Query Processing in Large-scale Networks

QIAO, Miao

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in

Systems Engineering and Engineering Management

The Chinese University of Hong Kong

July 2013

Thesis/Assessment Committee

ABSTRACT

Due to the massive size of graphs from various domains nowadays, even simple graph queries become challenging tasks. In this thesis, three queries with a wide range of applications are investigated on large graphs. One is *shortest distance* query, a fundamental query which computes the shortest distance between two nodes. Another query, *weight constraint reachability (*WCR*)*, checks if there is a feasible path between two nodes where edge weights along the path satisfy a side constraint. And the third one, a *top-k nearest keywords (*k-NK*)* query, reports, for a query node, the k nearest nodes bearing some user-specified keywords. When confronting with a large-scale graph with over tens of millions of nodes, we need to develop efficient indexing and query optimization techniques for these queries.

In this thesis, for a shortest distance query, we devise two landmark embedding schemes, an *error bounded landmark scheme* and a *local landmark scheme*, where the former can guarantee an error bound for estimated distance, and the latter can significantly improve the distance estimation accuracy without increasing the offline embedding or the online query complexity. For a WCR query, we propose a memory-based approach which promises a constant query time. Besides, in order to increase its scalability, we devise an I/O-efficient approach for answering a WCR query on massive graphs. For a k-NK query, we start with a special case when the graph is a tree, based on which we present our algorithm for approximate k-NK query on a graph. A global storage technique is devised to further reduce the index size and the query time. We did extensive experiments on the three queries respectively to show the effectiveness and efficiency of our methods.

摘要

由于现今在各个领域涌现的图数据规模都愈加庞大，在这些大规模图数据上进行任何一种简单的查询都成为一件有富有挑战性的工作。 在本文中，我们着重在大规模图上研究三个具有广泛应用的查询：最短路查询，权重限制查询和最近k关键字查询。具体来说， 最短路查询是一个计算两点间最短距离的基本查询。而权重限制查询判断两点间是否存在一条沿路边权都满足用户指定条件的可行路径。 对于一个查询节点，最近k关键字查询返回k个距离最近的带有指定关键字的节点。 在面对一个拥有超过一亿节点的图时，我们需要为这些查询开发有效的索引和查询优化算法。

在本文中，对于最短路查询，我们提出了两个基于地标嵌入的算法，一个是有误差控制的地标嵌入算法，另一个则是本地化地标嵌入算法。 前者通过对地标的筛选和组织，能对估计的最短距离给予一定的误差保证； 而后者提出的本地化机制能够在不增加预处理复杂度和在线查询复杂度的情况下大幅度提高估计的精准度。 对于权重限制查询，我们先提出一个能够保证常数查询时间的内存算法。 除此之外，为了提高算法对大规模数据的处理能力，我们使用编码技术设计了一个有效的外存算法。 对于最近k关键字查询，我们先在一个特殊的图，即一颗树上，开发一个有效算法来在常数时间内回答最近k关键字查询， 并由此得出一个图上的近似算法；此外我们还通过一个全局存储的技术来进一步减少索引大小和缩短查询时间。 我们在真实和模拟的数据上做了大量的实验，实验结果证明我们的算法在大图上对上述三个查询都具有高效性能。

ACKNOWLEDGEMENTS

It is my fortune to have worked in an exciting and supportive environment, and with a wonderful group of mentors and colleagues. I would like to express my heartfelt thanks to my advisors, Prof. Hong Cheng and Prof. Jeffrey Xu Yu, for their support and advice. They not only enlighten me with knowledge, but also affect me with their passions for research. I have benefited greatly from them in many perspectives especially from their rigorous and diligent attitude. In particular, I can remember vividly that before the deadline of each paper, Prof. Cheng worked and strove with me until the last minute. She is a patient listener on my problems, and gave me a lot of advice, guidance, and help in various aspects.

I am grateful to have the opportunity to work closely with Lijun Chang who gave me many invaluable advice when I started my research and helped me a great deal in refining my methods. I would like to thank Lu Qin, the collaboration with whom is tremendously beneficial for me. He is active and inspiring in discussions and shared with me a lot of insightful ideas. Without their help, this thesis would be a mission impossible. Additionally, I am indebted to Prof. Phillip S. Yu for his support and professional advice on my thesis.

Gratitude also goes to the committee members, Prof. Anthony So Man- Chao, Prof. Wai Lam, and Prof. Xuemin Lin for their comments and help through my Ph.D studies.

The environment of the Database Group in which I have been immersed is wonderful. I thank Yuanyuan Zhu who has kept me company, and shared my sorrow and happiness through my Ph.D life, and we have became close friends. Also I would like to thank Yiping Ke, Bingsheng He, Zheng Liu, Di Wu, Lu Qin, Lijun Chang, Ronghua Li, Xin Huang, Zhiwei Zhang, Zechao Shang, Wentao Tian, Yu Rong, Can Lu and Xiao Wen. The community of friends and fellow students in the Department of Systems Engineering and Engineering Management has also been great. Here I try to avoid listing too many names, but still want to mention Lidong Bing, Ke Zhou, Yun Shi, Bojun Lv, Baiyi Wu, Bingyang Li, Lanjun Zhou, Bo Chen, Wenting Hou, Quan Yuan and David Shu Wo Lai. I have special thanks to the staff of SEEM department, Leung Yuen Ling, So Wai Chun, Tsoi Wa Man and Leung Hoi Man, who had helped me a lot through my four-year study.

Finally, I would like to thank my parents, Xiaohua Qiao, and Yuxia Sun, for all their support. Without their unreserved and unconditional love, my four-year research would not have been such a delight.

I have been supported by a Postgraduate Studentship (PGS). This thesis is a joint work with several people mentioned above: Chapter 3 and 4 are based on the joint work with Lijun Chang, Chapter 4 and 5 are based on the joint work with Lu Qin, and Chapter 4 is based on the joint work with Prof. Phillip S. Yu.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

It is worth noting that nowadays, the size of graphs from various domains increases dramatically, and the number of nodes may reach the scale of hundreds of millions. Due to their massive size, even simple graph queries become challenging tasks. In this thesis, we concentrate on three important queries on large scale graphs, *shortest distance* query, *weight constraint reachability* query, and *top-k nearest keywords* query, and develop efficient indexing and query optimization techniques for them. In the following, we will introduce the three queries respectively.

1.1.1. Shortest Distance Query

As a fundamental query on graphs, a shortest distance query has been extensively studied for decades since 1950s. It has a wide range of applications in various domains including road networks, social networks, communication networks, web graphs, etc. For example, in a road network, the shortest distance between two locations is a key element for a route planning task; in a social network, the shortest distance between two users indicates the closeness of their social relationships such as friendship or collaboration; while in a communication network, the goal is to find the nearest server in order to reduce access latency for clients. Although classical algorithms such as breadth-first search (BFS), Dijkstra's algorithm [22], and A^* search [33, 29, 30] can compute the exact shortest paths in a network, the massive size of modern information networks and the online nature of such queries make it infeasible to apply the classical algorithms online. On the other hand, it is space inefficient to precompute and store the shortest distances between all pairs of nodes due to its quadratic space complexity.

Recently, there have been many different methods [26, 62, 81, 51, 84, 71, 53, 64, 78, 32, 68] for estimating the shortest distance between nodes based on graph embeddings. A commonly used embedding technique is *landmark embedding*, where a set of graph nodes is selected as *landmarks* [62, 64, 32] (also called *reference nodes* [53, 68], *beacons* [51], or *tracers* [26]) and the shortest distances from a landmark to all the other nodes in a graph are precomputed. Such precomputed distances can be used online to provide an approximate distance between two graph nodes based on the *triangle inequality*.

Error Bounded Landmark Scheme: For an embedding approach of computing the shortest distance, a theoretical error bound can guarantee the precision of the estimated distance, but the derivation of an error bound is closely related to the landmark selection strategy. According to the findings in the literature [26, 64], the problem of selecting the optimal landmark set is NP-hard, by a reduction from the classical NP-hard problems such as vertex cover or minimum K-center [27]. As a result, the existing studies use random selection [51, 71] or graph measure based heuristics [64] such as degree, betweenness centrality, closeness centrality, etc. These heuristics cannot derive an error bound to control the precision of the estimated distance.

In this thesis, we start with an investigation on how to provide a landmark embedding method with a user-specified error bound ϵ on their estimated distance. Specifically, we formulate a coverage-based landmark selection strategy, i.e., every node in a graph should be "covered" by some landmarks within a radius $c = \epsilon/2$. The coverage property will lead to a theoretical error bound of ϵ. Importantly, allowing a user-specified error bound increases the flexibility of our method in processing queries at different error tolerance levels. On the other hand, if a user specifies the number of landmarks he selects, we can find the corresponding value of c and the error bound. We will also show through experiments that by adjusting the radius c, we can achieve a tradeoff between the theoretical error bound and the offline computational time of the landmark embedding process.

Local Landmark Scheme: Despite that various rules are taken for landmark selection [51, 64], most existing methods use *triangular inequality*, which sums of the distances from two query nodes to a landmark as an estimation. As the landmark selection step is query independent, the landmark set provides a single global view for all possible queries which could be diameter apart or close by. Thus it is hard to achieve a uniformly good performance for all queries. As a consequence, the landmark embedding approach may introduce a large relative error, especially when the landmark set is distant from both nodes in a query but the two nodes themselves are close to each other. For example, in a US road network with 24 million nodes and 58 million edges, landmark embedding (with 50 randomly selected landmarks) has a maximum relative error of 68 for one query among $10,000$ random queries we tested.

This observation motivates us to find a query-dependent "local landmark" which is close to both query nodes for a more accurate distance estimation. In contrast, the original landmarks are called "global landmarks". In this thesis, we propose a *local landmark scheme* which identifies a local landmark specific to a pair of query nodes. Then the distance between the two query nodes is estimated to be the sum of their shortest distances to the local landmark, which is much closer than the global one. The query-dependent local landmark scheme is expected to reduce the distance estimation error in principle, compared with the traditional global landmark embedding.

1.1.2. Weight Constraint Reachability Query

Among many types of graph queries, graph reachability is an important type of query, which asks whether there exists a path from one node to another in a directed graph. Graph reachability has been studied extensively in the literature [2, 39, 19, 72, 79, 80, 34, 90, 86, 14, 17, 11, 45, 44, 42, 94, 24, 87, 41, 92] and has many potential applications. Most existing algorithms, however, do not consider realistic constraints on graph reachability that are very common and challenging in real-world applications, except a few recent works [42, 24, 41, 92] which consider adding categorical edge label constraint or distance constraint to reachability queries.

Many real-world networks contain real-value edge or node weights, for example, the bandwidth of a link in communication networks, the reliability of an interaction between two proteins in PPI networks, the handling capacity of a warehouse/storage point in a distribution network, etc. In many real-world applications, the answers to reachability queries are meaningful only if the edge or node weight is also captured in the reported path. Thus, in Chapter 4, we study, in weighted undirected graphs, a new type of reachability query called *weight constraint reachability* (WCR) query, which asks: is there a path between nodes a and b, on which each real-value edge (or node) weight satisfies a range constraint, e.g., $\geq x$, $\leq y$, or within $[x, y]$. The WCR query has many real application scenarios. Here we list several application examples.

