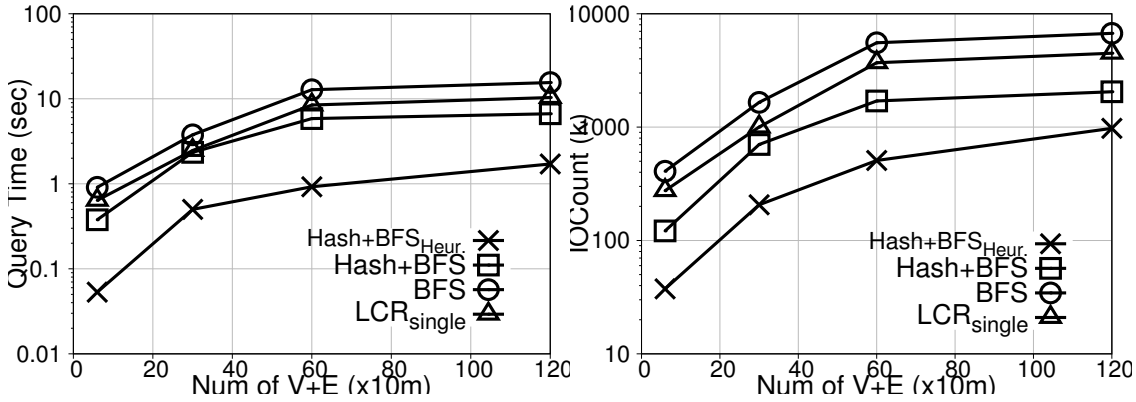


Figure 4.19: [**SmallWorld**]-Vary Graph Size

5.0 EFFECTIVE AND EFFICIENT HOW-TO-REACH ANSWER

In this Chapter, we study the problem of effective and efficient processing of How-to-Reach query on attributed graphs [23].

5.1 MOTIVATION APPLICATION

A reachability query with attribute constraints [19, 3, 4, 40] checks whether there is a constraint satisfied path from source to destination in an attributed graph. For this type of query, a user not only offer source and destination as input, he/she has to offer attribute constraints also. When a 'No' is returned by the query, the user can be surprised as the 'No' may be counterintuitive to the user. Hence, the user may want to know why: 'Am I setting the attribute constraint too restrictive?', 'Am I missing out some attributes?','How can I reach the destination from the source?'.

How-to-Reach Scenario

For example, a police officer may ask the social network database system whether there is a path from suspect S to a terrorist leader L such that all people on the path have the country attribute to be not USA and religion attribute to be A, B, or C. The social network database returns a 'No' to the police officer. The officer wonders why there is no such path between suspect A and terrorist leader L since the officer believes that suspect S and terrorist leader L must have some relationships or connections. Hence, the officer submits a How-to-Reach query to the social network database for a reason. The system discovers that there is such a path from S to L if the attribute constraints are set to any country to be not in USA and religion to be A, B, C, or D. After analyzing the answer of the How-to-Reach query, the officer

realizes that he/she forgot to consider religion D and indeed, adding religion D to attribute constraint can also imply the close relationship of suspect S and terrorist leader L. In this scenario, the How-to-Reach query cleared the question of the officer and helped him/her to fully utilize the social network for crime investigation. Without the How-to-Reach query, the intuition that there is a relationship between S and L may have already been given up by the officer.

How-to-Reach query is a kind of Why-Not query [47], which has been studied in many domains [48, 47, 49, 50, 51, 52, 53, 54, 55, 56] etc. However, to the best of our knowledge, we are the first to address Why-Not query on attributed graphs.

5.2 CHALLENGES AND TECHNICAL CONTRIBUTIONS

5.2.1 Challenges

The major challenges of computing answers for How-to-Reach queries lie in three folds.

Given a source and destination, there are an exponential number of attribute constraint combinations that would result in a 'Yes.' Hence, trial and error is not a practical approach. Finding the shortest path is a potential solution. However, the online computation of shortest path is costly. Furthermore, as the quality of a path is related to the initial attribute constraints, negative cost and cycle may happen during shortest path graph traversal, which makes efficient shortest path algorithm (e.g. Dijkstra's algorithm) to be inapplicable. A simple yet effective way for avoiding negative cost and cycle would be the first task for an efficient index construction and query processing.

Since attribute constraint is a query parameter, cost/penalty of a path can only be computed during query time. That makes all shortest path indexes that require knowledge of shortest paths between vertices cannot be directly adopted. Without knowing the attribute constraint, it is unclear which paths to precompute since any path can be the best or worst one.

Since attribute constraint is a query parameter, an index for how-to-reach query involves

storing of attributes on paths between vertices. Unlike directly storing topological distance, storing attributes on paths consumes a lot of storage space as attribute between vertices accumulates along the paths. That makes directly applying existing traversal based shortest path indexes (e.g. landmark technique) space inefficient.

5.2.2 Technical Contributions

Our first contribution is to introduce and define a new How-to-Reach query which can be implemented in attributed graph database systems for improving databases' usability. Since there may exist many attribute constraints that would allow source and destination to be connected, we further propose a metric that would reflect the importance of attribute values for computing answer quality.

Our second contribution is to make a traditional shortest path algorithm (i.e. Dijkstra's algorithm) which is commonly implemented in graph/attributed graph database systems [74, 75, 76] to be able to find optimal answers for How-to-Reach queries. We first studied the issues of directly applying Dijkstra's algorithm for How-to-Reach query. We discovered the problem of possible decreasing of cost and returning of sub-optimal answers. Then, we propose a simple trick that 1)does not require heavy modification of existing implementations in graph database systems and 2)is proven to be able to allow existing shortest path algorithms to return optimal answers.

Although after applying our trick, Dijkstra's algorithm can return optimal answer, we observed that 1)the computation time of Dijkstra's algorithm is unacceptance for inpatient users (i.e. $\approx$50sec) and 2)the hop distance of the optimal $s-t$ path tends to be meaningless (i.e. $\approx$1500 hops). Hence, our third contribution is to propose the station index, which is a time and space efficient non-traversal based index that returns high-quality approximate answers with reasonable hop distances.

Based on our experimental study(Figure 5.7, 5.8, and 5.9), we discover that by picking $\approx$5% of all vertices in graphs with degree $\approx$ 30 as stations, all sources and destinations can be almost adjacent to a station. Given this observation, the station index chooses to only focus on precomputing paths between stations using the Dijkstra's algorithm with our

trick. During query time, one of the paths between stations that are adjacent to source and destination are chosen as the answer. Furthermore, in order to make the answer more meaningful, we propose a simple hop reduction function that can harness the hop distance of answers while still maintaining high answer quality.

Finally, we evaluate our new techniques using both real and synthetic graphs with different parameter settings. We found that our new techniques can effectively reduce total computation time with a small trade-off in answer quality.

5.3 PROBLEM DEFINITION

Based on definitions in [19], we propose the definition of How-to-Reach query and How-to-Reach query answer quality(penalty) in this section. In this chapter, we assume that the higher the penalty is, the poorer the answer quality is.

Problem Statement [How-to-Reach Query $q_{hw}(C_0, s, t, G)$] Given attribute constraints C_0, s, and t, How-to-Reach Query return C' such that the answer of $q_r(C', s, t, G)$ is 'Yes', where the answer of $q_r(C_0, s, t, G)$ is 'No'.

5.3.1 Answer Quality

Definition 23. [Dropped Constraint Value $C_{drop}(C', C_0)$]

$$C_{drop}(C', C_0) = \{S_1^0/S_1^{'}, S_2^0/S_2^{'}, ..., S_{d_v}^0/S_{d_v}^{'}\}$$

$$where \;\; S_i^0 \in C_0 \;\; and \;\; S_i^{'} \in C'$$

Intuitively, $C_{drop}(C', C_0)$ contains the set of attribute values that are in C_0 but not in C'.

Definition 24. [Penalty of an Attribute Constraint Value P_a]

$$P_a(G) = \frac{|V_a|}{|V|}$$

$$where \;\; V_a = \{v | a \in D_i \wedge A_i(v) = a \wedge v \in V\}$$

$$and \;\; G = (V, E, A_v)$$

That is V_a is a set of vertices that have attribute value a in attribute i and $P_a(G)$ is the percentage of vertices in G that have attribute value a in attribute i. We argue that the penalty of attribute value should depend on how the attribute value affects the number of vertices that satisfy attribute constraints. For example, we knows that dropping a gender attribute value 'male' would make half of the vertices in a social network fail to satisfy attribute constraints while dropping a hobby 'baseball' may not have such a big effect.

Definition 25. [**Dropped Constraint Value Penalty** $\Delta C_{drop}(C', C_0, G)$]

$$\Delta C_{drop}(C', C_0, G) = \sum_{i=1}^{|C_{drop}(C', C_0)|} \left[\sum_{j=1}^{|S_i^0 / S_i^{'}|} P_j(G) \right]$$

Intuitively, the inner summation sums up the drop constraint penalty for 1 attribute dimension and the outer summation sums up drop constraint penalty of all attribute dimensions.

Definition 26. [**New Constraint Value** $C_{new}(C', C_0)$]

$$C_{new}(C', C_0) = \{S_1^{'}/S_1^0, S_2^{'}/S_2^0, ..., S_{d_v}^{'}/S_{d_v}^0\}$$

$$where \;\; S_i^0 \in C_0 \quad and \quad S_i^{'} \in C'$$

Intuitively, $C_{new}(C', C_0)$ contains the set of attribute values that are in C' but not in C_0.

Definition 27. [**New Constraint Value Penalty** $\Delta C_{new}(C', C_0, G)$]

$$\Delta C_{new}(C', C_0, G) = \sum_{i=1}^{|C_{new}(C', C_0)|} \left[\sum_{j=1}^{|S_i^{'} / S_i^0|} P_j(G) \right]$$

New constraint penalty is defined similar to dropped constraint penalty, except that $C_{new}(C', C_0)$ is used.

Definition 28. [**Penalty** $P(C', C_0, G)$]

$$P(C', C_0, G) = \Delta C_{drop}(C', C_0, G) + \Delta C_{new}(C', C_0, G)$$

Finally, penalty is defined to be the sum of dropped constraint penalty and new constraint penalty. This penalty value reflects how different is the answer of How-to-Reach query C' from original attribute constraints C_0.

5.4 FINDING OPTIMAL ANSWER

In this section, we will show an example of directly adopting shortest path algorithm for finding a path from s to t. Then, we will introduce how to transform the graph so that Dijkstra's algorithm can be adopted for finding the optimal answer. A trick is also mentioned for easy implementation in any existing attributed graph database systems.

5.4.1 Problem Of Applying Shortest Path Algorithm Directly

Suppose we directly adopt a shortest path algorithm (e.g. Dijkstra's algorithm) by propagating the attributes and defining the distance to be the penalty (Definition 28). Algorithm 7 is a typical implementation of Dijkstra's algorithm, except that constraint paths are maintained during graph traversal (line 12), and the distance is defined as the penalty (line 13). Unfortunately, this approach would have two major issues.

Definition 29. [**Constraint Path** $cp(v_i, v_j)$] *is the union of vertex attribute values on a path from* v_i *to* v_j.

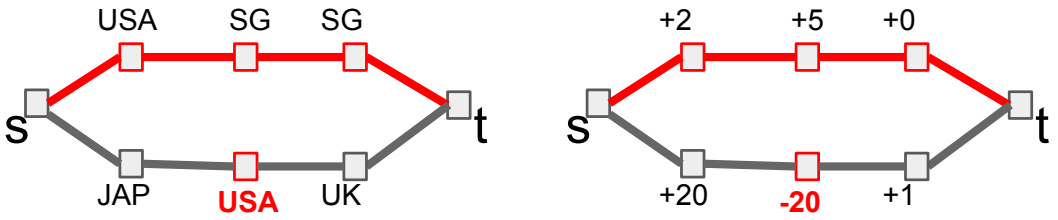
$$cp(v_i, v_j) = \bigcup_{k=2}^{|p|-1} v_k, where\ p = \{v_i, ..., v_j\}$$

Algorithm 7 How-to-Reach Algorithm

```
1: procedure Dijkstra(C_0, s, t, G)
2:     PrioirtyQueue q                          ▷ priority based on penalty P_cur
3:     q.put((s, {}, 0))                        ▷ (vertex,const. path,penalty)
4:     while q.empty() = false do
5:         (cur, cp_cur, P_cur) ← q.pop()
6:         if cur = t then
7:             return cp_cur
8:         end if
9:         if visited[cur] = true then
10:            continue
11:        end if
12:        visited[cur] ← true
13:        for all v ∈ G[cur].adjList do
14:            cp_v ← cp_cur ∪ G[v].attr
15:            P_v ← P(cp_v, C_0)
16:            q.put(v, cp_v, P_v)
17:        end for
18:    end while
19: end procedure
```

Suppose there is only 1 attribute - country and $C_0 = \{USA, JAP\}$. In Figure 5.1 and 5.2, vertex attributes are shown figure (a) and penalty of visiting a vertex is shown in figure (b).