Communication Networks: Transmission of multimedia streams imposes a minimum-bandwidth requirement on all the links on a path to ensure end-to-end Quality-of-Service (QoS) guarantees [58]. A WCR query can find whether there is a feasible path between two nodes in a network, on which each link has a bandwidth $\geq x$, i.e., a minimum-bandwidth requirement. The resulting path can support a rate of x bits per second for transmitting a stream, e.g., audio or video, with a bandwidth guarantee.

Biological Networks: In a PPI network, a node represents a protein, and an edge represents an interaction between two proteins with a real-value weight to denote the reliability of the interaction. A WCR query can find whether there is a path between two proteins where the reliability of every interaction is $\geq x$. A lot of research has been proposed to find signaling pathways from PPI networks, e.g., [7]. However, many false positive candidates will be generated. The WCR query can be used to prune these false positives.

Phone Call Networks: From a phone call log, we can construct a phone call network, where a node represents a caller ID and an edge represents a phone call labeled with the time stamp when the phone call is made, between two callers. A WCR query can find whether there is a chain of calls between two callers, each of which is made during a time period $[t_1, t_2]$. This query can be useful for security reasons such as crime detection. Similarly the WCR query can be applied to social networks for relationship analysis.

The WCR query can also be applied to node-weighted networks to ensure that the weight of each node on the reported path satisfies a constraint. An example is given below.

Distribution Networks: If a firm plans to ship its products from a factory to a retailer store located in distant locations, a WCR query can find whether there is a feasible delivery route between these two locations in the distribution network, on which each intermediate warehouse, storage point or distribution center has a proper handling capacity $\geq x$. This query can help facilitate delivery and distribution of the products, and improve the operational efficiency of the supply chain.

To the best of our knowledge, there is no existing reachability study on the real-value weight constraint. In the literature, [42, 92] study categorical edge label constraint reachability (LCR). In addition, Regular Path Query (RPQ) [59], Conjunctive RPQ (CRPQ) [25] and Reachability Query (RQ) [24] have been proposed for full or a subclass of regular expression constraint on categorical edge labels. The real-value weight constraint we study is very different from those in [42, 92, 59, 25, 24] in the following aspects: (1) there is a total order among real-value weights, but no order among categorical labels. A much more optimized solution can be designed for

WCR by exploiting the total-order property; (2) the cardinality of a real-value weight set is typically much larger than that of a categorical label set. A large weight set can substantially increase the indexing and query complexity of the existing methods. Take the Sampling-Tree method proposed in [42] for LCR query as an example, its index construction time grows exponentially with the number of distinct labels in a graph, and its query time increases linearly with the number of distinct labels. Given a large real-value weight set, the WCR query can hardly be answered efficiently by directly applying Sampling-Tree. Other works including NP-Hard query RPQ [59], NPC query CRPQ [25], and $O(|V|^2)$ query time RQ [24] face the same problem for handling real-value weight set.

In Chapter 4, we aim to design efficient algorithms to answer the WCR query with a compact index. We mainly focus on the edge weight constraint, and show that the node weight constraint can be easily reduced to the edge weight constraint. Given an undirected graph G and a WCR query, we exploit the cut property of minimum spanning tree (MST) to show that checking whether two nodes are reachable in G w.r.t. a constraint can be transformed to checking such reachability in an MST of G. This property serves as a building block for designing a novel index called Edge-Index. It organizes the MST edges hierarchically based on an elegant transformation of MST so that we can answer a WCR query in $O(1)$ time.

Considering the networks emerging nowadays typically contain hundreds of millions of vertices or even more, the index size of Edge-Index in $O(|\Sigma||V|)$[1] (Σ is the edge weight set and V is the vertex set of the graph) may easily exceed the memory limit. Therefore, to answer the WCR query, we further design an I/O-efficient disk-based index, Balanced-Index, which is constructed by recursively adjusting an MST into a balanced tree. A nice property of the balanced tree is the $\log_2 |V|$ worst-case tree height, which effectively compresses the disk-based index size to $O(|\Sigma||V| \log |V|)$ while guaranteeing to answer the WCR query with exactly four I/Os. Our algorithm

[1]The $O(|\Sigma||V|)$ space complexity is for handling the general bounded interval constraint $[x, y]$, $x, y \in \mathbb{R}$. For the half-bounded constraint $\geq x$ or $\leq y$, the complexity is $O(|V|)$.

Balanced-Index proves to be I/O-efficient and highly scalable. This is a very significant contribution, as all existing algorithms on graph reachability in the literature are limited to main memory based algorithms.

1.1.3. Top-k Nearest Keyword Query

Many real-world networks emerging nowadays have labels or textual contents on the nodes. For example in a road network, a location may have labels such as "McDonald's", "hospital", and "kindergarten". In a social network, a person may have information including name, interests and skills, etc.. In a bibliographic network, a paper may have keywords and abstract, and an author may have name, affiliation and email address. In this thesis, we consider the problem of *top-k nearest keyword* (k-NK) search on large networks. In a network G modeled as an undirected graph, each node may be attached with one or more keywords, and each edge is assigned with a weight measuring its length. Given a query node q in G and a keyword λ, a k-NK query in the form of $Q = (q, \lambda, k)$ looks for k nodes which contain λ and are nearest to q. Different from a large body of research on k-nearest neighbor (k-NN) search on spatial networks [52, 15, 18, 73, 76, 21], we define G as a general graph without coordinates. Thus our solution can apply to a wide range of networks.

In graph search, a k-NK query is important and useful. As a stand-alone query, it has a wide range of applications. Furthermore, it can serve as a building block for tackling complex graph pattern matching problems which impose both structural and textual constraints. Here we list a few applications of k-NK queries.

Consider the social network Facebook as an example. Note that personalized search based on graph structure and textual contents has become increasingly popular in Facebook[2]. A person looks for 20 friends or potential friends who like *hiking* to participate in a hiking activity. Intuitively, if two persons share some common friends, i.e., they are within two hops away, they are more likely to become friends. In contrast, if they are far away from each other in the network, they are less likely to

[2]https://www.facebook.com/about/graphsearch

establish a link. Thus, for the person who serves as the organizer, the problem is to find 20 nearest persons who like *hiking*. It can be answered by a k-NK query. More generally, we also consider a query containing multiple keywords connected by AND or OR operators to express more complex semantics, e.g., a person looks for k friends or potential friends who like *hiking* AND (OR) *photography* and are nearest to him.

Take a road network with locations associated with keywords as another example. For parents looking for k *kindergartens* nearest to their home for their children, their requirements can be expressed by a k-NK query where the query node is the home location, and the keyword is "kindergarten".

In the third example, we show how k-NK queries serve as a building block for solving the graph pattern matching problem. Consider a couple who wants to buy a house. They have some constraints like *having a kindergarten and a hospital within 3 km, and a supermarket within 1 km of their home*. These constraints can be expressed as a star pattern, and the pattern matching problem can be decomposed into three k-NK queries with keywords "kindergarten", "hospital" and "supermarket" respectively and $k = 1$ for each potential house location to be considered.

Recently, Bahmani and Goel [5] have designed a *Partitioned Multi-Indexing* (PMI) scheme to answer k-NK queries approximately. PMI is an inverted index built based on distance oracle [78] which is a distance estimation technique. Given a k-NK query $Q = (q, \lambda, k)$, it returns k nodes containing keyword λ in ascending order of their approximate distance from the query node q. PMI inherits the $2\log_2|V| - 1$ approximation factor for distance estimation from distance oracle [78], where V is the set of nodes in the graph. The major drawback of PMI is that its distance estimation error could be quite large in practice. This can greatly distort the ranking of the candidate nodes carrying the query keywords, and thus lead to a low result quality.

In this thesis, we study how to answer k-NK queries accurately and efficiently using a compact index. The key to an accurate k-NK result is a precise shortest distance estimation in a graph. In a general graph model, existing k-NN solutions on spatial networks [52, 15, 18, 73, 76, 21] cannot be applied, as they usually rely on specialized structures that leverage properties of spatial data to optimize their solutions. Instead, we use distance oracle [78] as the fundamental distance estimation framework. For each component of a distance oracle, we will build a shortest path tree, based on which we can estimate the shortest distance between two nodes by their tree distance. The tree distance is more accurate than the distance estimated by distance oracle, which we call *witness distance* to distinguish it. As we transform a distance oracle on a graph into a set of shortest path trees, the original k-NK query on the graph can be reduced to answering the k-NK query on a set of trees. Thus we first focus on processing k-NK queries to find the exact top-k answers on a tree. Then we study how to assemble the results obtained from the trees to form the approximate top-k answers on the graph.

1.2. Contributions

In this thesis, we devise two landmark embedding schemes for a shortest distance query, one is an *error bounded landmark scheme* [68] and the other is a *local landmark scheme* [65, 66].

An *error bounded landmark scheme* is based on a coverage-based strategy, which leads to a theoretical error bound of estimated distance. Since selecting the minimum set of landmarks is an NP-hard problem, we propose a greedy solution based on the submodular property of the proposed objective function. We show that the estimated distance is within a user-specified error bound of the true distance. To further reduce the offline computational complexity, we propose a graph partitioning-based heuristic for landmark embedding with a relaxed error bound.

A query-dependent *local landmark scheme* is proposed in light of an observation that in traditional landmark embedding, the query-independent global landmark selection introduces a large relative error, especially for nearby query nodes which are distant from the global landmarks. It finds a local landmark close to both query nodes to improve the distance estimation accuracy. The local landmark scheme proves to be a robust embedding solution that substantially reduces the dependency of query performance on the global landmark selection strategy. Given a query node pair, the proposed local landmark scheme finds a local landmark, which is defined as the *least common ancestor* (LCA) of the two query nodes in the SPT rooted at one of the global landmarks. An $O(1)$ time algorithm for finding the LCA on an SPT is introduced. We show that the SPT based local landmark scheme can significantly improve the distance estimation accuracy, without increasing the offline embedding or the online query complexity. Facing the challenge of immense graphs whose index may not fit in the memory, we also study to store the embedding in relational database, so that a query of the local landmark scheme can be expressed with relational operators.

In terms of *weight constraint reachability* (WCR) problem, to the best of our knowledge, our work [67] is the first study. We design a novel Edge-Index as an efficient memory-based index to answer a WCR query by exploiting the cut property of minimum spanning tree. When the index is too large to fit in the memory, we design a Balanced-Index as an I/O-efficient disk-based index. Remarkably, our in-memory algorithm Edge-Index achieves $O(1)$ query time and $O(|\Sigma||V|)$ index size, while our I/O-efficient algorithm with Balanced-Index uses four I/Os for query processing and $O(|\Sigma||V|\log|V|)$ disk space for indexing. We conducted extensive experiments on large real and synthetic networks. The query time of both Edge-Index and Balanced-Index algorithms is in microseconds and remains stable regardless of the network size, the density, the cardinality of the edge weight set or the weight distribution. Balanced-Index proves to be I/O-efficient and highly scalable for querying large networks. Finally our query processing is at least three orders of magnitude faster than basic search approaches.

We delve into *top-k nearest keyword* problem on graphs and our approaches [69] significantly outperforms the state-of-the-art solutions. We start with a special case when the graph is a tree under a common scenario when users are interested in a small number of answer nodes bounded by a small constant $\overline{k}$, i.e., $k \leq \overline{k}$ on a tree. We propose the first algorithm tree-boundk with query time $O(k + \log|V_\lambda|)$, where $|V_\lambda|$ is the number of nodes carrying the query keyword λ, and index size $O(\overline{k} \cdot |\mathsf{doc}(V)|)$, where $|\mathsf{doc}(V)|$ is the total number of keywords on all the nodes in the graph. Next we remove the $\overline{k}$ restriction and handle k-NK queries for an arbitrary k on a tree. We propose the second algorithm tree-pivot with query time $O(k \cdot \log|V|)$ and index size $O(|\mathsf{doc}(V)| \cdot \log|V|)$ which is independent of k, thus it is more scalable. Based on our proposed tree algorithms, we present our algorithm for approximate k-NK query on a graph. We propose a global storage technique to further reduce the index size and the query time. We also show how to extend our methods to handle a query with multiple keywords. Our experimental evaluation demonstrates the effectiveness and efficiency of our k-NK algorithms on large real-world networks. We show the superiority of our methods in ranking top-k answer nodes accurately when compared with the state-of-the-art top-k keyword search method PMI [5].

1.3. Roadmap

The rest of the thesis is organized as follows. Chapter 2 reviews the previous works related to ours. Chapter 3 introduces two landmark embedding schemes, an error bounded landmark scheme in Chapter 3.2 and a local landmark scheme in Chapter 3.3, for a shortest distance query. Chapter 4 shows both main-memory and disk-based methods for a WCR query. Chapter 5 elaborates on a series of methods we propose to answer a k-NK query. Finally, Chapter 6 concludes the thesis.

CHAPTER 2

RELATED WORK

2.1. Shortest Distance Query

As a fundamental query, *shortest distance query* has been extensively studied for decades since 1950s. In the literature, existing solutions are proposed either for exact shortest distance queries or approximate ones, so in the following, we will shall introduce them.

Exact Shortest Distance Query: For computing the exact shortest distance between two query nodes, there are two extreme approaches. One approach is to perform Dijkstra's algorithm (or BFS on unweighted graphs) from the source node online. It may incur, however, seconds of delay before reporting the result and thus is not appropriate for large-scale networks. Another approach is to precompute *All-Pair-Shortest-Path(APSP)* and then store them as an index. Although it takes $O(1)$ time for reporting the shortest distance of a query, its cubic precomputation time and quadratic index size impede the approach from being applied on large-scale networks.

For general graphs, as far as we know, there are two directions for finding a balance between space and query time. One is $\textsf{2-Hop Cover}$ [19, 16, 43, 3], also called *labeling method*. The basic idea is: each node u precomputes its distance to a set of nodes $L(u) \subseteq V$ as labels, such that for each node pair $u, v \in V$, $L(u) \cap L(v) \cap \mathsf{SP}(u, v) \neq \emptyset$, where set $\mathsf{SP}(u, v) = \{a \in V | \mathsf{dist}(u, a) + \mathsf{dist}(a, v) = \mathsf{dist}(u, v)\}$ contains all nodes that lie on some shortest path between u to v. The main drawback of these approaches is that there is no absolute guarantee on the size of $L(\cdot)$. Another direction is to devise an indexing and querying scheme based on the tree decomposition of a graph [13, 91, 4]. However, due to the NP-completeness of the optimal tree-decomposition, the tree-width found by an approximate algorithm could increase the index size to quadratic which is not practical for large-scale networks.

Approximate Shortest Distance Query: Approximate shortest distance computation is studied as a cost-effective balance between space, query time, and precision. An ideal approximate approach has a constant or logarithmic query time with a linear or loglinear space requirement. *Triangle inequality* is used as a part of *landmark embedding* technique to estimate the upper bound of a shortest distance. The basic idea is: each node u precomputes its distance to a set of nodes $L(u) \subseteq V$, and for a node pair $u, v \in V$, their approximate distance is estimated as $\min_{a \in L(u) \cap L(v)} \{\mathsf{dist}(u, a) + \mathsf{dist}(a, v)\}$. The nodes selected for the label set $L(\cdot)$ are called *landmarks*([62, 64, 32]), or *reference nodes*([53, 68]), or *beacons*([51]), or *tracers*([26]).

Landmark embedding technique [81, 53, 71, 64, 78, 32, 68, 26, 85, 84, 51] has been widely used to estimate the distance between two nodes in a graph in many applications including road networks [81, 53], social networks and web graphs [71, 64, 78, 32, 68] as well as the Internet [26, 62]. Shahabi et al. [81] utilize Linial, London and Robinovich (LLR) embedding to estimate the distance between two nodes. Kriegel et al. [53] propose a hierarchical reference node embedding approach which organizes reference nodes on multiple levels for a better scalability. Potamias et al. [64] formulate the reference node selection problem to select nodes with high betweenness centrality. [26] proposes an architecture, called IDMaps, which measures and disseminates distance information on the global Internet. [51] defines a notion of slack – a certain fraction of all distances that may be arbitrarily distorted as a performance guarantee based on randomly selected reference nodes. In addition, *distance oracle* [85, 84, 78, 32] reports an approximate distance with a $2k - 1$ stretch using an

$O(|V|^{1+\frac{1}{k}})$ sized index. To summarize, the major differences between the above methods lie in the following aspects: (1) landmark selection – some [51, 84, 71, 78, 32] select landmarks randomly, while others [26, 62, 53, 64] use heuristics; (2) landmark organization – some methods organize landmarks on multiple levels [84, 53, 78, 32], while other methods use a flat landmark embedding; and (3) whether they have an error bound or not – [51, 84, 78, 32, 68] analyze the error bound of the estimated distances, while most of the other methods have no error bounds or guarantees of the estimated distances.

Shortest distance in spatial networks: There have been a lot of studies on computing shortest paths queries in spatial networks [63, 38, 77, 75]. Papadias et al. [63] use the Euclidean distance as a lower bound to prune the search space and guide the network expansion for refinement. Hu et al. [38] propose an index called distance signature for distance computation and query processing, which discretizes the distances between objects and network nodes into categories to obtain an encoding. For a spatial network of dimension d, [77, 75] can retrieve an ε-approximation distance estimation in $O(\log n)$ time using an index termed path-distance oracle of size $O(n \cdot \max(s^d, \frac{1}{\epsilon}^d))$. In addition, on road networks, the problem of *route planning* has been extensively studied, where [29, 30] takes *graph annotation approaches*, and [28, 74, 6, 1] takes *hierarchical approaches*.

2.2. Reachability Query

Reachability query on directed graphs has been studied extensively with many algorithms proposed [2, 39, 19, 72, 79, 80, 34, 90, 86, 14, 17, 11, 45, 44, 42, 94, 24, 87, 41, 92]. These algorithms usually design a certain type of coding for graph nodes so that reachability queries can be answered by checking the coding of the nodes involved. The codings include tree cover [2], chain cover [39, 14], dual labeling [90], GRIPP index [86], path-tree cover [45], 2-hop cover [19, 79, 80, 17, 11], 3-hop [44], randomized interval labeling [94], and bit vector compression of transitive closure [87].

Most existing reachability algorithms do not consider vertex or edge label constraints except a few recent works [42, 92, 59, 25, 24]. [42] studies label-constraint reachability, given a set of categorical edge labels as the constraint. It utilizes the directed maximal weighted spanning tree and sampling techniques to compress the generalized transitive closure for the edge-labeled graphs. The index construction time increases exponentially with the label set size $|\Sigma|$, thus is not scalable to handle a large label set. [92] solves the same problem by proposing a Dijkstra-like algorithm to compute path-label transitive closure. [59] proves Regular Path Query (RPQ), a path query with regular expression constraints, is NP-Hard; [25] shows Conjunctive RPQ (CRPQ) is a NPC problem; and [24] studies adding a subclass of regular expressions (RQ) to specify the reachability via a path of certain edge types and of a possibly bounded length. [24] proposed two algorithms which answer a query in $O(|V|^2)$ time. One algorithm uses a matrix of shortest distances as index, and the other uses online bi-directional search. [41] studies distance-constraint reachability in uncertain graphs, and proposes two probabilistic estimators for the probabilistic reachability.

The existing reachability solutions cannot be directly or efficiently applied to answer the WCR query, as they focus on a different problem setting which does not have the total-ordering property on edge labels. In addition, all existing reachability algorithms in the literature are main memory based algorithms which assume the index resides in the memory and do not consider the I/O cost in query processing. Our work is the first to design a disk-based I/O-efficient algorithm to answer the WCR query.

2.3. Keyword Related Query

Keyword search: Keyword search in a graph(or database) finds a substructure of the graph containing the query keywords. The answer substructure can be a tree [37, 10, 47, 23, 35, 31], a subgraph [57, 70] or a r-clique [48]. A survey on keyword search in databases and graphs can be found in [95]. Keyword search has substantial differences from the k-NK query studied in this paper. In terms of problem definition, keyword search looks for a network structure, the nodes in which jointly contain all the query keywords, whereas a k-NK query looks for k nearest answer nodes, each one of which contains all the query keywords. In terms of solution, keyword search performs BFS or Dijkstra's algorithm to find the answer networks, whereas our proposed solutions build an index structure based on distance oracles and compact trees for keywords. Therefore, our query time efficiency is much higher than BFS and Dijkstra's algorithm, which has also been confirmed in our experiments. [93] and [12] study *keyword routing* on a road network. Given a keyword set, a source and a target locations, the goal is to find the shortest path that passes through at least one matching object for each keyword.

K **nearest neighbor in spacial networks(k-NN):** k-NN search has been extensively studied in spatial networks [52, 15, 18, 73, 76, 21]. [52] uses network Voronoi polygons to divide a graph into disjointed subsets for k-NN search. Jagadish et al. [40] compress high-dimensional data points to one-dimensional values based on a set of well selected reference nodes and then apply range search using a B^+ tree index to answer KNN queries. [15, 18] use R-tree to embed textual information on nodes, and augment a tree node with inverted index for spatial document within the MBR. [73] answers k-NN queries with the shortest path quadtree. [76] answers k-NN queries based on ε-approximated distance estimated by an index termed path-distance oracle of size $O(n \cdot \max(s^d, \frac{1}{\epsilon}^d))$ where d is the network dimension. [21] performs Dijkstra-like expansion from the query node. However the above approaches designed for spatial networks cannot be applied to graphs without coordinates.

Nearest Keyword Query: The most related work to our k-NK study includes nearest keyword search on XML documents [83] and top-k nearest keyword search on graphs [5], both of which will be introduced in detail in Chapter 5.2. Besides, Hermelin et al. [36] adapt the distance oracle [85] to answer 1-NK queries with a $4k-5$ stretch in $O(k)$ time using an $O(k|V|^{1+\frac{1}{k}})$ sized index.

CHAPTER 3

QUERYING SHORTEST DISTANCE

3.1. Landmark Embedding

Consider a weighted undirected graph $G = (V, E, w)$, where V is a set of vertices, E is a set of edges, and $w : E \mapsto \mathbb{R}^+$ is a weighting function mapping an edge $(u, v) \in E$ to a positive real number $w(u, v) > 0$, which measures the length of (u, v). We denote $n = |V|$ and $m = |E|$. For a pair of vertices $a, b \in V$, we use $\mathsf{dist}(a, b)$ to denote the shortest distance between a and b, and $P(a, b) = (a, v_1, v_2, \dots, v_{l-1}, b)$ to denote the shortest path, where $\{a, v_1, \dots, v_{l-1}, b\} \subseteq V$ and $\{(a, v_1), (v_1, v_2), \dots, (v_{l-1}, b)\} \subseteq E$.