5.4.1.1 Possible Decreasing Penalty Figure 5.1 shows an example of a possible decrease in the penalty during graph traversal. By using Dijkstra's algorithm, in Figure 5.1(b), the shortest path is $s \to USA \to SG \to SG \to t$ with $Penalty = 7$. However, the real shortest path is $s \to JAP \to USA \to UK \to t$ with $Penalty = 1$ since going through this path would visit the USA vertex which reduces the penalty by 20 i.e. $\Delta C_{drop}(\{JAP\}, C_0, G) = 20 \to \Delta C_{drop}(\{USA, JAP\}, C_0, G) = 0$. Unfortunately, Dijkstra's algorithm assumes that negative penalty does not exist. As a result, it cannot find the shortest path. Even though



(a)Original Attr. Graph (b)Ans:$\{USA, SG\}$, Penalty=7

Figure 5.1: Possible Decreasing Penalty

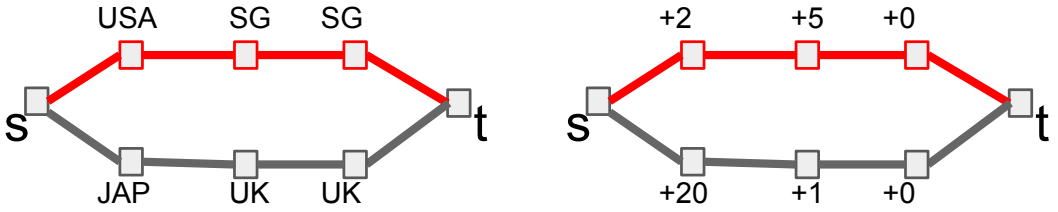
Bellman-Ford algorithm can be used for a graph with negative penalties, it has $O(|V||E|)$ time complexity, which makes it to be prohibitive for large graphs.

5.4.1.2 Sub-optimal Answer Quality Figure 5.2 shows an example of using shortest path algorithm (with penalty as distance) for finding a path from s to t. In this example, the negative penalty does not exist. Since $\Delta C_{drop}(\{USA\}, C_0, G) = 2$ is a lot smaller than $\Delta C_{drop}(\{JAP\}, C_0, G) = 20$, the shortest path algorithm would choose the upper path. Notice that the penalty of $\{USA, SG\}$ is not 12 as the penalty of $\Delta C_{new}(\{SG\}, C_0, G)$ should not be counted twice. Although $\{USA, SG\}$ is found using the shortest path algorithm, the answer found is not optimal in term of penalty. The optimal answer should be $\{USA, JAP, UK\}$ with penalty=1 as $\Delta C_{drop}(\{USA, JAP, UK\}, C_0, G)+$
$\Delta C_{new}(\{USA, JAP, UK\}, C_0, G)$=0+1.

Definition 30. [**Optimal Answer** C_{opt}] *is an attribute constraint such that* $P(C_{opt}, C_0, G)$ *is minimum among all possible constraints* C *where the answer of* $q_r(C, s, t, G)$ *is 'Yes'.*

$$\{\forall C \in CS, P(C_{opt}, C_0, G) \leq P(C, C_0, G)\} \wedge \{C_{opt} \in CS\}$$

$$where \;\; CS = \{C | q_r(C, s, t, G) =' Yes'\}$$



(a)Original Attr. Graph (b)Ans:$\{USA, SG\}$, Penalty=7

Figure 5.2: Sub-optimal Answer Quality

5.4.2 Attributed Graph Transformation

Given the sub-optimality and decreasing penalty issues mentioned in the previous section, in this section, We propose to transform the attributed graph G to the constraint union attributed graph G'.

Definition 31. [**Constraint Union Attributed Graph** G'] *is an attributed graph with attributes equal to the union of attributes in* G *and* C_0.

$$G' = (V, E, A_v \cup C_0)$$

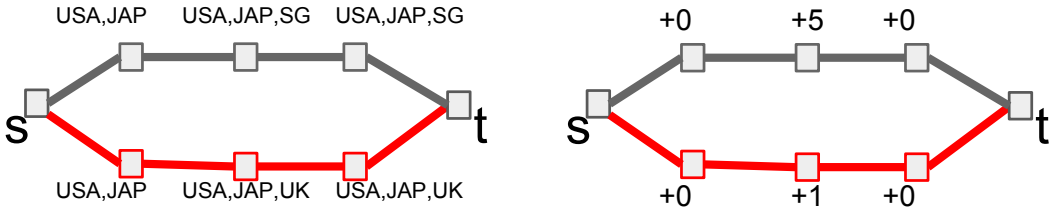
For example, all country vertex attributes in Figure 5.4(a) are union with $C_0 = \{USA, JAP\}$ and become the vertex attributes in Figure 5.3(a).

Lemma 3 states that given G', when penalty (Definition 28) is used as distance, non-decreasing of penalty can be achieved during graph traversal.

Lemma 3. [**Non-decreasing of Penalty**] *In* $Dijkstra(s, t, C_0, G')$ *(Algorithm 7),* P_v*(line 13) must be always greater than or equal to* P_{cur}*(line 5).*

Proof. We have to consider 2 cases for this proof.

- Case 1 $cur = s$: Since cur=s, we know that $P_{cur} = 0$ and $cp_{cur} = \{\}$. $\Delta C_{drop}(cp_v, C_0, G') = 0$ as cp_v are superset of C_0. $\Delta C_{new}(\{\}, C_0, G') \leq \Delta C_{new}(cp_v, C_0, G')$ as $cp_v = \{\} \cup v.attr$. Therefore, $P_{cur} = P(cp_{cur}, C_0, G') \leq P(cp_v, C_0, G') = P_v$.



(a)Const. Union Attr. Graph (b)Ans:$\{USA, JAP, UK\}$, Penalty=1

Figure 5.3: Optimal Answer Quality

- Case 2 $cur \neq s$: $\Delta C_{drop}(cp_{cur}, C_0, G') = \Delta C_{drop}(cp_v, C_0, G') = 0$ as cp_{cur} and cp_v are superset of C_0. $\Delta C_{new}(cp_{cur}, C_0, G') \leq \Delta C_{new}(cp_v, C_0, G')$ as $cp_v = cp_{cur} \cup v.attr$. Therefore, $P_{cur} = P(cp_{cur}, C_0, G') \leq P(cp_v, C_0, G') = P_v$.

□

5.4.2.1 Easy Implementation

The construction of constraint union attributed graph can actually be avoided. The trick is to make the initial constraint path to be C_0, instead of empty. Obviously, this can be easily implemented in Dijkstra's algorithm of any existing attributed graph database systems.

5.4.3 Proof of Optimal Answer Quality

In this section, we will prove that even though the Dijkstra's algorithm is performed on G', the answer is still optimal with respect to G.

Lemma 4. [Dropped Constraint Penalty is Zero] *Answer C' returned by $Dijkstra(s, t, C_0, G')$ (Algorithm 7) has $\Delta C_{drop}(C', C_0, G) = 0$.*

Proof. As C' is always a superset of C_0, $\Delta C_{drop}(C', C_0, G') = \Delta C_{drop}(C', C_0, G) = 0$. □

Lemma 5. [**New Constraint Penalty is Minimum**] *Answer C' returned by $Dijkstra(s, t, C_0, G')$ (Algorithm 7) has the minimum $\Delta C_{new}(C', C_0, G)$ among all constraints C that have $q_r(C, s, t, G) = Yes$.*

Proof. $\Delta C_{drop}(cp_{cur}, C_0, G')$ is always zero as $G' = (V, E, A_v \cup C_0)$. Hence, at anytime during the graph traversal, $P(cp_{cur}, C_0) = \Delta C_{new}(cp_{cur}, C_0)$-(♣). Lemma 3 states that penalty is non-decreasing in $Dijkstra(s, t, C_0, G')$. Since penalty is non-decreasing, $Dijkstra(s, t, C_0, G')$ can find C' with minimum $P(C', C_0, G')$-(♠). Based on (♣) and (♠), we know that $Dijkstra(s, t, C_0, G')$ can find C' such $\Delta C_{new}(C', C_0, G')$ is minimum among all constraints C that have $q_r(C, s, t, G') = Yes$-(★).

Now, we switch from G' to G. We use proof by contradiction. **Assumption:** Suppose there exists C'' such that $\Delta C_{new}(C'', C_0, G) < \Delta C_{new}(C', C_0, G)$ and $q_r(C'', s, t, G) = Yes$.

Step 1: Let $G_{new} = (V, E, A_v/C_0)$. Since $\Delta C_{new}(C'', C_0, G)$ and $\Delta C_{new}(C', C_0, G)$ are only contributed by attribute values in A_v/C_0, $\Delta C_{new}(C'', C_0, G_{new}) = \Delta C_{new}(C'', C_0, G) < \Delta C_{new}(C', C_0, G) = \Delta C_{new}(C', C_0, G_{new})$ -(♡).

Step 2: Let $G'_{new} = (V, E, (A_v \cup C_0)/C_0)$. Since A_v/C_0 equals to $A_v \cup C_0/C_0$, $G_{new} = G'_{new}$-(♢).

Step 3: By (♡) and (♢), we know that $\Delta C_{new}(C'', C_0, G'_{new}) < \Delta C_{new}(C', C_0, G'_{new})$-(□)

Step 4: Since $\Delta C_{new}(C'', C_0, G')$ and $\Delta C_{new}(C', C_0, G')$ are only contributed by attribute values in $(A_v \cup C_0)/C_0$, $\Delta C_{new}(C'', C_0, G') = \Delta C_{new}(C'', C_0, G'_{new})$ and $\Delta C_{new}(C', C_0, G'_{new}) = \Delta C_{new}(C', C_0, G')$ -(⊙).

Conclusion: Based on (□) and (⊙), we get $\Delta C_{new}(C'', C_0, G') < \Delta C_{new}(C', C_0, G')$. This result contradicts with (★) (there shoud not exist such C''.). Therfore, we know that $\Delta C_{new}(C', C_0, G)$ is also minimum among all constraints C that have $q_r(C, s, t, G) = Yes$. □

THEOREM 8. [**Answer Quality is Optimal**] *Answer C' returned by $Dijkstra(s, t, C_0, G')$ (Algorithm 7) has optimal quality i.e. $C' = C_{opt}$.*

Proof. As $\Delta C_{drop}(C', C_0, G)$ is 0 (by Lemma 4) and $\Delta C_{new}(C', C_0, G)$ is minimum (by Lemma 5), $P(C', C_0, G)$ must equal to $P(C_{opt}, C_0, G)$. □

5.5 STATION INDEX FOR SUBOPTIMAL ANSWER

We observe that the optimal constraint path found using the method in the previous section would probably be the union of attributes on a very long $s-t$ paths. For example, in Figure 5.5(e) and (f), we can see that the average length of $s-t$ path found is around 1000-1400 hops. Although the answer quality of such a long path is high, it is meaningless to users. Furthermore, since the propagation of constraint path involves frequent copy of constraint paths and computation of penalties (Algorithm 7 line 12 and line 13), the execution time can be very slow (Figure 5.5 (a) and (b)). Hence, in this section, we propose indexing techniques (Section 5.5.1) that can reduce execution time and a hop reduction function (Section 5.5.2.1) for reducing the length of $s-t$ path while maintaining high answer quality.

5.5.1 Station Index Construction

The idea of station index is to first pick a set of vertices as stations. Then, multiple constraint paths are pre-computed between stations for serving sources and destinations that are very close to the stations. Figure 5.4 illustrates the idea. In Figure 5.4, 9 vertices are picked as stations $(st_1, .., st_9)$. Constraint paths are precomputed between every pair of stations. Given source s and destination t, constraint path is found by concatenating paths from s to st_1, st_1 to st_9, and st_9 to t. If s is very close to st_1 and t is very close to st_9, then basically, no graph traversal is needed. Furthermore, if s and t are adjacent to st_1 and st_9 and the quality of the precomputed constraint paths between st_1 and st_9 are very high, the sub-optimal constraint path $s \rightarrow st_1 \rightarrow st_9 \rightarrow t$ would have very high quality. Based on this intuition, we design our station index.

5.5.1.1 Index Construction Algorithm Algorithm 8 is the pseudo code for station index construction. Firstly, a set of vertices is picked as stations (line 2). The algorithm performs a graph traversals for every station (lines 5-19). For each station, it assumes C_0 is empty and finds $|AttrSub|$ of optimal constraint paths from that station to all other stations (lines 6-19), where $AttrSub$ is a set that contains subsets of attributes that a graph traversal

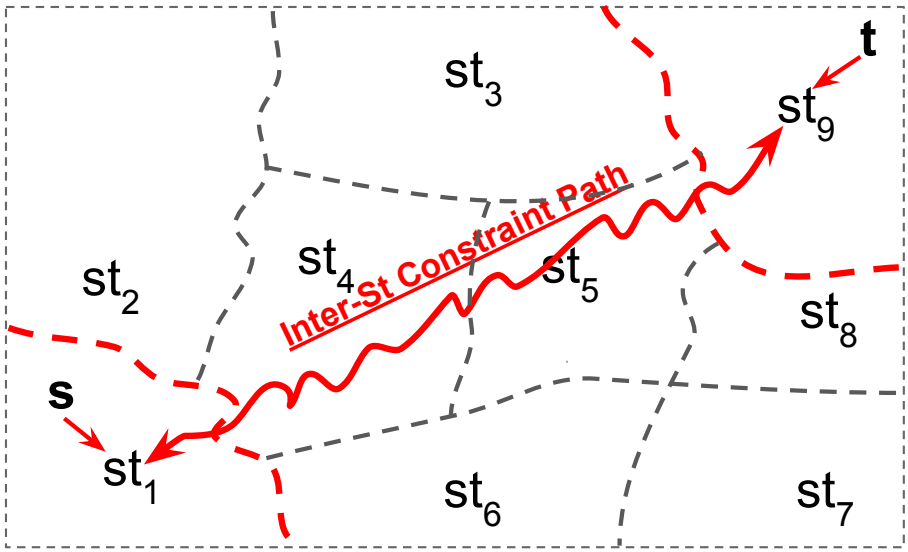


Figure 5.4: Station Index: s,t are served by nearby stations.

focuses on. For example, if $AttrSub$ contains $\{\{A_1A_2\},\{A_5\}\}$, the penalty computation of the first graph traversal (line 18) only considers attribute 1 and 2; the penalty computation of the second graph traversal (line 18) only considers attribute 5.