Given a pair of query nodes (a, b), to efficiently estimate an approximate shortest distance between a and b, a commonly adopted approach is *landmark embedding*. Consider a set of nodes $S = \{l_1, \dots, l_k\} \subseteq V$, or denoted as $\mathcal{R}$, which are called *landmarks*. For each $l_i \in S$, we compute the shortest distances to all nodes in V. Then for every node $v \in V$, we can use a k-dimensional vector $\overrightarrow{\mathsf{dist}}(v) = \langle \mathsf{dist}(l_1, v), \mathsf{dist}(l_2, v), \dots, \mathsf{dist}(l_k, v) \rangle$ to represent its distances to the k landmarks. This is called landmark embedding, which can be used to compute an approximate shortest distance between nodes a and b based on the triangle inequality as

$$\widetilde{\mathsf{dist}}(a, b) = \min_{l_i \in S} \{\mathsf{dist}(l_i, a) + \mathsf{dist}(l_i, b)\} \tag{3.1}$$

This general embedding approach has been widely used in many existing methods in the literature.

Evaluation: For a node pair (s,t), we use the relative error, denoted as $\mathsf{err}(s,t)$ defined as below, to evaluate the quality of an estimated distance in Chapter 3.2 and 3.3.

$$\mathsf{err}(s,t) = \frac{|\widetilde{\mathsf{dist}}(s,t) - \mathsf{dist}(s,t)|}{\mathsf{dist}(s,t)}$$

3.2. Error Bounded Landmark Scheme

3.2.1. Problem Statement

Problem 3.1 (Distance Estimation with a Bounded Error)**.** Given a graph G and a user-specified error bound ϵ as input, for any pair of query vertices (s,t), we study how to efficiently provide an accurate estimation of the shortest distance $\widetilde{\mathsf{dist}}(s,t)$, so that the estimation error $|\widetilde{\mathsf{dist}}(s,t) - \mathsf{dist}(s,t)| \leq \epsilon$.

In the rest of Chapter 3.2, we will discuss the following questions: 1) Given a graph G and an error bound ϵ, how to select the minimum number of landmarks to ensure the error bound ϵ in the distance estimation? 2) How to estimate the shortest distance with an error bound given a query (s,t)?

3.2.2. Proposed Algorithm

The quality of the estimated shortest distance is closely related to the landmark selection strategy. Given a graph G and an error bound ϵ, we will first formulate a coverage-based landmark selection approach to satisfy the error bound constraint. We will then define an objective function over a set of landmarks and discuss how to select the minimum set of landmarks according to the objective function.

Coverage-based Landmark Selection

Definition 3.1 (Coverage)**.** Given a graph $G = (V, E, w)$ and a radius c, a vertex $v \in V$ is covered by a landmark r if $\mathsf{dist}(r, v) \leq c$.

The set of vertices covered by a landmark r is denoted as C_r, *i.e.*, $C_r = \{v | v \in V, \mathsf{dist}(r, v) \leq c\}$. In particular, we consider a landmark r is covered by itself, *i.e.*, $r \in C_r$, since $\mathsf{dist}(r, r) = 0 \leq c$. Here we formulate the problem of optimal landmark selection.

Problem 3.2 (Coverage-based Landmark Selection)**.** Given a graph $G = (V, E, w)$ and a radius c, our goal is to select a minimum set of landmarks $\mathcal{R}^* \subseteq V$, *i.e.*, $\mathcal{R}^* = \arg\min_{\mathcal{R} \subseteq V} |\mathcal{R}|$, so that $\forall v \in V - \mathcal{R}^*$, v is covered by at least one landmark from $\mathcal{R}^*$.

Given a user-specified error bound ϵ, we will show in Chapter 3.2.2, when we set $c = \epsilon/2$, the coverage-based landmarks selection method can guarantee that the error of the estimated shortest distance is bounded by ϵ.

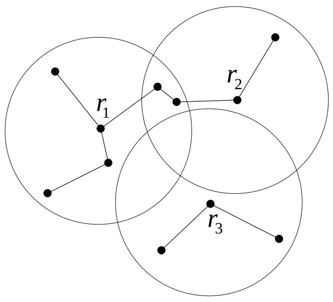


Figure 3.1: Coverage-based Landmark Selection

Example 3.1. Figure 3.1 shows a graph with three landmarks r_1, r_2 and r_3. The three circles represent the area covered by the three landmarks with a radius c. If a vertex lies within a circle, it means the shortest distance between the vertex and the corresponding landmark is bounded by c. As shown in the figure, all vertices can be covered by selecting the three landmarks.

Besides the coverage requirement, a landmark set should be as compact as possible. To evaluate the quality of a set of landmarks $\mathcal{R}$, we define a gain function over

$\mathcal{R}$.

Definition 3.2 (Gain Function)**.** The gain function over a set of landmarks $\mathcal{R}$ is defined as

$$g(\mathcal{R}) = |\bigcup_{r\in\mathcal{R}} C_r| - |\mathcal{R}| \tag{3.2}$$

In Figure 3.1, $g(\{r_1\}) = 5$, $g(\{r_2\}) = 3$, $g(\{r_3\}) = 2$ and $g(\{r_1, r_2, r_3\}) = 8$.

The gain function g is a submodular function, as stated in Theorem 3.1.

Definition 3.3 (Submodular Function)**.** Given a finite set N, a set function $f : 2^N \to R$ is submodular if and only if for all sets $A \subseteq B \subseteq N$, and $d \in N \setminus B$, we have $f(A \cup \{d\}) - f(A) \geq f(B \cup \{d\}) - f(B)$.

Theorem 3.1. *For two landmark sets $A \subseteq B \subseteq V$ and $r \in V \setminus B$, the gain function g satisfies the submodular property:*

$$g(A \cup \{r\}) - g(A) \geq g(B \cup \{r\}) - g(B)$$

Proof. According to Definition 3.2, we have

$$\begin{aligned} g(A \cup \{r\}) - g(A) &= |C_A \cup C_r| - (|A| + 1) - |C_A| + |A| \\ &= |C_A \cup C_r| - |C_A| - 1 \\ &= |C_r - C_A| - 1 \end{aligned}$$

where $C_r - C_A$ represents the set of vertices covered by r, but not by A.

Since $A \subseteq B$, we have $C_r - C_B \subseteq C_r - C_A$, hence $|C_r - C_B| \leq |C_r - C_A|$. Therefore, the submodular property holds. □

As our goal is to find a minimum set of landmarks $\mathcal{R}^*$ to cover all vertices in V, it is equivalent to maximizing the gain function g:

$$\max_{\mathcal{R}} g(\mathcal{R}) = \max_{\mathcal{R}}(|\bigcup_{r\in\mathcal{R}} C_r| - |\mathcal{R}|) = |V| - \min_{\mathcal{R}} |\mathcal{R}| = g(\mathcal{R}^*)$$

In general, maximizing a submodular function is NP-hard [50]. So we resort to a greedy algorithm. It starts with an empty set of landmarks $\mathcal{R}_0 = \emptyset$ with $g(\mathcal{R}_0) = 0$. Then it iteratively selects a new landmark which maximizes an additional gain, as specified in Eq.(3.3). In particular, in the k-th iteration, it selects

$$r_k = \arg \max_{r \in V \setminus \mathcal{R}_{k-1}} g(\mathcal{R}_{k-1} \cup \{r\}) - g(\mathcal{R}_{k-1}) \tag{3.3}$$

The algorithm stops when all vertices in V are covered by the landmarks. The greedy algorithm returns the landmark set $\mathcal{R}$.

Continue with our example. According to the greedy selection algorithm, in the first step, we will select r_1 as it has the highest gain. Given $\mathcal{R}_1 = \{r_1\}$, we have $g(\{r_1, r_2\}) - g(\{r_1\}) = 1$ and $g(\{r_1, r_3\}) - g(\{r_1\}) = 2$. So we will select r_3 in the second step. Finally we will select r_2 to cover the remaining vertices. Note that to simplify the illustration, we only consider selecting landmarks from r_1, r_2, r_3 in this example. Our algorithm actually considers every graph vertex as a candidate for landmarks.

To effectively control the size of $\mathcal{R}$, we can further relax the requirement to cover all vertices in V. We observe that such a requirement may cause $|\mathcal{R}|$ unnecessarily large, in order to cover the very sparse part of a graph or the isolated vertices. So we set a parameter *Cover Ratio* (CR), which represents the percentage of vertices to be covered. The above greedy algorithm terminates when a fraction of CR vertices in V are covered by $\mathcal{R}$.

Error Bound Analysis

In this subsection, we will show that, given a query (s, t), when s or t is covered within a radius c by some landmark from $\mathcal{R}$, the estimated distance $\widetilde{\mathsf{dist}}(s, t)$ is within a bounded error of the true distance $\mathsf{dist}(s, t)$.

Theorem 3.2. *Given any query (s, t), the error of the estimated shortest distance $\widetilde{\mathsf{dist}}(s, t)$ can be bounded by $2c$ with a probability no smaller than $1 - (1 - \mathsf{CR})^2$, where c is the coverage radius and CR is the cover ratio.*

Proof. Given a query (s, t) and a landmark set $\mathcal{R}$, assume s is covered by a landmark, denoted as r^*, *i.e.*, $\mathsf{dist}(s, r^*) \leq c$. Without loss of generality, we assume $\mathsf{dist}(s, r^*) \leq \mathsf{dist}(r^*, t)$. Note that the following error bound still holds if $\mathsf{dist}(s, r^*) > \mathsf{dist}(r^*, t)$. The error of the estimated shortest distance between (s, t) is bounded by

$$\begin{aligned}
\mathsf{err}(s,t) &= \widetilde{\mathsf{dist}}(s,t) - \mathsf{dist}(s,t)\\
&= \min_{r\in\mathcal{R}}(\mathsf{dist}(s,r) + \mathsf{dist}(r,t)) - \mathsf{dist}(s,t)\\
&\leq \mathsf{dist}(s,r^*) + \mathsf{dist}(r^*,t) - \mathsf{dist}(s,t)\\
&\leq \mathsf{dist}(s,r^*) + \mathsf{dist}(r^*,t) - |\mathsf{dist}(s,r^*) - \mathsf{dist}(r^*,t)|\\
&= 2\mathsf{dist}(s,r^*)\\
&\leq 2c
\end{aligned}$$

The first inequality holds because $\min_{r\in\mathcal{R}}(\mathsf{dist}(s,r) + \mathsf{dist}(r,t)) \leq \mathsf{dist}(s,r^*) + \mathsf{dist}(r^*,t)$; and the second inequality holds because we have the lower bound property $\mathsf{dist}(s,t) \geq |\mathsf{dist}(s,r^*) - \mathsf{dist}(r^*,t)|$.