Algorithm 8 Station Index Construction

1: **procedure** $StationIndexConstruction(m, G)$
2: $Stations \leftarrow ChooseStations(m, G)$
3: $StationBP[m][m][|AttrSub|]$ ▷ for storing multiple BP between stations
4: /*Compute mulitple inter-station*/
5: **for all** $st \in Stations$ **do**
6: **for all** $A'() \in AttrSub$ **do** ▷ $AttrSub$ is a subset of attr. dimension
7: $PrioirtyQueue\ q$ ▷ priority based on penalty P_{cur}
8: $q.put((st, A'(\{\}), 0))$ ▷ (vertex,BP,penalty)
9: **while** $q.empty() = false$ **do**
10: $(cur, cp_{cur}, P_{cur}) \leftarrow q.pop()$
11: **if** $visited[cur] = true$ **then**
12: $continue$
13: **end if**
14: $visited[cur] \leftarrow true$
15: **if** $isStation(cur)$ **then**
16: $StationBP[st][cur].add(cur.cp)$
17: **end if**
18: **for all** $v \in G[cur].adjList$ **do**
19: $cp_v \leftarrow cp_{cur} \cup A'(G[v].attr)$
20: $P_v \leftarrow P(A'(cp_v), A'(\{\}))$
21: $q.put(v, cp_v, P_v)$
22: **end for**
23: **end while**
24: **end for**
25: **end for**
26: **return** $(StationBP, Stations)$
27: **end procedure**

5.5.1.2 Station Picking Strategy

We propose to pick vertices in local cover as stations.

Definition 32. [Local Cover] *LC is a subset of vertices in $|V|$ such that for any vertex v in $|V|$, there exists a vertex v' in LC, v is adjacent to v'.*

$$LC \subset V s.t. \forall v \in V, \exists v' \in LC \ s.t. \ v \ \ adj. \ \ v'$$

Notice that local cover is different from vertex cover and edge cover. Based on our experimental study (Figure 5.7, 5.8, and 5.9), we discover that for our social network datasets, the size of local cover would only be around 5% of the total number of vertices. We believe that for other social networks with higher degree, the size of local cover would be even smaller than 5% of the total number of vertices. Since there are many possible LC is a graph, we propose a simple heuristic for computing LC. The intuition of the heuristic is to greedily find stations that can serve more adjacent vertices so as to reduce the size of LC. We will leave more complicated strategy for future study.

Algorithm 9 is the pseudo code. Firstly, degrees of all vertices are computed (line 3). Then, all vertices are inserted into a priority queue with the degree as the priority (lines 4-5). Then, the vertex v with the highest number of adjacent vertices that is not covered is pop from q (line 7). v is chosen as a station (line 8). As adjacent vertices v' of v is covered by v, all v' are deleted from q (lines 9-10) and priorirty of adjacent vertices v'' of v' in q need to be updated (lines 12-13).

Algorithm 9 Station Picking Strategy

```
1: procedure ChooseStations(m, G)
2:     Stations[m]                                  ▷ to store m stations
3:     Deg_v ← ComputeVDeg(G)
4:     PriorityQueue q
5:     Insert(q, G.V, Deg_v)                        ▷ priority on deg.
6:     for all i = 1..m do
7:         v ← q.pop()
8:         Stations[i] ← v
9:         for all v' ∈ Adj(v) do
10:            Delete(q, v')                        ▷ v' is covered
11:            for all v'' ∈ Adj(v') do
12:                UpdatePriority(q, v'')           ▷ update pri. of v"
13:            end for
14:        end for
15:    end for
16:    return Stations
17: end procedure
```

Assume that attribute values and query workload are uniformly distributed, we believe that the performance of station index would decrease with the number of station. Also, the performance of station index is sensitive to the station picking strategy. That is because for other strategies (e.g., random station), they cannot maintain the property of local cover. The breaking of the local cover property makes some sources and destinations in the query workload being far away from a station. As a result, the answer quality of those queries is very poor.

5.5.1.3 *AttrSub* **Picking Strategy** We propose to pick subsets of attributes into *AttrSub* based on historical query workload. We suggest picking top-k most frequent subsets of attributes in a given workload, where k depends on available storage space.

5.5.1.4 Space Consumption Analysis The space consumption of station index can be derived by:

$$\frac{(1+m)\times m}{2} \times s_1 \times |AttrSub|$$

where m is the number of stations, s_1 is the average size of a inter-station constraint path. For comparison, the space consumption of landmark index can be derived by:

$$|V| \times m \times s_2 \times |AttrSub|$$

where m is the number of landmarks and s_2 is the average bound path size to a landmark. In general, s_2 is close to s_1.

For example, we assume that $s_1 = s_2 = 30B$,$|AttrSub| = 30$ and $|V| = 10m$. The space consumption of 2 indexing techniques are shown in Table 5.1. We can see that given the same amount of storage capacity, a lot more stations can be computed than landmarks.

Table 5.1: Space Consumption

Technique	**Num of LM/Station (m)**	**Storage Space**
Station Index	5000	≈10.5 GB
Landmark Index	5000	≈31.2 TB
Station Index	50000	≈1.0 TB
Landmark Index	50000	≈40.8 TB

5.5.1.5 Time Complexity Analysis Time complexity of algorithm 8 can be derived by:

$$m \times |AttrSub| \times Dijkstra_{full}$$

where $Dijkstra_{full}$ is the complexity of a shortest path algorithm on the whole graph. For station index construction, parallel computation in different machines can be used for speeding up station index construction after all stations are chosen as every full graph traversal is independent.

5.5.2 Efficient How-to-Reach Query Processing

5.5.2.1 Hop Reduction Since we observe that the hop distance of optimal answer is a large number, we want to harness the hop distance so as to make the result more meaningful. This can be done by redefining the priority function as below:

$$priority = max(hop - ExpectedHop, 0) + penalty \times hop$$

Here, hop is the hop distance from s and expected hop is a user input parameter. The intuition of this function is to a)offer credits to paths that are less than $ExpectedHop$ from s and b)penalize more on paths that are far away from s. This function is also used for station index construction. The expected hop can be decided based on historical query workload.

5.5.2.2 Query Algorithm During query time, s and t looks for the closest stations st_s and st_t. Since there must be a station adjacent to s and t if all vertices in LC are chosen as stations. The closest stations of s and t are actually very close to them. After finding st_v and st_t, precomputed constraint paths between them are retrieved from primary/secondary storage. If $AttrSub$ contains attributes in C_0, only the constraint path that is computed based on the attribute in C_0 is retrieved; otherwise, all constraint paths computed using attributes that overlap with attributes in C_0 are retrieved, and the best one is adopted. The final constraint path is the union of constraint path s to st_s, best constraint path between st_s and st_t, and constraint path st_t to t.

5.6 EXPERIMENTAL RESULT

5.6.1 Experiment Setup and Dataset

All experiments were performed using C++ implementations under a Linux machine with an Intel 4GHz CPU (4-core), 64 GB of memory, and 1 TB solid state drive with 512k block size.

Table 5.2 is a summary of our graph dataset. In order to control the number of attributes and attribute domain sizes, we generate attributes (Table 5.3) based on vertex attributes in Facebook graph-API [70].

Table 5.2: Dataset Information

Real Graph	**Num of Vertex**	**Num of Edge**
fb-bfs1 [72]	*1.18m*	*29.78m*
soc-pokec [71]	*1.63m*	*30.6m*
Synthetic Graph	**Num of Vertex**	**Num of Edge**
Small-World [71]	*1m*	*50m*
	2m	*100m*
	5m	*250m*
	10m	*500m*
Erdos-Renyi [71]	*1m*	*50m*
	2m	*100m*
	5m	*250m*
	10m	*500m*

Table 5.3: Attributes

Vertex Attribute	**Domain Size,Distribution (μ,σ)**
AgeGroup	*10, gau(5,2.5)*
Education	*5, gau(3,1.25)*
Gender	*2, uni.*
HomeCountry	*100, gau(50.25)*
Interested in	*3, uni.*
Languages	*50, gau(25,12.5)*
Relationship status	*2, uni.*
Religion	*20, gau(10,5)*
Work	*50, uni.*
Political	*10, gau(5,2.5)*

Table 5.4: Parameter Setting

Parameter	**Value**
Num V Attr	*10*
Num V Constraint	*10, 20,30,40,50*
Number of Stations	*5000*
Number of BP per Station	*20*
Synthetic Graph Size($\|V\|+\|E\|$)	*1m+50m, 2m+100m,5m+250m*

We compare 5 different approach:

1. SP finds constraint paths using Dijkstra's algorithm.
2. Pen_{opt} finds optimal quality constraint paths using approach in Section 5.4.
3. $PenHop$ finds constraint paths using approach in Section 5.4 with hop reduction.
4. $StationHop$ finds constraint paths using station index and hop reduction.
5. LM finds constraint paths using landmark index. LM precomputes constraint paths from every landmark to every vertex. During query time, constraint paths can be computed

by $cp(s, LM_i) \cup cp(LM_i, t)$. Thes best quality constraint path among all $cp(s, LM_i) \cup cp(LM_i, t), i = 1...|LM|$ is returned as the answer.

Expected pros and cons of each approach is shown in Table 5.5.

Table 5.5: Expected Pros and Cons of Baselines

Solution	**Ans. Quality**	**Query Time**	**Num. of Hop**
SP	Low	Fast	Shortest
Pen_{opt}	Optimal	Slow	Very long
PenHop	High	Slow	Reasonable
StationHop	**High**	**Fast**	**Reasonable**
LM	Low	Fast	Reasonable

5.6.1.1 Experiment 1 - Comparison of Different Approaches In this experiment, we try to vary the number of vertex attribute constraints and graph size (Section 5.6.2 and Section 5.6.3) so as to observe the change of overall running time, answer quality, and hop distance of our techniques and baselines. Parameter settings are summarized in Table 5.4. We do not vary the number of stations since it is obvious that the more station we have, the better the answer quality. We also do not vary the number of attributes since we assume that there would not be very specific queries that would involve more than ten attributes in attribute constraints. The number of vertex attribute constraint in this experiment are defined as:

$$\sum_{i=1}^{d_v}(|D_i^v| - |S_i^v|) \tag{5.1}$$

(D_i^v and S_i^v, are the same as D_i, S_i in Definition 11,12).

The station index is constructed based on historical query workload. The historical query workload contains 1,000 queries. The involved attributes and the expected hop of these queries are generated based on two Gaussian distributions - $gau(3, 2)$ and 10. The attribute constraint values are uniformly generated. The top 20 most frequent attribute combinations are used for constructing station index (i.e. $|AttrSub| = 20$ for station index). The landmark index uses 50 landmarks for constructing index (roughly the same index

storage size as station index). The 20 most frequent attribute combinations are used for computing constraint paths for every landmark to every vertex (i.e. $|AttrSub| = 20$ for landmark index). For both station index and landmark index, every constraint path is quality optimal with respect to the combinations of attributes used for computing the paths when C_0 of the attribute combination is empty. The same experiment design is used for real and synthetic graph experiments. All results are averages of 1000 queries. The involved attributes and the expected hop of these queries are generated based on two Gaussian distributions - $gau(3, 2)$ and $gau(10, 3)$. The attribute constraint values are again uniformly generated.

5.6.1.2 Experiment 2 - Hop Distance to Nearest Station/Landmark Analysis

After evaluating the performances of different approaches, we study average hop distance from sources and destinations to their nearest stations (Section 5.6.4) of $fb - bfs1$ and $soc - pokec$. We generate 5000 queries and vary the number of stations and landmarks to see how the hop distances change. We also estimate the storage space of station index and landmark index when $|AttrSub| = 20$ and $s_1 = s_2 = 30$ based on formula in Section 5.5.1.4.

5.6.2 Performance on Real Graphs

In this section, we will present the experimental results of 2 real graphs - fb-$bfs1$ and $soc - pokec$ with analysis. Results for varying number of vertex/edge constraints using default setting are shown in Figure 5.5. The general trend of average running time does not vary too much when increasing the number of attribute constraint. That is because in general, execution time of SP,$StationPen$, and LM does not really depend on number of attribute constraints; for Pen_{opt} and $PenHop$, the increase in number of attribute constraint affects priority of vertices in priority queue, but it is averaged out by large number of queries.