The error bound holds when either s or t, or both are covered by some landmarks. When neither s nor t is covered by some landmarks within a radius c, $\mathsf{err}(s,t)$ is unbounded. The probability for this case is $(1 - \mathsf{CR})^2$. Thus we have $P(\mathsf{err}(s,t) \leq 2c) \geq 1 - (1 - \mathsf{CR})^2$. □

Given a user-specified error bound ϵ, we will have $P(\mathsf{err}(s,t) \leq \epsilon) \geq 1 - (1 - \mathsf{CR})^2$, here $c = \epsilon/2$.

When $\mathsf{CR} = 0.8$, the bound is satisfied with a probability $P(\mathsf{err}(s,t) \leq \epsilon) \geq 0.96$.

3.2.3. Graph Partitioning-based Heuristic

For the landmark embedding method we propose above, the offline complexity is $O(|E|+|V|\log|V|)$ to compute the single-source shortest paths for a landmark $v \in \mathcal{R}$. It can be simplified as $O(n\log n)$ ($n = |V|$) when the graph is sparse. Therefore, the total embedding time is $O(|\mathcal{R}|n\log n)$, which could be very expensive when $|\mathcal{R}|$ is large. In this subsection, we propose a graph partitioning-based heuristic for the landmark embedding to reduce the offline time complexity with a relaxed error bound. To distinguish the two methods we propose, we name the first method RN-basic and the partitioning-based method RN-partition.

Partitioning-based Landmark Embedding

The first step of RN-partition is landmark selection, which is the same as described in Chapter 3.2.2. In the second step, we use KMETIS [49] to partition the graph into K clusters $C_1, \ldots, C_K$. As a result, the landmark set $\mathcal{R}$ is partitioned into these K clusters. We use $\mathcal{R}_i$ to denote the set of landmarks assigned to C_i, *i.e.*, $\mathcal{R}_i = \{r | r \in \mathcal{R} \ and \ r \in C_i\}$. It is possible that $\mathcal{R}_i = \emptyset$ for some i. For a cluster C_i with $\mathcal{R}_i = \emptyset$, we can select the vertex from C_i with the largest degree as a within-cluster landmark, to improve the local coverage within C_i. Note that the number of such within-cluster landmarks is bounded by the number of clusters K, which is a small number compared with $|\mathcal{R}|$.

The idea of the partitioning-based landmark embedding is as follows. For the cluster C_i, we compress all landmarks in $\mathcal{R}_i$ as a supernode SN_i and then compute the single-source shortest paths from SN_i to every vertex $v \in V$. The landmark compression operation is defined as follows.

Definition 3.4 (Landmark Compression)**.** The landmark compression operation compresses all landmarks in $\mathcal{R}_i$ into a supernode SN_i. After compression, for a vertex $v \in V \setminus \mathcal{R}_i$, $(\mathsf{SN}_i, v) \in E$ iff $\exists r \in \mathcal{R}_i$, s.t. $(r, v) \in E$, and the edge weight is defined as $w(\mathsf{SN}_i, v) = \min_{r \in \mathcal{R}_i} w(r, v)$.

Then the shortest path between SN_i and v is actually the shortest path between a landmark $r \in \mathcal{R}_i$ and v with the smallest shortest distance, *i.e.*,

$$\mathsf{dist}(\mathsf{SN}_i, v) = \min_{r \in \mathcal{R}_i} \mathsf{dist}(r, v)$$

and we denote the closest landmark $r \in \mathcal{R}_i$ to v as $r_{v,i}$, which is defined as

$$r_{v,i} = \arg\min_{r \in \mathcal{R}_i} \mathsf{dist}(r, v)$$

Note $\mathsf{dist}(\mathsf{SN}_i, v) = \mathsf{dist}(r_{v,i}, v) = \min_{r \in \mathcal{R}_i} \mathsf{dist}(r, v)$. In the following, we will use $\mathsf{dist}(\mathsf{SN}_i, v)$ and $\mathsf{dist}(r_{v,i}, v)$ interchangeably.

The time complexity for computing shortest paths from the supernodes in each of the K clusters to all the other vertices in V is $O(Kn \log n)$. In addition, we compute the shortest distances between every pair of landmarks within the same cluster. The time complexity of this operation is $O(|\mathcal{R}|n/K \log n/K)$, if we assume the nodes are evenly partitioned into K clusters. We further define the diameter d for a cluster as follows.

Definition 3.5 (Cluster Diameter)**.** Given a cluster C, the diameter d is defined as the maximum shortest distance between two landmarks in C, *i.e.*,

$$d = \max_{r_i, r_j \in C} \mathsf{dist}(r_i, r_j)$$

where $\mathsf{dist}(r_i, r_j)$ is the shortest distance between r_i and r_j.

Then the diameter of the partitioning $C_1, \ldots, C_K$ is defined as the maximum of the K cluster diameters, *i.e.*,

$$d_{max} = \max_{i \in [1,K]} d_i$$

Partitioning-based Shortest Distance Estimation

Given a query (s, t), for the supernode SN_i representing a cluster C_i, based on the triangle inequality we have

$$\mathsf{dist}(s, t) \le \mathsf{dist}(s, \mathsf{SN}_i) + \mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(t, \mathsf{SN}_i)$$

Figure 3.2 shows an illustration of the shortest distance estimation between (s, t) in RN-partition, where the circle represents a cluster C_i. Note that in general s and $r_{s,i}$ may not necessarily belong to the same cluster, and the shortest distance $\mathsf{dist}(s, r_{s,i})$ may not necessarily be bounded by the radius c. But these factors will not affect the distance estimation strategy.

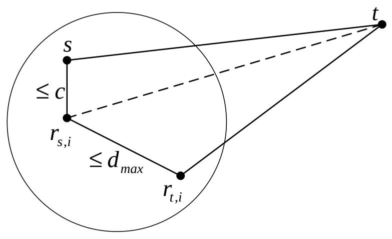


Figure 3.2: Distance Estimation in RN-partition

By considering all K clusters, we have a tighter upper bound

$$\mathsf{dist}(s,t) \leq \min_{i\in[1,K]} (\mathsf{dist}(s,\mathsf{SN}_i) + \mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(t,\mathsf{SN}_i))$$

We denote this estimated distance upper bound as $\widetilde{\mathsf{dist}}^P(s,t)$.

Error Bound Analysis

In the following theorem, we will show that, when s or t is covered within a radius c by some landmark from a cluster C_i for some i, the estimated distance $\widetilde{\mathsf{dist}}^P(s,t)$ is within a bounded error of the true distance $\mathsf{dist}(s,t)$.

Theorem 3.3. *Given any query (s,t), the error of the estimated shortest distance $\widetilde{\mathsf{dist}}^P(s,t)$ by* RN-partition *can be bounded by $2(c + d_{max})$ with a probability no smaller than $1 - (1 - \mathsf{CR})^2$, where c is the coverage radius,* CR *is the cover ratio and d_{max} is the maximum cluster diameter.*

Proof. Given a query (s,t), assume s is covered by at least one landmark from $\mathcal{R}$ within a radius c. Without loss of generality, assume such a landmark is from the cluster C_i for some i and denote it as $r_{s,i}$. According to the triangle inequality, we have

$$\mathsf{dist}(r_{s,i},t) - \mathsf{dist}(s,t) \leq \mathsf{dist}(s,r_{s,i}) \leq c$$

By adding $\mathsf{dist}(s,r_{s,i})$ on both sides, we have

$$\mathsf{dist}(s,r_{s,i}) + \mathsf{dist}(r_{s,i},t) - \mathsf{dist}(s,t) \leq 2\mathsf{dist}(s,r_{s,i}) \leq 2c \tag{3.4}$$

Denote the closest landmark in C_i to t as $r_{t,i}$. Then we have

$$\mathsf{dist}(r_{t,i}, t) - \mathsf{dist}(r_{s,i}, t) \leq \mathsf{dist}(r_{s,i}, r_{t,i}) \leq d_{max}$$

Since $r_{s,i}, r_{t,i}$ belong to the same cluster, their distance is bounded by d_{max}. By adding $\mathsf{dist}(r_{s,i}, r_{t,i})$ on both sides, we have

$$\mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(r_{t,i}, t) - \mathsf{dist}(r_{s,i}, t) \leq 2\mathsf{dist}(r_{s,i}, r_{t,i}) \leq 2d_{max} \tag{3.5}$$

By adding Eq.(3.4) and Eq.(3.5), we have

$$\mathsf{dist}(s, r_{s,i}) + \mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(r_{t,i}, t) - \mathsf{dist}(s, t) \leq 2(c + d_{max})$$

As we have defined $\widetilde{\mathsf{dist}}^{P}(s,t) = \min_{i\in[1,K]}(\mathsf{dist}(s, \mathsf{SN}_i) + \mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(t, \mathsf{SN}_i))$, the error of the estimated shortest distance between (s,t) is bounded by

$$\begin{aligned}\mathsf{err}^P(s,t) &= \widetilde{\mathsf{dist}}^{P}(s,t) - \mathsf{dist}(s,t)\\ &\leq \mathsf{dist}(s, r_{s,i}) + \mathsf{dist}(r_{s,i}, r_{t,i}) + \mathsf{dist}(r_{t,i}, t) - \mathsf{dist}(s,t)\\ &\leq 2(c + d_{max})\end{aligned}$$

The error bound holds when either s or t, or both are covered by some landmarks with a radius c. When it happens that neither s nor t is covered by some landmarks, Eq.(3.4) does not hold in general, thus $\mathsf{err}^P(s,t)$ is unbounded. The probability for this case is $(1 - \mathsf{CR})^2$. For a similar reason as explained in Theorem 3.2, *i.e.*, even when neither s nor t is covered, if there are landmarks $r_{s,i}$, $r_{t,i}$, for some i, on the shortest path from s to t, we can still have an accurate estimation which satisfies the error bound. Therefore the probability that the error of an estimated distance is unbounded is at most $(1 - \mathsf{CR})^2$. Thus, we have $P(\mathsf{err}^P(s,t) \leq 2(c + d_{max})) \geq 1 - (1 - \mathsf{CR})^2$ □

Compared with RN-basic, RN-partition reduces the offline computational complexity to $O(Kn \log n + |\mathcal{R}|n/K \log n/K)$. As long as we choose a reasonably large K such that $|\mathcal{R}|/K \leq K$, the complexity of RN-partition is dominated by $O(Kn \log n)$. As a tradeoff, the error bound is relaxed from $2c$ to $2(c + d_{max})$. The cluster diameter d_{max} is determined by the size of the graph and the number of clusters K. Table 3.1 compares RN-basic and RN-partition on time/space complexity and

Table 3.1: Comparison between RN-basic and RN-partition

	RN-basic	RN-partition
Offline Time Complexity	$O(\|\mathcal{R}\|n \log n)$	$O(Kn \log n + \|\mathcal{R}\|n/K \log n/K)$
Offline Space Complexity	$O(\|\mathcal{R}\|n)$	$O(Kn + \|\mathcal{R}\|^2/K)$
Distance Query Complexity	$O(\|\mathcal{R}\|)$	$O(K)$
Error Bound	$2c$	$2(c + d_{max})$

the error bound. In experimental study, we will study the relationship between K, the offline computation time and the accuracy of the estimated distances.