5.6.2.1 Query Time Figure 5.5(a) and (b) show the query time of different approaches. We can see that the execution times of Pen_{opt} and $PenHop$ are the largest. That is mainly due to 3 reasons: 1)a large part of the graph is traversed so as to find a high-quality path, 2)the propagation of constraint path require copy of constraint path from parent vertex to

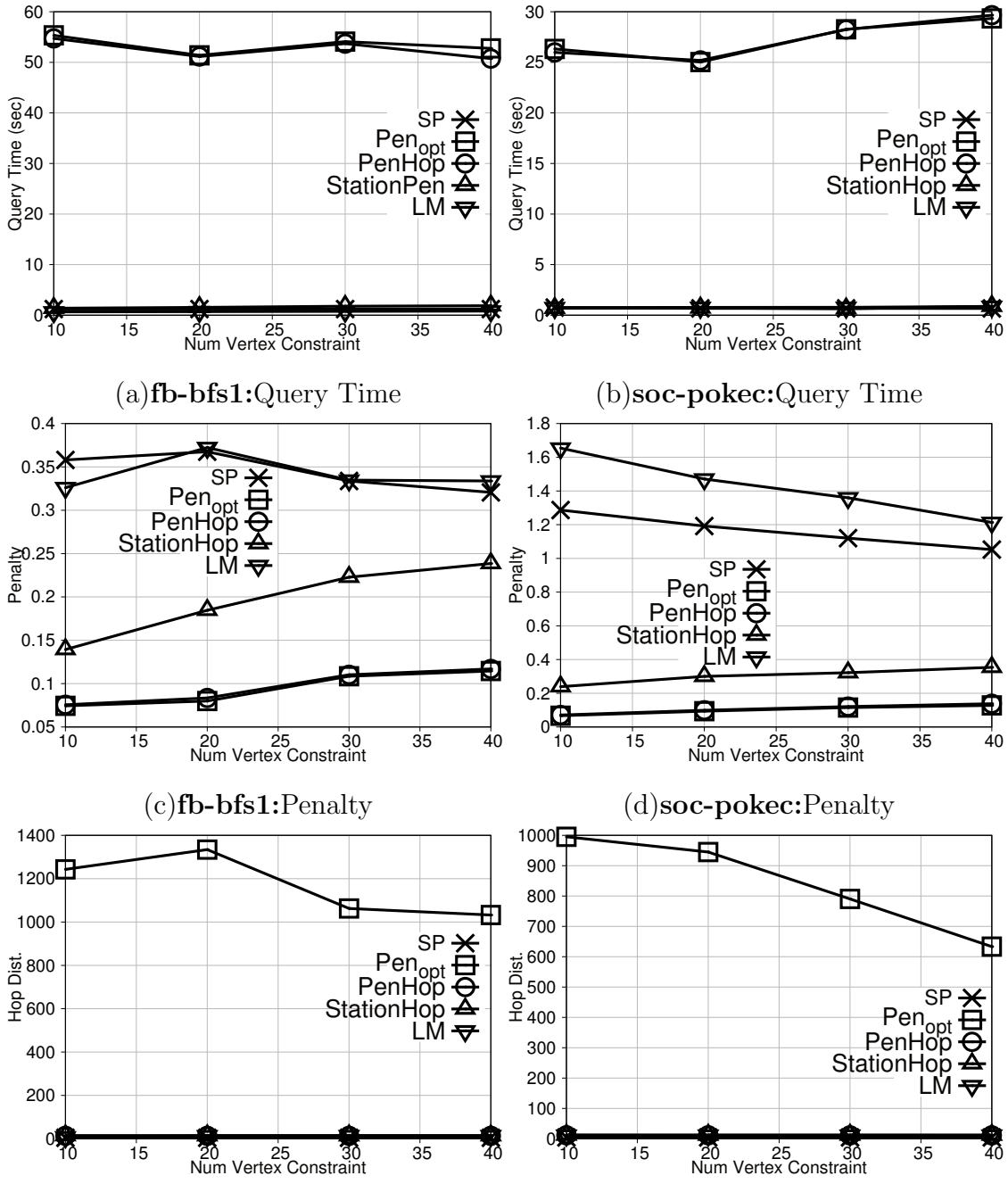
child vertex, and 3)the computation of constraint path penalty is done for every traversed vertex. As the (hop distance) shortest path is very short on a social network, SP can be completed relatively fast. $StationPen$ and LM are basically non-traversal based approach, so their execution time is relatively small.

5.6.2.2 Penalty Figure 5.5(c) and (d) show the penalty of different approaches. The penalty of SP and LM are the two largest since SP only consider hop distance and LM replies on landmarks that are far away from source and destination. Obviously, Pen_{opt} should have the smallest penalty since it is proven to have optimal answer quality. Surprisingly, $PenHop$ also has penalty close to Pen_{opt}. This shows that actually, there exist multiple paths with similar penalties from a source to a destination and different hop distances. $StationPen$ also return answers with penalties close to optimal even though we just used 5000 stations due to space limit. The success of $StationPen$ approach is contributed by those high-quality precomputed paths between stations and the fact that those stations are indeed very close to sources and destinations.

5.6.2.3 Hop Distance Figure 5.5(e) and (f) show the hop distance of paths found by different approaches. Obviously, SP would have the shortest hop distance. The hop distance of Pen_{opt} is amazingly large since Pen_{opt} would look for optimal paths without caring about hop distance. The hop distance of $PenHop$ is controlled by the hop reduction function. Since the precomputed paths of $StationHop$ are computed based on $PenHop$, hop distance of $StationHop$ would be similar to $PenHop$.

5.6.3 Performance on Synthetic Graphs

In this experiment, we used setting in Table 5.4 with number of vertex constraint 20 and attributes in Table 6.2 for $Small - World$ graphs with different sizes. In Figure 5.6(a), we can see that in general, query time of Pen_{opt} and $PenHop$ increase with graph size. However, query time of $StationPen$ and LM do not really increase since they are non-traversal based approaches. The penalties of all approaches increase with data size. However, the increase is

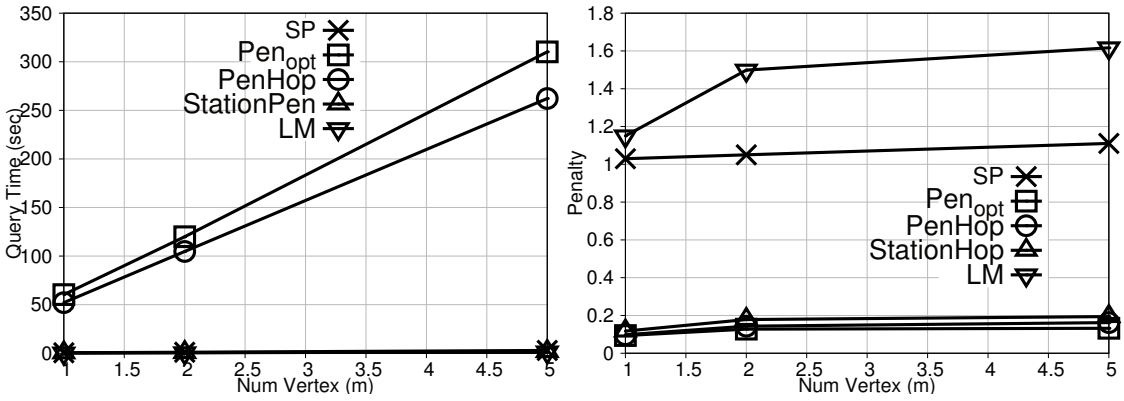


(a)**fb-bfs1:**Query Time (b)**soc-pokec:**Query Time

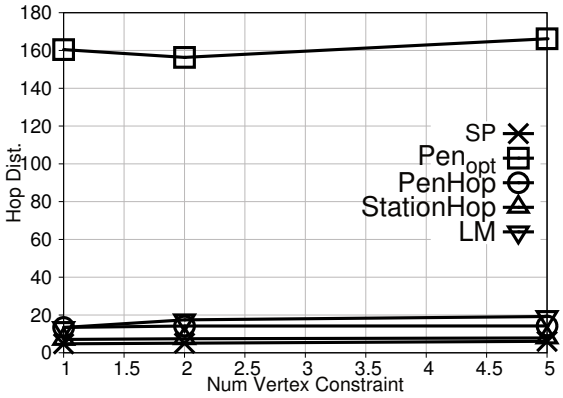
(c)**fb-bfs1:**Penalty (d)**soc-pokec:**Penalty

(e)**fb-bfs1:**Num of Hops (f)**soc-pokec:**Num of Hops

Figure 5.5: Vary Number of V Constraint



(a)**SmallWorld:**Query Time (b)**SmallWorld:**Penalty



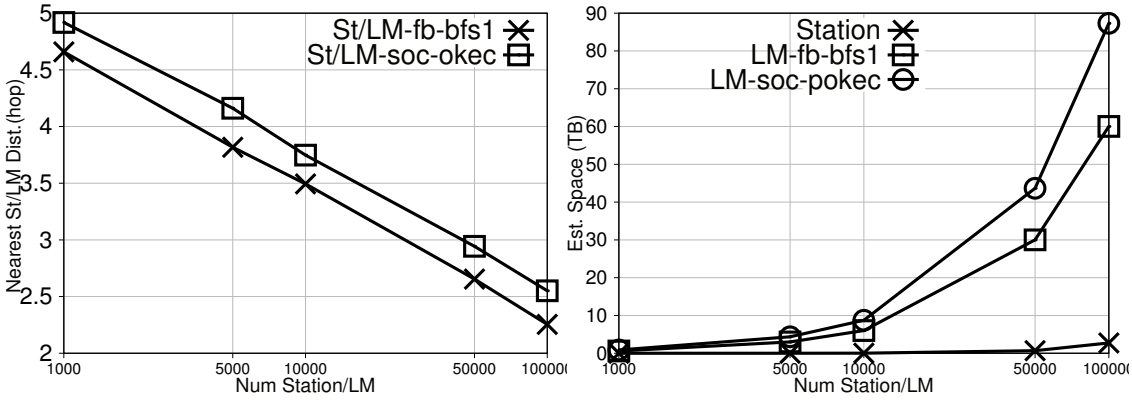
(c)**SmallWorld:**Num of Hops

Figure 5.6: Vary Graph Size

not significant as $SmallWorld$ is a social network which contains a lot of short paths between vertices with similar attributes. That matches with the slight increase of hop distances in Figure 5.6(c).

5.6.4 Hop Distance From Nearest Station/Landmark

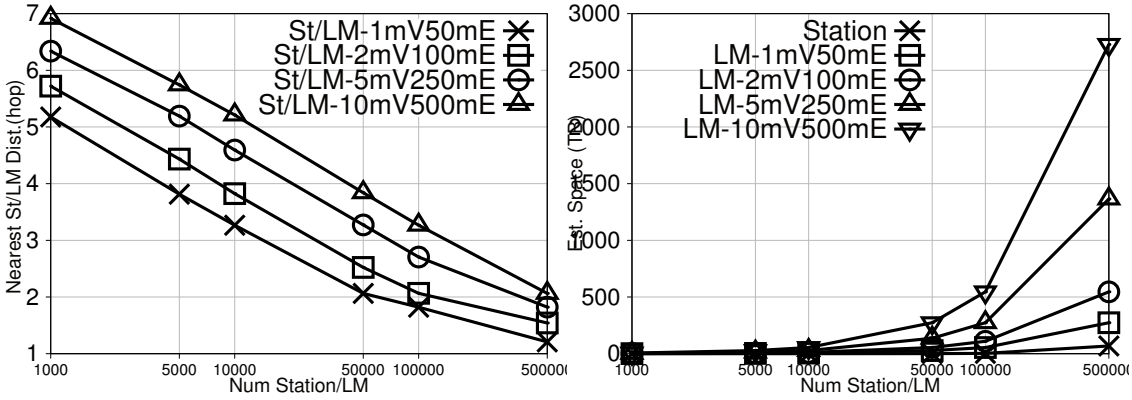
Figure 5.7, 5.8, and 5.9(a) show the average hop distance from source to the nearest station/landmark plus the hop distance from destination to the nearest station/landmark. We can see that the average hop distance can easily go to around 2 when the number of stations or landmarks is around 5% of the total number of vertices. For example, when there are 50k



(a) Dist. from Nearest St/LM (b) Estimated Storage Space

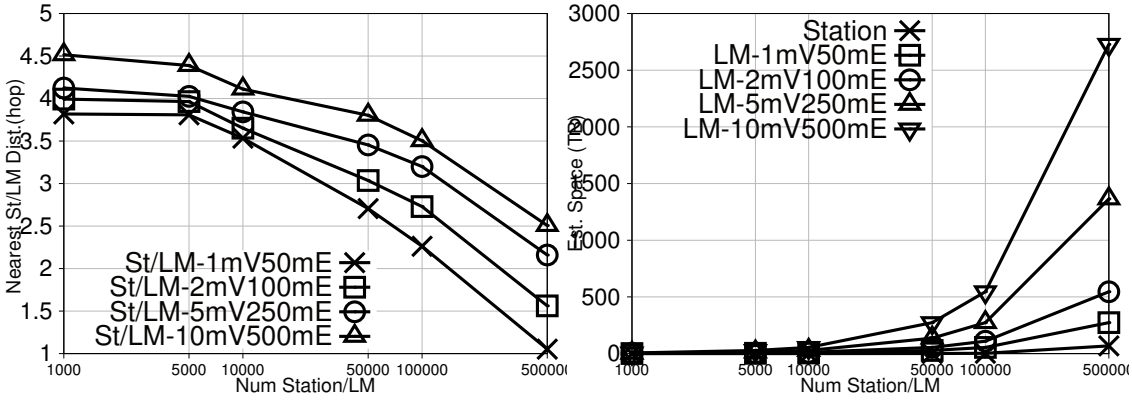
Figure 5.7: [**Real Graphs**] Hop Dist to Nearest St/LM and Est. Space

stations/landmarks, the average distance is very close to 2 for the SmallWorld graph with 1 million vertices ($St/LM - 1mV50mE$ in Figure5.8(a)). That means almost every vertex is adjacent to a station/landmark. If we assume the same number of stations and landmarks are used, *StationHop* and *LM* would result in very similar performance. However, from Figure 5.7, 5.8, and 5.9(b), we can see that the space consumption of landmark is actually very large and would grow very fast.



(a) Dist. from Nearest St/LM (b) Estimated Storage Space

Figure 5.8: [**SmallWorld**] Hop Dist to Nearest St/LM and Est. Space



(a) Dist. from Nearest St/LM (b) Estimated Storage Space

Figure 5.9: [**Erdos-Renyi**] Hop Dist to Nearest St/LM and Est. Space

6.0 VISUALIZABLE PATH SUMMARY COMPUTATION

In this Chapter, we study the problem of effective computation of visualizable path summary [24].

6.1 MOTIVATION APPLICATION

Attributed graph is widely used for modeling a variety of information networks [2, 1], such as the web, sensor networks, biological networks, economic graphs, and social networks. When a new dataset is modeled as an attributed graph or users are not familiar with the data, users may not know what can be retrieved from the attributed graph. Sometimes, users may have some intuition about the query, but how to exactly formulate queries (e.g. what attribute constraints to use) is still unclear to users.

In this chapter, we propose the idea of visualizable attributed path summary. In general, an attributed path summary is a grouping of vertices such that vertices in each group contain a path from source to destination and the entropy of attributed values within a group is low and biased toward the intuition (i.e. attribute values) given by users. In addition, we argue that a visualizable attributed path summary can be easily visualized and understood by users.

Social Network: For example, a police officer has a social network, but he/she is not familiar with the attribute values and graph structure of the social network. The agent wants to investigate the relationship between Duncan and a terrorist leader using the social network as the officer believes that social network would contain a lot of useful insight for investigation. The agent just got an intuition that people between Duncan and the terrorist

leader may live in the country C_1 and believe in religion R_1,R_2. The attributed path summary query computes a summary of paths from Duncan to the terrorist leader that are close to the offered attribute values (i.e. C_1,R_1 or R_2). The path summary offers insight for the agent to formulate different path queries for investigation.

Metabolic Network: In metabolic networks, each vertex is a compound, and an edge between two compounds indicates that one compound can be transformed into another one through a certain chemical reaction. Vertex attributes can be features of the compound; edge attributes can be details of a chemical reaction between two compounds. A reachability query on metabolic networks discovers whether compound A can be transformed to compound B under given path attribute constraints. A biologist wants to study how to transform compound A to compound B. The biologist only knows that cost-to-trigger-reaction has to be around \$10. The attributed path summary computes a summary of paths from compound A to compound B that are close to the offered attribute value (e.g. cost-to-trigger-reaction$\approx$ \$10). The path summary offer insight for the biologist to formulate path queries for the study.

6.2 CHALLENGES AND TECHNICAL CONTRIBUTIONS

6.2.1 Challenges

Nowadays, a big graph with a few million vertices is common, and that results in an exponential number of paths between any two vertices. A large number of possible paths between 2 vertices makes computing path summary a challenging task.

Among a huge number of possible paths between 2 vertices, which type of path the user prefers is unknown since even the user is not familiar with the graph, and he/she may not know what he/she can get from the graph. Therefore, our task is to compute a path summary for the user.

Computing an effective summary for a user is non-trivial as no user would prefer to read a lot of text to understand the summary. Hence, an effective path summary is a summary

that can be easily visualized by users. Visualizing a large portion of the graph is not feasible as that would overwhelm the user. On the contrary, if the summary is too concise, the user may not get the information he/she wants.

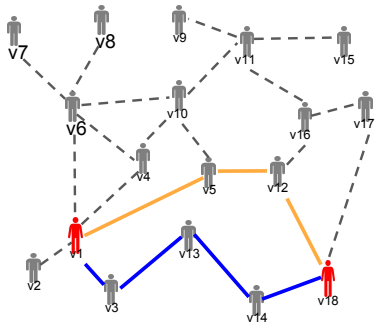
6.2.2 Technical Contributions

Our first contribution is to introduce and define the attribute path summary query on attributed graph problem. We define attributed path summary to be groups of vertices that contain users' intuition as well as satisfy some path properties. The users' intuition is expressed as hints for computing the path summary. Users can offer whatever attribute values that they consider as the hint. These summaries offer insight to users about the attribute values and connection between the given source and destination vertices.

Our second contribution is to propose an efficient and effective approach for finding attributed path summary. Our proposed approach consist of three phrases. The first phrase efficiently finds all key vertices that have attribute values belonging to the hint offered by the user. Including key vertices ensures the summary would represent paths with attribute values that are close to the intuition of users. Then, based on those key vertices, a novel stitching algorithm is proposed to connect the source, the destination, and key vertices together to form a relatively small key vertex graph. The stitching algorithm finds paths with a small variation in attribute values between key vertices so that users can easily understand the attribute distribution between key vertices. After that, high-quality candidate paths between the source and the destination are found on that small key vertex graph efficiently. Finally, candidate paths are inflated to vertex groups by greedily including adjacent vertices. Including adjacent vertices would offer more attribute values choices for users to formulate their queries.

6.3 PROBLEM DEFINITION

Definition 33. [**Attributed Graph**] *An attributed graph [1] G, is an undirected graph*



vID	Country	Job
$v1$	USA	IT
$v3$	USA	IT
$v5$	USA	IT
$v12$	SG	IT
$v13$	JAP	FIN
$v14$	UK	EDU
$v18$	USA	FIN
..	...	...

Figure 6.1: Path Summary (P_1-blue, P_2-orange)

denoted as $G = (V, E, A_v)$*, where* V *is a set of vertices,* $E \subseteq V \times V$ *is a set of edges, and* $A_v = \{A(v)\}$ *is a set of* d_v *vertex-specific attributes, i.e.* $\forall v \in V$*, there is a multidimensional tuple* $A(v)$ *denoted as* $A(v) = (A_1(v), A_2(v), ..., A_{d_v}(v))$*.*

Definition 34. [**Attribute Hint** H] *is a set of distinct attribute values.*

$$H = \{H_1, H_2, .., H_{d_v}\}$$

Definition 35. [**Contain Function** $\phi(P_i, H)$]

$$\phi(P_i, H) = \sum_{j=1}^{|P_i|} contain(v_j, H)$$

$$[right =]contain(v_j, H) = \begin{cases} 1, & \forall k = 1..d_v \ \ if \ \exists A_k(v_j) \in H_k \\ 0, & otherwise \end{cases}$$

For example in Figure 6.1, given that $H = \{\{USA, SG, JAP\}, \emptyset\}$,$P_1 = \{v1, v3, v13, v14, v18\}$, and $P_2 = \{v1, v5, v12, v18\}$, $\phi(P_1, H) = 1+1+1+0+1 = 4$ and $\phi(P_2, H) = 1+1+1+1 = 4$.

Definition 36. [**Attributed Path Summary** $PSum(G, s, t, l, H)$] *For an attributed graph* G*,* $PSum(G, s, t, l, H)$ *is a set of vertices* $\{P_1, P_2, ..., P_k\}$ *such that:*

1. $\forall$ $v \in P_i$ *are connected,*

2. $\exists\ v, v' \in P_i$, v *is adjacent to* s *and* v' *is adjacent to* t,

3. $\forall P_i, P_j \in G, i \neq j, P_i \cap P_j = \emptyset$,

4. $\forall P_i, \phi(P_i, H) \geq l$,

5. $\forall v \in P_i, dist(s, v) + dist(v, t) \leq l$, *and*

6. $\nexists P \in G \wedge P \notin PSum(G, s, t, l, H)$ *satisfing condition 1 to 5.*

where $\phi(P_i, H)$ *is the quality of the summary and* $dist(v, v')$ *is the shortest distance from* v *to* v'.

Continuing the above example, in Figure 6.1, given that $l = 4$, there are two paths P_1, P_2 in the attributed path summary. They are $P_1 = \{v1, v3, v13, v14, v18\}$ and $P_2 = \{v1, v5, v12, v18\}$. For example, all vertices in P_1 are connected, $\exists\ v_3$ and v_14 adjacent to s and t, $P_1 \bigcap P_2 = \emptyset$, $\phi(P_1, H) = \phi(P_2, H) = 4 = l$, and for all $v_i \in P_1, P_2$, and $dist(s, v_i) + dist(v_i, t) \leq 4$.

Problem Statement [Attributed Graph Path Summary Query q_p] Given an attributed graph $G = (V, E, A_v)$, source s, destination t, attribute hint H, and lower bound of number of vertices in every P_i that contains at least one attribute value in attribute hint l, q_p return an attributed path summary $PSum(G, s, t, l, H)$.

6.3.1 Quality of Path Summary

The quality of path summary is defined as the entropy in [25] and is reformulated in definition 37.

Definition 37. [Path Summary Quality]

$$entropy(P_j) = \sum_{i=1}^{d_v} entropy(a_i, P_j)$$

where

$$entropy(a_i, V_j) = -\sum_{n=1}^{n_i} p_{ijn} log_2 p_{ijn}$$

and k *is the number of* P_i *in* $PSum$, p_{ijn} *is the percentage of vertices in* P_j *which have value* a_n *on attribute* a_i.

For example, $entropy(Country, P_1) = -(\frac{3}{5} log_2 \frac{3}{5} + \frac{1}{5} log_2 \frac{1}{5} + \frac{1}{5} log_2 \frac{1}{5})$.

6.4 PATH SUMMARY VISUALIZATION ALGORITHM

In this section, we introduce our path stitching approach for computing attributed path summary effectively based on attribute hint.

6.4.1 Algorithm Design

Our heuristic approach has the following steps and design principles.

1. Firstly, we want to find all vertices - key vertices, that are related to the given attribute hint. The search of key vertices ensures that all vertices that match any attribute value in the hint and fulfill the distance requirement (Condition 5, Definition 36) are used for computing a path summary.
2. Given those key vertices, we perform a concurrency graph traversal that systematically stitches key vertices, the source, and the destination. Using stitched key vertices, we find candidate paths that go from the source to the destination via key vertices based on the entropy of attribute values on the path. Key vertices are vertices that users care and want to see in the visualized path summary. The stitching algorithm can effectively connect key vertices so that attribute values on the path between key vertices are consistent. That offers a clear view for users to understand the attribute distribution between key vertices.
3. Finally, given the candidate paths, we perform a candidate path inflation for computing the path summary. Candidate path inflation includes vertices close to vertices in candidate paths into the candidate paths. That allows users to understand attribute distributions around key vertices. When users are considering what attribute constraint to use for their attribute graph queries, they can consider attribute values on candidate paths as well as attribute values close to the candidate path as an alternative.

   Algorithm details are presented in below sections with conceptual examples.

6.4.2 Finding Key Vertices

We first introduce the concept of key vertex (Definition 38). Then, we present two steps that exploit existing approach to efficiently find all key vertices.

Definition 38. [**Key Vertex** v^k] *is a vertex that has at least one attribute value belonging to an attribute value in the attribute hint* H*.*

$$\forall i = 1..d_v \forall j = 1..d_v \;\; \exists A_i(v^k) \in H_j$$

$$where \;\; H_j \in H$$

The first step is to retrieve all key vertices. Traditional indexes that support range query (e.g. B+ tree) can be used to index each attribute. Given H, for each non-empty $S_j \in H$, we query the corresponding index for a set of vertices that have attribute values in S_j. Then, we do a union of all these vertices and get the key vertex set V_k. After that, all vertices v that does not satisfy $dist(s, v) + dist(v, t) \leq l$ are filtered out.

For example, in Figure 6.1, if the hint contains only $Country = USA$, all vertices with attribute value $Country = USA$ (e.g. v_1, v_3, v_5, v_{18}) are retrieved from the precomputed index (e.g. B+ tree).

6.4.3 Finding Candidate Path

After all key vertices are found, the second step is to find candidate paths that satisfy constraints in Definition 36.

6.4.3.1 Stitching Algorithm Since key vertices are essential (so as to satisfy condition 4 in Definition 36) for paths in path summary, we do not want to find candidate paths that do not contain any key vertex. The idea of the stitching algorithm is to connect $s - t$ and key vertices so as to form candidate paths. During the graph traversal, entropy and hop distance values are taken into account.