3.2.4. Experiments

We performed extensive experiments to evaluate our algorithms on two types of networks – a road network and a social network. The road network and the social network exhibit quite different properties on: (1) degree distribution, *i.e.*, the former roughly follows a uniform distribution while the latter follows a power law distribution; and (2) network diameter, *i.e.*, the social network has the shrinking diameter property [56] and the small world phenomenon, which, however, do not hold in the road network. All experiments were performed on a Dell PowerEdge R900 server with four 2.67GHz six-core CPUs and 128GB main memory running Windows Server 2008. All algorithms were implemented in Java.

Comparison Methods and Evaluation

We compare our methods RN-basic and RN-partition with two existing methods:

- **2RNE** [53] by Kriegel et al. uses a two level landmark embedding which examines K nearest landmarks for both nodes in a query to provide a distance estimation. We select landmarks uniformly and set $K = 3$.
- **Centrality** [64] by Potamias et al. selects landmarks with low closeness centrality. According to [64], the approximate centrality measure is computed by

selecting a sample of S random seeds, where we set $S = 10,000$ in our implementation.

As it is expensive to exhaustively evaluate all node pairs in a large network, we randomly sample a set of $10,000$ node pairs in the graph as queries and evaluate the average relative error on the sample set.

Case Study 1: Road Network

We use the New York City road network, which is an undirected planar graph with $264,346$ nodes and $733,846$ edges. A node here represents an intersection or a road endpoint while the weight of an edge represents the length of the corresponding road segment. The data set can be downloaded from http://www.dis.uniroma1.it/∼challenge9/.

The degrees of most nodes in the road network fall into the range of $[1,4]$ and the network has no small world phenomenon. For the $10,000$ random queries we generate, we plot the histogram of the shortest distance distribution in Figure 3.3. The average distance over the $10,000$ queries is $d_{avg} = 26.68$KM. So if we set the radius $c = 0.8$KM, the average relative error can be roughly bounded by $2c/d_{avg} = 0.06$.

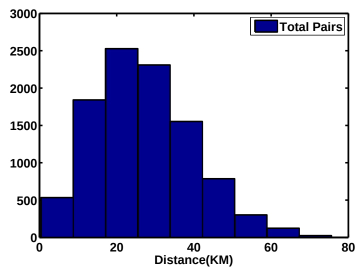


Figure 3.3: Shortest Distance Distribution on Road Network

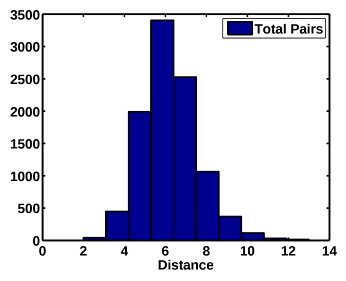


Figure 3.4: Shortest Distance Distribution on Social Network

Parameter Sensitivity Test on CR In this experiment, we vary the cover ratio CR and compare the average error, the landmark set size and offline index time by RN-basic and RN-partition with $K = 100, 250, 500$, respectively. We fix the radius $c = 0.8$KM.

Figure 3.5 shows the average error of RN-basic and RN-partition with different K values. The average error of RN-basic is below 0.01 and slightly decreases as CR increases. The average error of RN-partition decreases very sharply when the number of partitions K increases and becomes very close to that of RN-basic when $K = 500$.

Figure 3.6 shows that the number of landmarks $|\mathcal{R}|$ increases linearly with CR. As RN-basic and RN-partition have the same landmark selection process, the number is the same for both methods. When CR $= 1.0$, we need $9,000$ landmarks to cover the road network with $264,346$ nodes.

Figure 3.7 shows the offline index time in logarithmic scale for RN-basic and RN-partition to compute the single-source shortest paths from every landmark. RN-partition reduces the index time of RN-basic by one order of magnitude. In addition, as the number of landmarks $|\mathcal{R}|$ increases linearly with CR, the index time of RN-basic also increases linearly with CR, because the time complexity is $O(|\mathcal{R}|n \log n)$. On the other hand, the index time of RN-partition remains quite stable as CR increases, because RN-partition only computes the shortest paths from each of the K clusters as the source.

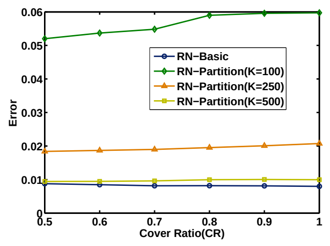


Figure 3.5: Average Error vs. Cover Ratio (CR) on Road Network

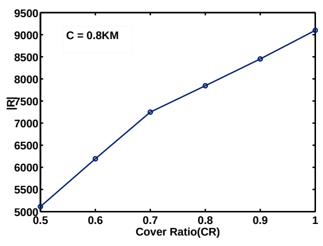


Figure 3.6: Landmark Set Size vs. Cover Ratio (CR) on Road Network

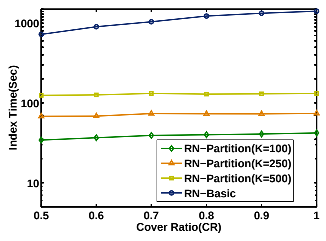


Figure 3.7: Index Time vs. Cover Ratio (CR) on Road Network

Parameter Sensitivity Test on c In this experiment, we vary the radius c and compare the average error, the landmark set size and offline index time by RN-basic and RN-partition with $K = 100, 250, 500$, respectively. We fix the cover ratio CR $= 1.0$.

Figure 3.8 shows the average error of RN-basic and RN-partition with different

K values. We can make the following observations from the figure: (1) RN-partition ($K = 500$) achieves an average error very close to that of RN-basic when $c \geq 0.8$KM; (2) The average error of RN-basic monotonically increases with c, which is consistent with the theoretical error bound of $2c$; and (3) Different from RN-basic, the average error of RN-partition shows a decreasing trend with c. When c is very small, the number of landmarks is very large. So RN-partition may choose suboptimal landmarks for distance estimation, which leads to a larger error.

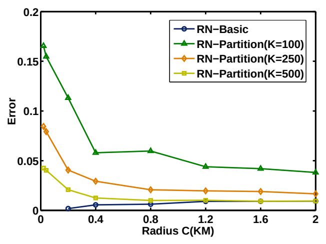


Figure 3.8: Average Error vs. Radius (c) on Road Network

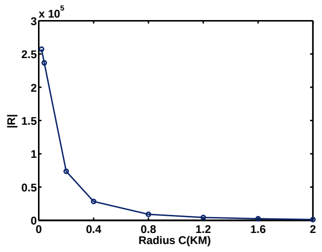


Figure 3.9: Landmark Set Size vs. Radius (c) on Road Network

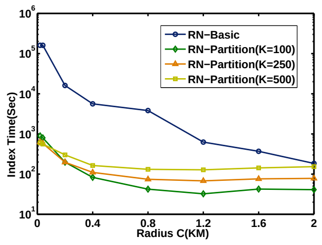


Figure 3.10: Index Time vs. Radius (c) on Road Network

Figure 3.9 shows that the number of landmarks $|\mathcal{R}|$ decreases with c. When $c < 0.4$KM, $|\mathcal{R}|$ decreases sharply with c. Figure 3.10 shows the offline index time of RN-basic and RN-partition in logarithmic scale. As $|\mathcal{R}|$ decreases with c, the index time of RN-basic also decreases with c. RN-partition reduces the index time of RN-basic by two orders of magnitude or more when $c < 0.2$KM but the difference becomes smaller as c increases. RN-basic cannot finish within 10 hours when $c \leq 0.08$KM. On the other hand, the index time of RN-partition increases moderately when c decreases to 0.2KM or below.

Comparison with 2RNE and Centrality We compare our approaches with 2RNE [53] and Centrality [64] in terms of average error, index time and average query time, as we vary the number of landmarks. For our methods, we set CR $= 1.0$. From Figure 3.11 we can see that both RN-basic and RN-partition (for most cases) outperform 2RNE and Centrality by a large margin in terms of average error. Figure 3.12 shows

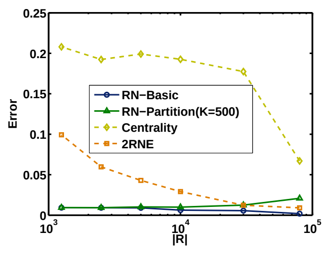


Figure 3.11: Average Error vs. $|\mathcal{R}|$ on Road Network

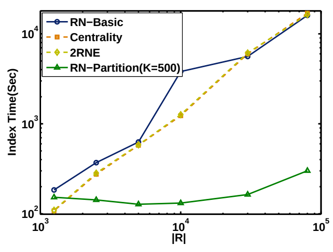


Figure 3.12: Index Time vs. $|\mathcal{R}|$ on Road Network

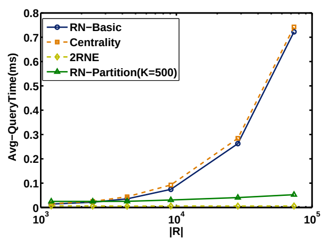


Figure 3.13: Average Query Time vs. $|\mathcal{R}|$ on Road Network

that RN-partition reduces the index time of the other three methods by up to two orders of magnitude. The index time of RN-basic, 2RNE and Centrality increases linearly with $|\mathcal{R}|$, as they all have the same time complexity of $O(|\mathcal{R}|n \log n)$, while RN-partition slightly increases the index time. Figure 3.13 shows that the query time of RN-partition and 2RNE remain almost constant, while that of RN-basic and Centrality increase linearly with the number of landmarks.

Case Study 2: Social Network

We download the DBLP dataset from http://dblp.uni-trier.de/xml/ and construct an undirected coauthor network, where a node represents an author, an edge represents a coauthorship relation between two authors, and all edge weights are set to 1. This graph has several disconnected components and we choose the largest connected one which has $629,143$ nodes and $4,763,500$ edges. The vertex degree distribution follows the power law distribution.

We randomly generate $10,000$ queries and plot the histogram of the shortest distance distribution in Figure 3.4. The average distance between two nodes over the $10,000$ queries is $d_{avg} = 6.34$, which conforms with the famous social networking rule "six degrees of separation". Given $2c/d_{avg}$ as a rough estimation of the relative error bound, if we set $c = 3$, the relative error bound is $2 \times 3/6.34 = 94.64\%$. Therefore, we only test our methods given $c \in \{1, 2\}$, to control the relative error bound in a reasonably small range. Note that $c = 1$ defines the coverage of a node based on the number of its neighbors, *i.e.*, degree; while $c = 2$ measures the coverage based on the number of neighbors within two hops.