Algorithm 10 is the pseudo code of the stitching algorithm. The stitching algorithm first puts s, t, and all key vertices (lines 7-13) into the priority queue. Each node in the priority

queue contains the current vertex, a key vertex, parent of current vertex, the distance from a key vertex to the current vertex, and the entropy of path from a key vertex to the current vertex, where the entropy value is used to determine the priority.

Then, the graph is traversed starting from each of the key vertices. When the algorithm reaches a visited node $cur.v$ (line 16), if key vertex of current node's parent ($key1$) is not equal to the key vertex of curent node ($key2$) (line 19), the path from $key1$ to $key2$ is recovered and put into $PathMap$ (line 21), edges between vertice $key1$ and $key2$ are added in to $KeyVertexG$ (lines 22-23), and the algorithm continue (line 22); when the algorithm reaches a non-visited node (line 25), parent and key vertex of current node is saved (line2 26-27) and current node becomes visited (line 28).

After that, adjacent neighbors of $cur.v$ that satisfy the upper bound distance constraint (line 30) are put into the priority queue (line 33), where the entropy of the path from the key vertex to $cur.v$ as well as the distance are taken into account. Finally, the graph traversal continues until the priority queue becomes empty.

A conceptual example will be presented below.

6.4.3.2 Candidate Path Search After executing the stitching algorithm, $KeyVertexG$ and $Path$ are found. The path search algorithm is used to find paths from s to t via key vertices in the key vertex graph - $KeyVertexG$. The actual path are recovered using $path$ after $s - t$ in $KeyVertexG$ are found. Both entropy and distance from s are taken into account in the priority queue. We set priority as $entropy + current\ \ distance/l$ if $current\ \ distance < l$; otherwise, we set priority as $entropy + current\ \ distance$, in order to pennalize path in $KeyVertexG$ that are longer than l.

Algorithm 11 is the pseudo code of the path search algorithm. Algorithm 11 finds shortest path from s to t on $KeyVertexG$ based on entropy value (line 5). After a path p is found, p is removed from $KeyVertexG$ (line 7). The algorithm continues until no more path can be found.

6.4.3.3 Conceptual Example Figure 6.2 illustrates the concept of stitching algorithm and candidate path search. Suppose v_5 and v_{15} are key vertices retrieved from the index and

Algorithm 10 Stitching Algorithm

```
1: procedure STITCHING(G, V_k, s, t, l)
2:     Array < bool >  visited
3:     Array < int >  parents
4:     Array < Array < int >>  KeyVertexG
5:     Array < int >  keys
6:     priority_queue < node >  qu                                   ▷ lower entropy first
7:     node src(s, s, s, 0, 0)                                       ▷ (vert., keyVert, parent, dist, entropy)
8:     qu.push(src)
9:     node dest(t, t, t, 0, 0)
10:    qu.push(dest)
11:    for all v^k ∈ V_k do
12:        node n(v^k, v^k, v^k, 0, 0)
13:        qu.push(n)
14:    end for
15:    while !qu.empty() do
16:        cur ← q.pop()
17:        if visited[cur.v] == true then
18:            key1 ← keys[parents[cur.v]]
19:            key2 ← cur.keyVert
20:            if key1 == key2 then   ▷ it is just a cycle but not meeting of 2 traversals from diff KeyVertex
21:                continue
22:            end if
23:            PathMap ← ComputePathBetween(key1, key2)
24:            KeyVertexG[key1].push(key2)
25:            KeyVertexG[key2].push(key1)
26:            continue
27:        else visited[cur.v] == false
28:            parents[cur.v] ← cur.parent
29:            keys[cur.v] ← cur.keyVert
30:            visited[cur.v] ← true
31:        end if
32:        for all v ∈ G[cur.v].adj do int v = topology[cur.v][i];
33:            if dist_s[v] + dist_t[v] > l then
34:                continue
35:            end if
36:            en ← CompEntropy(cur.keyVert, cur.v) + (cur.dist + 1)/l
37:            node n(v, cur.keyVert, cur.v, cur.dist + 1, en)
38:            qu.push(n)
39:        end for
40:    end while
41:    return (KeyVertexG, Path)
42: end procedure
```

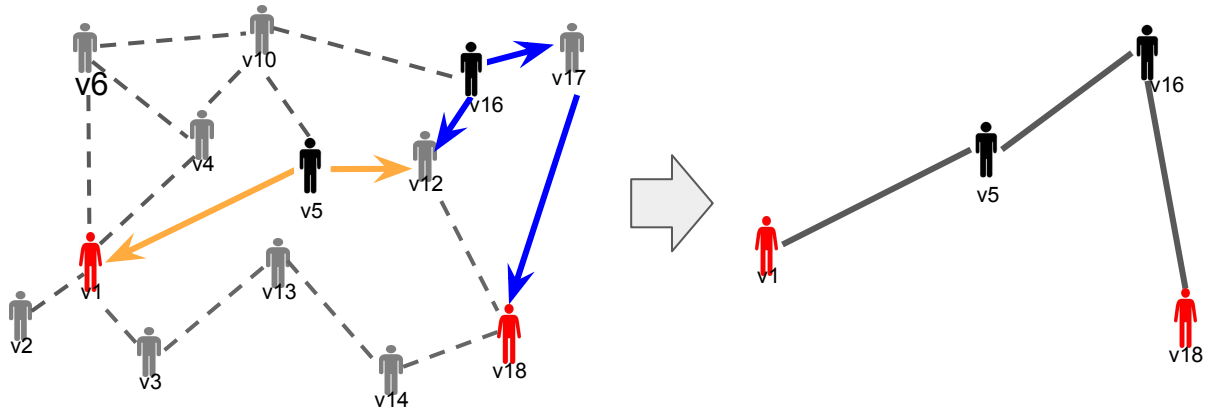


Figure 6.2: Stitching Algorithm (left) and Candidate Path (right)

v_1 and v_{18} are s and t respectively. v_5 expands to v_1, and edges from v_1 to v_5 and v_5 to v_1 are put into $KeyVertexG$. v_5 also expands to v_{12}. v_{15} expands to v_{17} and v_{12}. When v_{15} expands to v_{12}, v_{12} was occupied by v_5 already. Hence, we can put edges v_{15} to v_5 and v_5 to v_{15} into $KeyVertexG$. After that, v_{15} expands to v_{18} , and edges from v_{15} to v_{18} and v_{18} to v_{15} are put into $KeyVertexG$. Given the $KeyVertexG$, candidate paths from s to t are found based on entropy values, and those candidate paths will be used for path inflation in the next phrase.

6.4.4 Candidate Path Inflation

After all candidate paths, $CandPath$ are found, the candidate paths are used to form path summary. We developed the path inflation algorithm which greedily includes vertices into path vertex groups.

Algorithm 12 is the pseudo code of the path inflation algorithm. Firstly, all vertices in the $CandPath$ are put into a priority queue which uses entropy as the priority (lines 4-8). Then, the vertex cur in the candidate path with the lowest entropy are popped from the priority queue (line 10). If cur was not visited before, cur is included in the path group $PathSummary[cur.PathID]$ (line 14). After that, all adjacent vertices of cur that satisfy $dist_s[v] + dist_t[v] > l$ *or* $(cur.dist + 1) * 3 > dist_s[t]$ (line 19) are pushed into the priority

Algorithm 11 Path Search Algorithm

1: **procedure** PATHSEARCH($KeyVertexG, s, t, l, \alpha$)
2: $boolean\ \ PathFound \leftarrow true$
3: $Array < Path > \ \ CandPath$
4: **while** $PathFound == true$ **do**
5: $p \leftarrow FindShortestPath(KeyVertexG, s, t, l, \alpha)$
6: **if** $p! = \emptyset$ **then**
7: $RemovePath(p, KeyVertexG)$
8: $CandPath.push\_back(p)$
9: **else**
10: $PathFound \leftarrow false$
11: **end if**
12: **end while**
13: **return** $CandPath$
14: **end procedure**

queue with $entropy(cur.P \cup v)$ as cost. $(cur.dist+1)*3 > dist_s[t]$ is included so as to prevent vertices that are too far away from vertices in $CandPath$ are included in the path summary. The algorithm terminates when the priority queue becomes empty.

Figure 6.3 illustrates the concept of candidate path inflation. Given that $v_1 \rightarrow v_5 \rightarrow v_{12} \rightarrow v_{16} \rightarrow v_{17} \rightarrow v_{18}$ is the candidate path. The path inflation algorithm first puts all vertices (i.e. $v_5, v_{12}, V_{16}, v_{17}$) in the candidate path into the priority queue with entropy of the candidate path as priority. Firstly, vertices that are adjacent to $v_5, v_{12}, V_{16}, v_{17}$ are included into the candidate path. Then, other vertices that are adjacent to vertices (e.g. v_4, v_6) in the candidate path are gradually included in the candidate path until the distance constraint. Finally, we will get a subgraph shown in Figure 6.3 (right).

Algorithm 12 Path Inflation Algorithm

```
1: procedure PathInflation(G, CandPath, s, t, l)
2:     priority_queue < node > qu
3:     Array < bool > visited
4:     for all p ∈ CandPath do
5:         for all v ∈ p do
6:             entropy ← ComputeEntropy(p)
7:             node n(v, i, v, 0, l, entropy)
8:             qu.push(n)
9:         end for
10:    end for
11:    while !qu.empty() do
12:        cur ← qu.pop()
13:        if visited[cur.v] == true then
14:            continue
15:        end if
16:        visited[cur.v] ← true
17:        PathSummary[cur.pathID].push(cur.v)          ▷ assign v into that path group
18:        for all v ∈ G[cur.v].adj do
19:            if dist_s[v] + dist_t[v] > l or (cur.dist + 1) * 3 > dist_s[t] then
20:                continue
21:            end if
22:            en = ComputeEntropy(PathSummary[cur.pathID], v)
23:            node n(v, cur.pathID, cur.dist + 1, cur.l, cur.keyVertex, en)
24:            qu.push(n)
25:        end for
26:    end while
27:    return PathSummary                                ▷ return Path Summary
28: end procedure
```

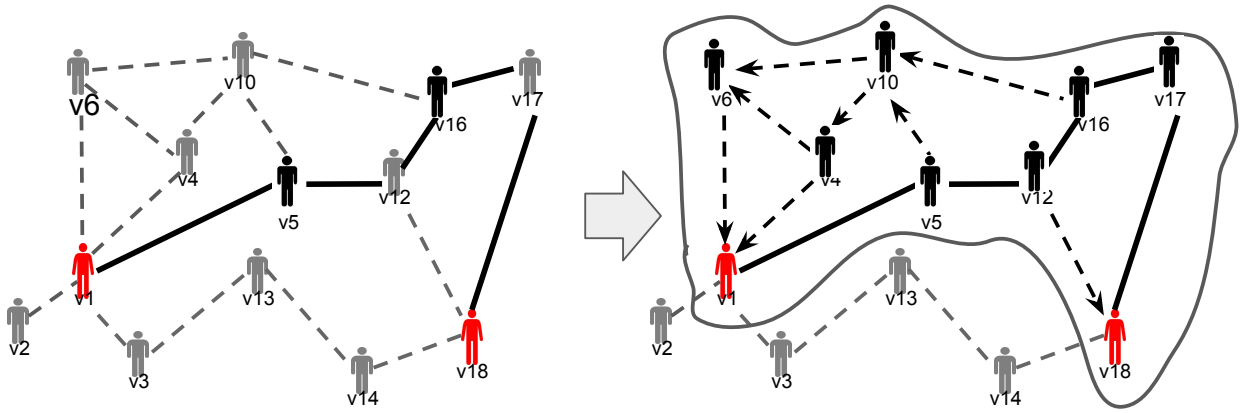


Figure 6.3: a Candidate Path (left) and a Vertex Group P_i (right)

6.5 CASE STUDY AND EVALUATION

All experiments were performed under 64-bit Linux Ubuntu 14.04 on a machine with an Intel 4GHz CPU (4-core), 16 gigabytes of memory, and 1 terabyte solid state drive with 512k block size. All our implementations are in C++ without parallelism.

We first introduce the graph dataset and attributes that we used for the experiments. Then, we present the result of our case studies. Finally, we look at the change of change of path summary quality (i.e. change of entropy) along with the change in the expected number of key vertex l and the number of hints H.

Table 6.1: Dataset and Parameter

Real Graph	**Num of Vertex**	**Num of Edge**
fb-bfs1 [72]	*1.18m*	*29.78m*
Parameter	**Default**	**Vary**
Exp. Num of Key Vert.	*6*	*3,6,9,12*
Num of Hint	*3*	*1,3,6,12*

6.5.1 Datasets

We used a real social network dataset $fb\text{-}bfs1$ [72], which has 1.63m vertices and 15.14m edges, for our experiments. To control the number of attributes and attribute domain sizes, we generate attributes (Table 6.2) based on vertex attributes in facebook graph-API [70].

Table 6.2: Attributes

Vertex Attribute	**Domain Size,Distribution (μ,σ)**
AgeGroup	*10, gau(5,2.5)*
Education	*5, gau(3,1.25)*
Gender	*2, uni.*
HomeCountry	*100, gau(50.25)*
Interested in	*3, uni.*
Languages	*50, gau(25,12.5)*
Relationship Status	*2, uni.*
Religion	*20, gau(10,5)*
Work	*50, uni.*
Political	*10, gau(5,2.5)*

6.5.2 Case Study

Figure 6.4 and 6.5 are four case studies using the $fb - bfs1$ [72] graph. Each of the figures contains a visualization of one of the paths in the path summary. At the top of each figure, we can see the key vertices (man icon) from source to destination. Below each key vertex is the attribute value of the key vertex that matches attribute value in the hint. Attribute value summaries of the path between every two key vertices are shown above the edges between every two key vertices. The pie chats below the path are the summaries of attribute value found by the inflation algorithm. This attribute value summary summarizes the attribute value near to the key vertices. The average execution time of computing a path summary in our case studies is around 200s

Case 1: For the first case study in Figure 6.4(a), we set the expected number of key vertex $l = 6$, the number of hint $H = 3$, and the hint contains attribute $Country = AUS$,

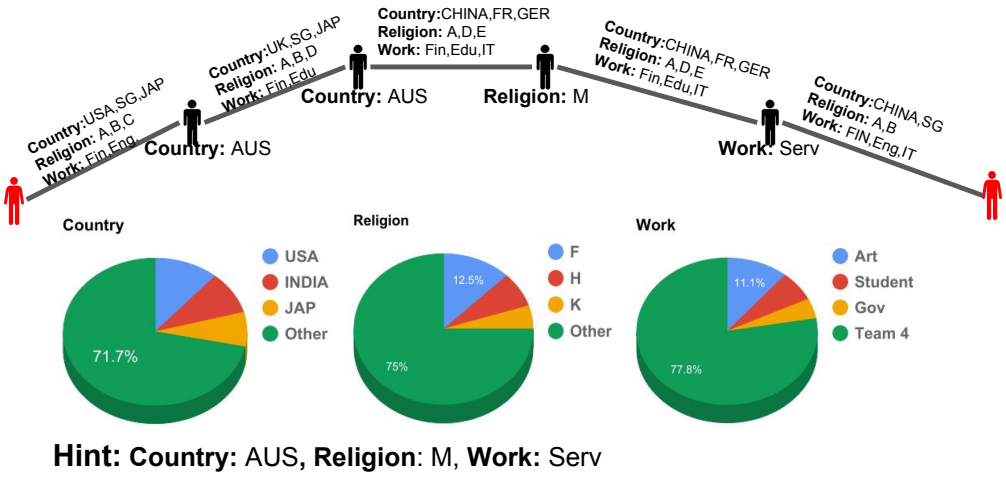
$Religion = M$, and $Work = Service$. We can see there are 6 key vertices (including source and destination), which match our expected number of key vertices. Furthermore, the summaries of attribute values on the path between every two key vertices are concise. That gives users a clear idea of attribute values between key vertices. From the pie charts, we can see that *other* (green) occupies a large portion of the pie. That tells users that attribute values close to the path are inconsistent and probably having large attribute value domain.

Case 2: For the first case study in Figure 6.4(b), we set the expected number of key vertex $l = 6$, the number of hint $H = 3$, and the hint contains attribute $Education = Primary$, $Interested\ \ In = Men$, and $Politic = B$. We can see there are 6 key vertices (including source and destination), which matches our expected number of key vertices. Furthermore, the summaries of attribute values on the path between every two key vertices are concise. That gives users a clear idea of attribute values between key vertices. From the pie charts, we can see that the top-2 attribute values (blue and red) in each attribute occupies a large portion of the pie. That tells users that attribute values close to the path are consistent and that helps users to efficiently construct their queries.

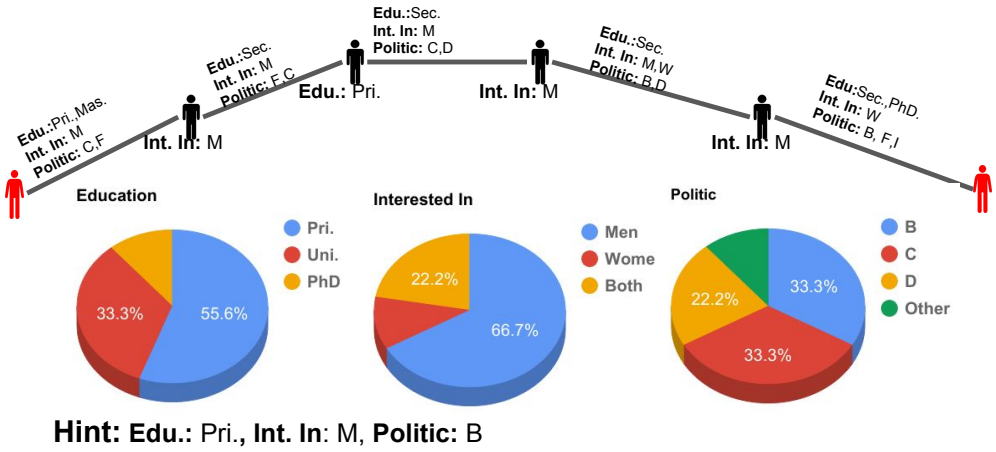
After we study path summary with 6 key vertices and 3 hints, we try to look at cases with less key vertices and hints.

Case 3: For the first case study in Figure 6.5(a), we set the expected number of key vertex $l = 3$, the number of hint $H = 1$, and the hint contains attribute $Age = 20 - 30$. We can see there are 3 key vertices (including source and destination), which matches our expected number of key vertices. Furthermore, the summaries of attribute values on the path between every two key vertices are also concise. From the pie charts, we can see that the top-1 attribute values (blue) in each attribute occupies a large portion of the pie. On the contrary, the "*other*" attribute value (green) only occupies a small portion. That tells users that attribute values close to the path are very consistent.

Case 4: For the first case study in Figure 6.5(b), we set the expected number of key vertex $l = 3$, the number of hint $H = 1$, and the hint contains attribute $Education = Master$. We found similar result as in Figure 6.5(a).



(a) Case 1



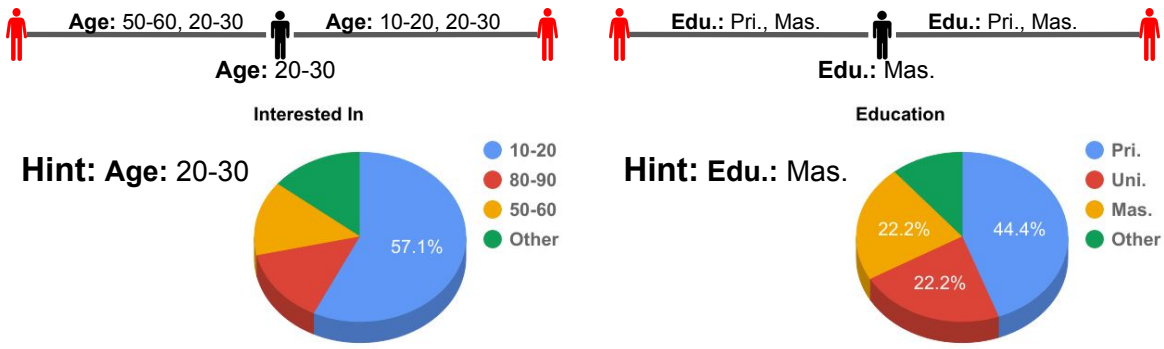
(b) Case 2

Figure 6.4: Path Summary (Expected Num of Key Vertex=6, Num of Hint=3)

6.5.3 Query Formulation Using Path Summary

In order to connect source and destination via vertices that satisfy attribute hint, we suggest users take into account major attribute values and alternative attribute values when they are formulating queries.

Major Attribute Values: Major attribute values are attribute values that appear on the



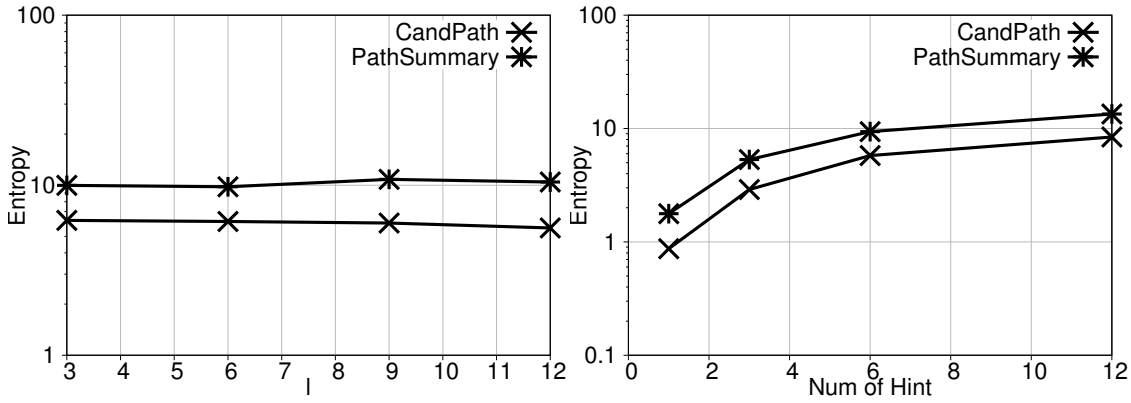
(a) Case 3 (b) Case 4

Figure 6.5: Path Summary (Expected Num of Key Vertex=3, Num of Hint=1)

path between key vertices. For example, in Figure 6.4(b), "**Edu.**:Sec.,PhD., **Int. In**: W, **Politic**: B,F,I" are major attribute values between destination and the last key vertex. By putting these attribute values into the query, key vertices can be connected. However, based on users preferences, they may not always want to include these major attribute values. Continue with the example in Figure 6.4(b), users may not want to include "**Edu.:**Sec.,PhD" into the query. If that is the case, users can consider the alternative attribute values.

Alternative Attribute Values: Alternative attribute values are attribute values displayed in the pie charts. They are the distribution of attribute values near to paths between key vertices. Continue with the example in Figure 6.4(b), if users do not prefer to have "**Edu.**:Sec.,Ph.D." in the query, they may consider to replace it by "**Edu.**: Uni". Based on the "Education" pie chart, there are 33.3% of vertices has attribute value "**Edu.**: Uni" near to the paths between key vertices. Therefore, conceptually, choosing "**Edu.**: Uni" is similar to rerouting the path between the destination and the last key vertex.

In Figure 6.4(a), a clear indication of attribute distribution can be seen. However, this does not always happen. When the attribute distribution is chaotic, that indicates that it is not "easy" to find a constraint satisfy path. Hence, we argue that a chaotic attribute value distribution can also be a useful indication for users.



(a)Vary l (b)Vary Num of Hint

Figure 6.6: [**fb-bfs1**] Entropy

6.5.4 Change of Entropy

The default expected number of key vertex and number of hint are 6 and 3 respectively. We randomly generate 200 pairs of source and destination and measure the average entropy and execution time.

Figure 6.6(a) shows the change of entropy along with l. We can see that for both *CandPath* and *PathSummary*, the entropy does not really increase with l. Although it seems that a longer path would contain more vertices and is more likely to contain different attribute values, this intuition is not supported by Figure 6.6(a). Since large l offers more opportunity for the algorithm to search for $s - t$ paths with similar attribute values, the increase in path length does not directly imply an increase in entropy.

Figure 6.6(b) shows the change of entropy along with H. We can see that for both *CandPath* and *PathSummary*, the entropy increases with H. That is contributed by the fact that more attribute hints mean more attribute are involved, which makes the consistency of attribute values lower.

7.0 CONCLUSION

7.1 SUMMARY OF CONTRIBUTIONS

Attributed Graph is a popular data structure that, not only contains information of entities but also can efficiently represent relationships between them. Hence, by asking attributed graph queries, users can obtain meaningful information about topological and attribute relationship between entities in attributed graphs.

We first introduced and defined the reachability query on attributed graph problem. Based on this definition, we developed our approach in a 2-level storage framework. We proposed a new constraint verification approach which takes the advantage of a 'perfect' hash function [20, 21] for compressing a multi-dimensional attribute into a unique hash value so as to bound the expected number of secondary storage access. In order to further reduce the computation cost, we developed a heuristic search technique that takes into account graph structure as well as attribute distribution during graph traversal.

Then, we introduced and defined the How-to-Reach query, which answers why there is no attribute constraint satisfied path between entities in attributed graphs. We proposed a simple trick that 1)does not require heavy modification of existing implementations in graph database systems and 2)is proven to be able to allow existing shortest path algorithms to return optimal answers for How-to-Reach queries. Deal to the drawback of optimal answers (i.e. high computation cost and the extremely long hop distance), we proposed the station index, which is a time and space efficient non-traversal based index that returns high-quality sub-optimal answers with reasonable hop distances.

Finally, we introduced and defined the visualizable path summary query, which summarizes topological and attributes relationship between vertices based on users' intuition

on attribute values. We proposed a three-phrase approach, which finds, stitches, and inflates key vertex paths based on users' intuition on attribute values. The computed path summaries are a path-like set of vertices that can be visualized to users.