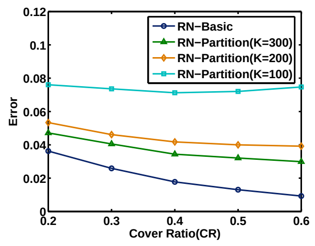


Figure 3.14: Average Error vs. Cover Ratio (CR) on Social Network

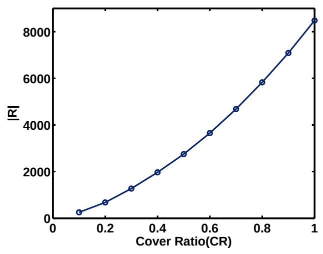


Figure 3.15: Landmark Set Size vs. Cover Ratio (CR) on Social Network

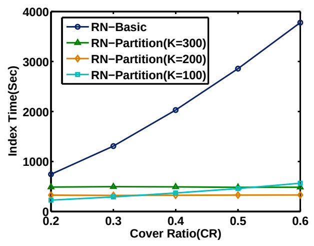


Figure 3.16: Index Time vs. Cover Ratio (CR) on Social Network

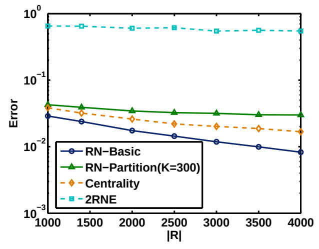


Figure 3.17: Average Error vs. $|\mathcal{R}|$ on Social Network

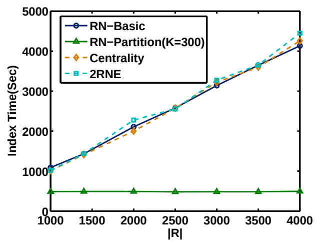


Figure 3.18: Index Time vs. $|\mathcal{R}|$ on Social Network

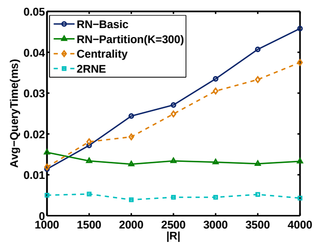


Figure 3.19: Average Query Time vs. $|\mathcal{R}|$ on Social Network

Parameter Sensitivity Test on CR We vary the cover ratio CR and compare the average error, the landmark set size and offline index time by RN-basic and RN-partition with $K = 100, 200, 300$, respectively. We fix the radius $c = 1$.

Figure 3.14 shows that the average error of RN-basic is in the range of $[0.009, 0.04]$ and it decreases quickly as CR increases. The average error of RN-partition is slightly higher than that of RN-basic and it decreases as K increases.

Figure 3.15 shows the number of landmarks $|\mathcal{R}|$ as we vary CR in the range of $[0, 1.0]$. Different from the road network which shows a linear relationship between

Table 3.2: Parameter Sensitivity Test on Radius c on Social Network

		RN-basic		RN-partition	
Radius c	$\|\mathcal{R}\|$	**Average Error**	**Index Time (sec)**	**Average Error**	**Index Time (sec)**
1	3653	0.009	3778.17	0.030	485.88
2	31	0.138	30.70	0.144	65.71

$|\mathcal{R}|$ and CR, we observe that $|\mathcal{R}|$ increases slowly when CR is small, but much faster when CR is large. This is due to the power law degree distribution in the social network – we first select the authors with the largest number of collaborators as landmarks; but in the later stage, with the decrease of node degrees, we need to use more landmarks to achieve the same amount of coverage.

Figure 3.16 shows the offline index time for RN-basic and RN-partition. We observe that the index time of RN-basic increases quickly when CR increases. When CR $= 0.6$, RN-basic is about 10 times slower than RN-partition. We also observe that the index time of RN-partition slightly increases with CR when $K = 100$. This is because a large portion of time is spent on computing the shortest distances between all pairs of landmarks within the same partition. When CR increases, the number of landmarks falling into the same partition is larger, which causes the time increase.

Parameter Sensitivity Test on c In this experiment, we vary the radius $c \in \{1, 2\}$ and compare the average error, the landmark set size and offline index time by RN-basic and RN-partition ($K = 300$). We fix CR $= 0.6$. Table 3.2 shows that the number of landmarks is reduced by 100 times when c is increased to 2. As a result, the offline index time for RN-basic is also reduced by 100 times with the increase of c because the time complexity is $O(|\mathcal{R}|n \log n)$. The index time for RN-partition is seven times smaller than RN-basic when $c = 1$, but slightly higher when $c = 2$ due to the within partition computational overhead. The average error of RN-partition is slightly higher than that of RN-basic, and the error of both methods increases with c, which is consistent with the theoretical error bound.

Comparison with 2RNE and Centrality We compare our approaches with 2RNE and Centrality in terms of average error, index time and average query time, as we vary the number of landmarks. For our methods, we set $c = 1$. Figure 3.17 shows that RN-basic achieves the smallest error, followed by Centrality and RN-partition. 2RNE performs the worst, because it selects landmarks uniformly, rather than selecting landmarks with large degrees. Figure 3.18 shows that the index time of RN-partition remains stable when $|\mathcal{R}|$ increases, while the time of the other three methods increases linearly with $|\mathcal{R}|$. Figure 3.19 shows that the query time of RN-partition and 2RNE remain almost constant, while that of RN-basic and Centrality increase linearly with the number of landmarks.

3.3. Query-Dependent Local Landmark Scheme

3.3.1. Problem Statement

Landmark Selection

In the landmark embedding approach, a key question is how to select the landmark set S from V, as the landmarks can heavily influence the estimation accuracy of shortest distance queries. However, selecting the optimal set of landmarks has been proven to be NP-hard, by a reduction from the classical NP-hard problems such as vertex cover [64] or minimum K-center [26]. Due to the hardness of the landmark selection problem, previous studies (e.g., [26, 64, 68]) proposed various heuristics, including random selection, degree, centrality, and coverage based selection heuristics. But the performance of different heuristics heavily depends on the graph properties, e.g., degree distribution, diameter, etc. There is no heuristic that excels in all kinds of graphs.

Factors on Embedding Performance

Here we briefly discuss the factors that affect the performance of landmark embedding.

A globally selected query-independent landmark set: Most existing methods select a single set of global landmarks which are independent of queries. Such a query-independent landmark set provides a single global view for all possible queries which could be diameter apart or close by, thus it cannot achieve uniformly good performance on all queries. The landmark set can only provide a very rough distance estimation for a query, especially when it is distant from both query nodes, and the two query nodes are close by, as shown in Figure 3.20.

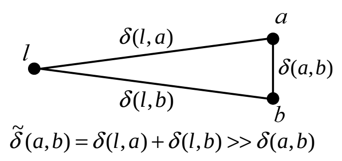

l
a
b
$\delta(l,a)$
$\delta(l,b)$
$\delta(a,b)$
$\tilde{\delta}(a,b) = \delta(l,a) + \delta(l,b) >> \delta(a,b)$

Figure 3.20: Distance Estimation with a Landmark

The number of landmarks k: In general, increasing the number of landmarks k will improve the performance of landmark embedding. An extreme case is $k = |V|$ which leads to zero estimation error. This actually corresponds to computing all pair shortest paths as an embedding. As a side effect, increasing k will cause an increase of the query processing time and the index size, as the query complexity is $O(k)$ and the index space complexity is $O(kn)$. Thus, increasing k is not an efficient or scalable solution to improve the embedding performance.

A Query-Dependent Local Landmark Scheme

In this subsection we propose a novel framework, called *local landmark scheme*, for estimating the shortest distance with a small number of query-dependent local landmarks. In this framework, the problem is formulated as follows.

Problem Statement: Given an arbitrary query node pair (a, b) and a global landmark set S, our goal is to identify a query-specific local landmark which is closer to the true shortest path $P(a, b)$ than any global landmark in a graph G. The approximate shortest distance between a and b is computed as the sum of their distances to the local landmark.

The local landmark can be defined in an abstract way as:

Definition 3.6 (Query-Dependent Local Landmark)**.** Given a global landmark set S and a query (a, b), a query-dependent local landmark function is

$$L_{ab}(S) : V^k \mapsto V$$

which maps S to a vertex in V called a local landmark.

With the local landmark, we can estimate a shortest distance of query (a, b) as

$$\widetilde{\mathsf{dist}}^L(a, b) = \mathsf{dist}(L_{ab}(S), a) + \mathsf{dist}(L_{ab}(S), b) \tag{3.6}$$

Let us see an example.

Example 3.2. Figure 3.21 shows an example graph with the global landmark set $S = \{l_1, l_2, l_3\}$. In this graph, a solid line between two nodes represents an edge of unit length, while a dashed line between two nodes represents a path with zero or more intermediate connecting nodes and thus a length no less than one.

For a pair of query nodes (a, b), the path in bold (a, e, f, g, b) is the shortest path between a and b. The shortest paths from the global landmark l_1 to a and b are $(l_1, \ldots, c, e, a)$ and $(l_1, \ldots, c, d, g, b)$, respectively. Based on l_1, the estimated shortest distance between a and b is

$$\widetilde{\mathsf{dist}}(a, b) = \mathsf{dist}(l_1, a) + \mathsf{dist}(l_1, b) \tag{3.7}$$

But if we have the shortest distances $\mathsf{dist}(c, a)$ and $\mathsf{dist}(c, b)$, we can have a more accurate distance estimation based on c than that based on l_1:

$$\widetilde{\mathsf{dist}}^{L}(a, b) = \mathsf{dist}(c, a) + \mathsf{dist}(c, b) \tag{3.8}$$

as we have $\widetilde{\mathsf{dist}}(a, b) = \widetilde{\mathsf{dist}}^{L}(a, b) + 2\mathsf{dist}(l_1, c)$.

As opposed to the concept of global landmark, we call node c a local landmark with respect to query nodes a, b.

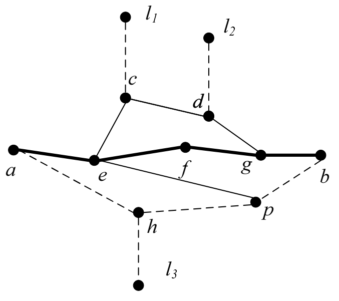


Figure 3.21: Local Landmarks

3.3.2. Shortest Path Tree Based Local Landmark

In landmark embedding, the shortest distances from each global landmark $l \in S$ to all vertices in V are precomputed for the embedding purpose. To preserve more delicate information, we can further consider the *shortest path tree* (SPT) rooted at each global landmark $l \in S$. Here we rephrase the the definition of SPT as follows.

Definition 3.7 (Shortest Path Tree)**.** Given a graph $G = (V, E, w)$, the shortest path tree rooted at a vertex $r \in V$ is a spanning tree of G, such that the path from the root r to each node $v \in V$ is a shortest path between r and v, and the path length is the shortest distance between r and v.

Figure 3.22 shows an SPT rooted at the global landmark l_1 according to the graph in Figure 3.21. An SPT not only contains the shortest distance information from the tree root, it also preserves the more delicate structure information on how the two query nodes are connected. Based on the tree structure, we can identify a node, e.g.,

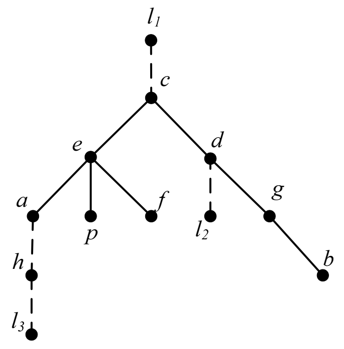


Figure 3.22: Shortest Path Tree Rooted at l_1

c, which is closer to nodes a and b than l_1. Based on this intuition, we propose an SPT based local landmark function.

SPT Based Local Landmark Function

In Example 3.2, we find that the distance estimation based on node c, i.e., $\widetilde{\mathsf{dist}}^L(a,b) = \mathsf{dist}(c,a) + \mathsf{dist}(c,b)$ is a tighter upper bound than that based on l_1. If we look at the SPT rooted at l_1 in Figure 3.22, it is not hard to find that c is the *least common ancestor* (LCA) of a and b.