7.2 INTENTION

We intended to draw people's attentions on attributed graph query processing research. We believe that there is still penalty of meaningful attributed graph query types that have not been proposed and studied by the database and data mining community since attributed graph is a very rich source of information. These new analytical query types will help users in understanding more about hidden information contained in attributed graphs and offer support for decision making.

7.3 FUTURE WORK

7.3.1 Motivation

Attributed graph query tells the relationship between entities in attributed graphs. For example, attributed graph reachability query [19] answers whether there exists an attribute constraint satisfied path from the source to the destination; How-to-Reach query [23] offers answers for why there does not exist an attribute constraint satisfied path from the source to the destination. In both query types, we assumed that the topology and attribute values are truely reflecting the relationship information modeled by the attributed graph. For example, we assume that the Facebook social network contains everything about our personal information and social relationship, and by directly querying the Facebook social network, we can always find the relationship between 2 persons. We argue that indeed, this assumption may not always be true.

Attributed graph query may not be able to capture everything since some attribute or

topological information may not be provided, or are maliciously falsified or evolved.

- **Missing Information:** People has a lot of friends. However, some of Peter's friends are not his friend on the social network, and that is a common phenomenon. If we can imply the friendship between persons on the social network, we may be able to discover relationships between more people.
- **Malicious Falsification:** A terrorist is an active user in a social network. However, to hide himself/herself, the terrorist deliberately deleted some of his/her connections or modified values of some attributes in his/her profile. In fact, by recovering the deleted connections and modified attribute values, investigators may be able to identify the relationship between a target person and the terrorist leader.
- **Outdated View and Legacy Query:** From time to time, both topology, attribute value, and attribute schema of attribute graph may evolve so as to reflect the change in the real world as well as cater the needs of modeling new information. Hence, old views and legacy queries may not be able to fulfill the intended information needs since they were constructed based on an outdated understanding of the attributed graph.

7.3.2 Potential Direction

To improve answer quality of the attributed graph queryies (e.g. reachability query [19] and the How-to-Reach query [23]), we are seeking to build techniques that can effectively imply missing information, detect malicious falsifications, and synchronize historical queries in attributed graphs. We propose to add the 'Attributed Graph Profiler', 'Attributed Graph Query Modifier', and 'Attributed Graph Query Synchronizer' to our query processing system framework in Figure 2.3.

Attributed Graph Profiler: Conceptually, the attributed graph profiler is a component that implies missing information and detect malicious falsifications in attributed graphs. We are investigating the potential of building part of the core of this component by extending relaxed functional dependency mining techniques [77, 78] so that missing and falsified attribute values can be recovered and detected. Relaxed functional dependency mining looks for approximate relationships between attribute values. For example, we may be able to find

that country attribute is related to dietary habit attribute. Therefore, if someone modified the value of his/her dietary habit attribute, we may discover that person by looking at the value of his/her country attribute. Similarly, the other part of the core of this component is techniques that can discover hidden relationships in the topology. For example, if 2 persons have a lot of common friends, go to the same school, and are at the same age, probably, they know each other even though there is no edge between them on the social network. We need an effective approach to imply such relationship based on topological structure and attribute values.

Attributed Graph Query Modifier: The attributed graph query modifier refines user queries based on information offered by the attributed graph profiler so that the damage of missing information and malicious falsifications can be alleviated and ultimately, the answer quality can be improved. In order to refine the query, we need an effective way to exploit the information offered by the attributed graph profiler. For example, we need a metric to determine whether a certain attribute dependency should be taken into account when a vertex is visited during the graph traversal since the attribute dependency may not be valid for every vertex.

Attributed Graph Query Synchronizer: The attributed graph synchronizer updates old views and legacy queries so that the intended information needs can still be fulfilled in the evolved attributed graph. Although there exists query synchronization techniques [79] which could solve this problem in traditional databases, how to model the intended information needs and how to apply those techniques on attributed graph are still unclear.

8.0 BIBLIOGRAPHY

[1] Z. Wang, Q. Fan, H. Wang, K. Tan, D. Agrawal, and A. El Abbadi, "Pagrol: Parallel graph olap over large-scale attributed graphs," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pp. 496–507, 2014.

[2] S. Sakr, S. Elnikety, and Y. He, "G-SPARQL: a hybrid engine for querying large attributed graphs," in *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pp. 335–344, 2012.

[3] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 123–134, 2010.

[4] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao, "Efficient processing of label-constraint reachability queries in large graphs," *Inf. Syst.*, vol. 40, pp. 47–66, 2014.

[5] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pp. 405–416, 2003.

[6] M. Shoaran, A. Thomo, and J. H. Weber-Jahnke, "Zero-knowledge private graph summarization," in *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pp. 597–605, 2013.

[7] N. Zhang, Y. Tian, and J. M. Patel, "Discovery-driven graph summarization," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pp. 880–891, 2010.

[8] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *SIGMOD*, pp. 567–580, 2008.

[9] Y. Wu, Z. Zhong, W. Xiong, and N. Jing, "Graph summarization for attributed graphs," in *2014 International Conference on Information Science, Electronics and Electrical Engineering*, vol. 1, pp. 503–507, April 2014.

[10] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge," *J. Artif. Intell. Res. (JAIR)*, vol. 1, pp. 231–255, 1994.

[11] K. Khan, W. Nawaz, and Y. Lee, "Set-based approximate approach for lossless graph summarization," *Computing*, vol. 97, no. 12, pp. 1185–1207, 2015.

[12] L. Shi, H. Tong, J. Tang, and C. Lin, "VEGAS: visual influence graph summarization on citation networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 12, pp. 3417–3431, 2015.

[13] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han, "Mining graph patterns efficiently via randomized summaries," *PVLDB*, vol. 2, no. 1, pp. 742–753, 2009.

[14] S. Cebiric, F. Goasdoué, and I. Manolescu, "Query-oriented summarization of RDF graphs," *PVLDB*, vol. 8, no. 12, pp. 2012–2015, 2015.

[15] S. B. Roy, T. Eliassi-Rad, and S. Papadimitriou, "Fast best-effort search on graphs with multiple attributes," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 3, pp. 755–768, 2015.

[16] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *Proceedings of the 13th ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007, pp. 737–746, 2007.

[17] I. Herman, G. Melançon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Trans. Vis. Comput. Graph.*, vol. 6, no. 1, pp. 24–43, 2000.

[18] R. Pienta, J. Abello, M. Kahng, and D. H. Chau, "Scalable graph exploration and visualization: Sensemaking challenges and opportunities," in *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015*, pp. 271–278, 2015.

[19] D. Yung and S. Chang, "Fast reachability query computation on big attributed graphs," in *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pp. 3370–3380, 2016.

[20] "https://sites.google.com/site/murmurhash/,"

[21] "http://www.burtleburtle.net/bob/hash/spooky.html,"

[22] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pp. 1785–1794, 2011.

[23] D. Yung and S. Chang, "Answering how-to-reach query in big attributed graph databases," in *Third IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2017, Redwood City, CA, USA, April 6-9, 2017*, pp. 141–148, 2017.

[24] D. Yung and S. Chang, "Effective path summary visualization on attributed graphs," in *The 23th International Conference on Distributed Multimedia Systems: Research papers*

on distributed multimedia systems, distance education technologies and visual languages and computing, Pittsburgh, PA, USA, July 7-8, 2017., 2017.

[25] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *PVLDB*, vol. 2, no. 1, pp. 718–729, 2009.

[26] Y. Zhou, H. Cheng, and J. X. Yu, "Clustering large attributed graphs: An efficient incremental approach," in *ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, 14-17 December 2010*, pp. 689–698, 2010.

[27] Y. Liu, A. Dighe, T. Safavi, and D. Koutra, "A graph summarization: A survey," *CoRR*, vol. abs/1612.04883, 2016.

[28] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *PVLDB*, vol. 7, no. 12, pp. 1191–1202, 2014.

[29] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, 2003.

[30] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *PVLDB*, vol. 6, no. 14, pp. 1978–1989, 2013.

[31] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pp. 813–826, 2009.

[32] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pp. 595–608, 2008.

[33] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *Proceedings of the 22nd International Con-*

ference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, p. 75, 2006.

[34] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based algorithms for pattern matching on dags," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pp. 493–504, 2005.

[35] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum, "FERRARI: flexible and efficient reachability range assignment for graph indexing," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pp. 1009–1020, 2013.

[36] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pp. 845–856, 2007.

[37] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: a scalable index for reachability queries in very large graphs," *VLDB J.*, vol. 21, no. 4, pp. 509–534, 2012.

[38] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou, "I/O cost minimization: reachability queries processing over massive graphs," in *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pp. 468–479, 2012.

[39] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*, pp. 181–215, 2010.

[40] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," *Frontiers of Computer Science*, vol. 6, no. 3, pp. 313–338, 2012.

[41] K. A. Ross, D. Srivastava, and D. Papadias, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, ACM, 2013.

[42] K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, ACM, 2012.

[43] F. Wei, "TEDI: efficient shortest path query answering on graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 99–110, 2010.

[44] T. Akiba, C. Sommer, and K. Kawarabayashi, "Shortest-path queries for complex networks: exploiting low tree-width outside the core," in *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pp. 144–155, 2012.

[45] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in Ross *et al.* [41], pp. 349–360.

[46] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pp. 1601–1606, 2013.

[47] A. Chapman and H. V. Jagadish, "Why not?," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pp. 523–534, 2009.

[48] N. Bidoit, M. Herschel, and K. Tzompanaki, "Query-based why-not provenance with nedexplain," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pp. 145–156, 2014.

[49] M. Herschel and M. A. Hernández, "Explaining missing answers to SPJUA queries," *PVLDB*, vol. 3, no. 1, pp. 185–196, 2010.

[50] M. Herschel, M. A. Hernández, and W. C. Tan, "Artemis: A system for analyzing missing answers," *PVLDB*, vol. 2, no. 2, pp. 1550–1553, 2009.

[51] J. Huang, T. Chen, A. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," *PVLDB*, vol. 1, no. 1, pp. 736–747, 2008.

[52] C. Zong, X. Yang, B. Wang, and J. Zhang, "Minimizing explanations for missing answers to queries on databases," in *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part I*, pp. 254–268, 2013.

[53] S. S. Bhowmick, A. Sun, and B. Q. Truong, "Why not, wine?: towards answering why-not questions in social image search," in *ACM Multimedia Conference, MM '13, Barcelona, Spain, October 21-25, 2013*, pp. 917–926, 2013.

[54] Z. He and E. Lo, "Answering why-not questions on top-k queries," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 6, pp. 1300–1315, 2014.

[55] M. S. Islam, R. Zhou, and C. Liu, "On answering why-not questions in reverse skyline queries," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pp. 973–984, 2013.

[56] Q. T. Tran and C. Chan, "How to conquer why-not questions," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 15–26, 2010.

[57] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pp. 204–215, 2002.

[58] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel, "Horton+: A distributed system for processing declarative reachability queries over partitioned graphs," *PVLDB*, vol. 6, no. 14, pp. 1918–1929, 2013.

[59] S. Das, J. Srinivasan, M. Perry, E. I. Chong, and J. Banerjee, "A tale of two graphs: Property graphs as RDF in oracle," in *Proceedings of the 17th International Conference*

on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014., pp. 762–773, 2014.

[60] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pp. 41–52, 2012.

[61] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 31–46, 2012.

[62] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, pp. 135–146, 2010.

[63] "http://giraph.apache.org/,"

[64] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in Ross *et al.* [41], pp. 505–516.

[65] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 17–30, 2012.

[66] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[67] L. H. O. Rios and L. Chaimowicz, "A survey and classification of a* based best-first heuristic search algorithms," in *Advances in Artificial Intelligence - SBIA 2010 - 20th Brazilian Symposium on Artificial Intelligence, São Bernardo do Campo, Brazil, October 23-28, 2010. Proceedings*, pp. 253–262, 2010.

[68] "https://www.linkedin.com/company/linkedin-economic-graph,"

[69] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng, "A model-based approach to attributed graph clustering," in Candan *et al.* [42], pp. 505–516.

[70] "https://developers.facebook.com/docs/graph-api/reference/user."

[71] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++." http://snap.stanford.edu/snap, June 2014.

[72] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in Facebook: A Case Study of Unbiased Sampling of OSNs," in *Proceedings of IEEE INFOCOM '10*, (San Diego, CA), March 2010.

[73] "http://konect.uni-koblenz.de/networks/munmun_twitterex_ui."

[74] "https://www.mongodb.com/,"

[75] "http://orientdb.com/orientdb/,"

[76] "https://neo4j.com/,"

[77] L. Caruccio, V. Deufemia, and G. Polese, "On the discovery of relaxed functional dependencies," in *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*, pp. 53–61, 2016.

[78] L. Caruccio, V. Deufemia, and G. Polese, "Relaxed functional dependencies - A survey of approaches," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 1, pp. 147–165, 2016.

[79] L. Caruccio, G. Polese, and G. Tortora, "Synchronization of queries and views upon schema evolutions: A survey," *ACM Trans. Database Syst.*, vol. 41, no. 2, pp. 9:1–9:41, 2016.