Definition 3.8 (Least Common Ancestor)**.** Let T be a rooted tree. The least common ancestor of two nodes u and v in T, denoted as $\mathsf{LCA}_T(u,v)$, is the node furthest from the root that is an ancestor of both u and v.

In light of this, we propose an SPT based local landmark function, which returns the LCA of the query nodes a, b in the SPT rooted at each landmark $l \in S$.

Definition 3.9 (SPT Based Local Landmark Function)**.** Given a global landmark set S and a query (a,b), the SPT based local landmark function is defined as:

$$L_{ab}(S) = \arg \min_{r \in \{\mathsf{LCA}_{T_l}(a,b) | l \in S\}} \{\mathsf{dist}(r,a) + \mathsf{dist}(r,b)\}$$

where $\mathsf{LCA}_{T_l}(a,b)$ denotes the least common ancestor of a and b in the SPT T_l rooted at $l \in S$.

We can show that the distance estimation with the SPT based local landmarks is more accurate, or at least the same accurate as that with the global landmark set.

Theorem 3.4. *Given a global landmark set S, $\forall a, b \in V$, we have*

$$\mathsf{dist}(a,b) \leq \widetilde{\mathsf{dist}}^{L}(a,b) \leq \widetilde{\mathsf{dist}}(a,b).$$

Proof. First, $\mathsf{dist}(a,b) \leq \widetilde{\mathsf{dist}}^{L}(a,b)$ holds according to the triangle inequality.

Next, $\forall l \in S$, $r = \mathsf{LCA}_{T_l}(a,b)$, we have

$$\mathsf{dist}(l,a) + \mathsf{dist}(l,b) = \mathsf{dist}(r,a) + \mathsf{dist}(r,b) + 2\mathsf{dist}(r,l)$$

As $\mathsf{dist}(r,l) \geq 0$, we have

$$\mathsf{dist}(r,a) + \mathsf{dist}(r,b) \leq \mathsf{dist}(l,a) + \mathsf{dist}(l,b)$$

Consequently,

$$\begin{aligned}\widetilde{\mathsf{dist}}^{L}(a,b) &= \min_{l \in S}\{\mathsf{dist}(\mathsf{LCA}_{T_l}(a,b),a) + \mathsf{dist}(\mathsf{LCA}_{T_l}(a,b),b)\}\\ &\leq \min_{l \in S}\{\mathsf{dist}(l,a) + \mathsf{dist}(l,b)\}\\ &= \widetilde{\mathsf{dist}}(a,b)\end{aligned}$$

□

To efficiently calculate $\widetilde{\mathsf{dist}}^{L}(a,b)$, $a \neq b \in V$, we have:

$$\begin{aligned}\widetilde{\mathsf{dist}}^{L}(a,b) &= \min_{l \in S}\{\mathsf{dist}(\mathsf{LCA}_{T_l}(a,b),a) + \mathsf{dist}(\mathsf{LCA}_{T_l}(a,b),b)\}\\ &= \min_{l \in S}\{\mathsf{dist}(l,a) + \mathsf{dist}(l,b) - 2\mathsf{dist}(l,\mathsf{LCA}_{T_l}(a,b))\}\end{aligned} \tag{3.9}$$

When $\mathsf{LCA}_{T_l}(a,b)$, $\forall l \in S$ is known, Eq.(3.9) can be computed in $O(|S|)$ time. For each $l \in S$, it simply looks up three embedded distances $\mathsf{dist}(l,a)$, $\mathsf{dist}(l,b)$ and $\mathsf{dist}(l,\mathsf{LCA}_{T_l}(a,b))$.

LCA **Computation**

The techniques in [8] can efficiently find the LCA of two nodes in an SPT T in $O(1)$ time with an $O(n)$ size LCA index, where n is the number of nodes in T.

Complexity Analysis

We analyze the online query complexity and the offline embedding complexity of LLS.

Online Query Time Complexity: The query is based on Eq.(3.9). For each global landmark $l \in S$, there are three lookup operations to retrieve the embedded distances, which take $O(1)$ time. In addition, there is an LCA query which can be answered in $O(1)$ time. For all global landmarks in S, the query time complexity is $O(|S|)$.

Offline Embedding Space Complexity: The space requirement of LLS can be partitioned into three parts: (1) embedded distances from each global landmark to every node in the graph in $O(|S|n)$ space; (2) shortest path trees and the corresponding *trace*, L and *stamp* arrays for all global landmarks in $O(|S|n)$ space; and (3) LCA index tables for all global landmarks in $O(|S|n)$ space. Combining the above three factors, the offline embedding space complexity of LLS is $O(|S|n)$.

Offline Embedding Time Complexity: Given a global landmark set S, the time complexity to compute the single-source shortest paths from a landmark by Dijkstra's algorithm [22] is $O(m + n \log n)$. It can be simplified to $O(n \log n)$ when the graph G is sparse. We also have to build the LCA index in $O(n)$ time for each global landmark in S. Thus the total embedding time complexity is $O(|S|n \log n)$.

When compared with GLS, it is not hard to verify that our LLS has the same online query complexity and offline embedding complexity, although our complexities have slightly larger constant factors.

Extending LLS to Directed Graphs

Our LLS method is mainly designed for undirected graphs. Here we briefly discuss how to extend it to handle directed graphs. For each landmark l, we build two SPTs rooted at l. One is a backward tree TB_l, where the tree path $P_{TB_l}(v, l)$ is the shortest path from v to l for $v \in V$. The other is a forward tree TF_l, where the tree path $P_{TF_l}(l, v)$ is the shortest path from l to v for $v \in V$. Given a query (x, y), for landmark l, we first retrieve two tree paths, $P_{TB_l}(x, l)$ and $P_{TF_l}(l, y)$, then find one of their common nodes with the smallest distance estimation, i.e., $\min_{v \in P_{TB_l}(x,l) \cap P_{TF_l}(l,y)} \{\mathsf{dist}(x, v) + \mathsf{dist}(v, y)\}$, as a local landmark candidate. Finally, we report

$$\widetilde{\mathsf{dist}}(x, y) = \min_{l \in S} \{ \min_{v \in P_{TB_l}(x,l) \cap P_{TF_l}(l,y)} \{\mathsf{dist}(x, v) + \mathsf{dist}(v, y)\}\}$$

as the approximate distance. Using hashing techniques, the query time is $O(\Sigma_{l \in S}(|P_{TB_l}(x, l)| + |P_{TF_l}(l, y)|))$ where $|P_{TB_l}(x, l)|$ and $|P_{TF_l}(l, y)|$ denote the number of hops in the corresponding paths. The embedding uses $O(|S|n)$ space to store the forward and backward SPTs for all landmarks in S.

3.3.3. Optimization Techniques

In this subsection, we propose two additional techniques, *graph compression* and *local search* to further optimize the performance of our local landmark scheme. Graph compression aims to reduce the embedding index size by compressing the graph nodes, and local search performs limited scope online search to improve the distance estimation accuracy.

Index Reduction with Graph Compression

As we have shown, the embedding index takes $O(|S|n)$ space, which is a linear function of the graph node number n. Thus we can effectively reduce the embedding index size if the graph nodes can be compressed. Towards this goal, we propose graph compression techniques which reduce some simple local graph structures with low-degree nodes to a representative node. Our compression techniques are lossless, thus do not sacrifice the distance query accuracy.

Graph Compression and Index Construction

We first define two types of special graph nodes, i.e., *tree node* and *chain node*.

Definition 3.10 (Graph Incident Tree and Tree Node)**.** A tree $T = (V_T, E_T, r)$ with the root r is a graph incident tree on a graph $G = (V, E)$ if (1) $V_T \subseteq V$, $E_T \subseteq E$; and (2) for any path $P(u, v)$ between $u \in V_T$ and $v \in V - V_T$, P must go through the tree root r. A graph incident tree is maximal if it is not contained in another graph incident tree. The nodes $V_T - \{r\}$ are called tree nodes and the root r is the entry node of all tree nodes in T.

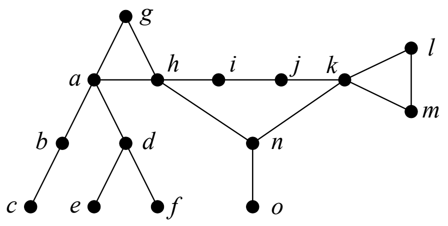


Figure 3.23: Graph Compression Example

For example, in Figure 3.23, the tree with nodes a, b, c, d, e, f is a maximal graph incident tree, where the root a is the *entry node* and nodes b, c, d, e, f are the *tree nodes*. A graph incident tree can be simply discovered by recursively removing graph nodes with degree 1 until the entry node is met (with degree > 1). As the entry node is the only access point for a tree node to connect to the rest of the graph, it is sufficient to keep the entry node as a representative. The graph incident tree is thus compressed to the entry node by removing all the tree nodes. In addition, the distance from each tree node to the entry node is saved in an array. After we remove all tree nodes, we next identify the chain nodes.

Definition 3.11 (Chain Node)**.** Given a graph $G = (V, E)$, a chain node $v \in V$ is a non tree node with $degree(v) = 2$.

If we trace through the two edges incident on a chain node respectively, the two nodes which are first encountered through the chain with a degree greater than 2 are called *end nodes*. The two end nodes may be identical when a cycle exists. For example in Figure 3.23, nodes i and j are *chain nodes* with *end nodes* h and k. We will remove the chain nodes and the incident edges, and then connect the two end nodes with a new edge, whose length is equal to the length of the chain. The distances from a chain node to both end nodes are saved in an array.

After we remove the tree nodes and chain nodes and their incident edges from a graph $G = (V, E)$, we obtain a compressed graph $G' = (V', E')$. Then the index structures including the embedded distances, shortest path trees and LCA index tables are constructed on top of G', instead of G. As $|V'| < |V|$, the index size can be effectively reduced.

One point worth noting is that, as a graph incident tree is compressed to a single entry node, the tree structure is lost. When given two query nodes from the same tree, their LCA cannot be identified from the compressed graph G'. For example, for nodes e, f in Figure 3.23, their LCA d cannot be identified from the compressed graph, as d has been removed as a tree node. In order to handle such queries, we select one global landmark $l \in V$ and build the embedding index including the shortest path tree and LCA index on the *original graph*; and select the other global landmarks from V' and build the index on the *compressed graph*. For any two tree nodes on the same graph incident tree, e.g., e, f, their LCA is the same in all shortest path trees rooted at any global landmarks. Thus it is sufficient to build the full index on the original graph for one global landmark only. The space complexity is $O(n + (|S| - 1)n')$, which is smaller than $O(|S|n)$ on the original graph.

Query Processing on Compressed Graph

Given a query (a, b), if $a, b \in V'$, the local landmark based approximate shortest distance $\widetilde{\mathsf{dist}}^L(a, b)$ can be estimated by Eq.(3.9) in the same way as in the original graph; otherwise, if at least one of a, b is a tree node or chain node and not in V', then

$$\widetilde{\mathsf{dist}}^L(a, b) = \min_{\substack{r_a \in map(a),\\ r_b \in map(b)}} \{\mathsf{dist}(a, r_a) + \widetilde{\mathsf{dist}}^L(r_a, r_b) + \mathsf{dist}(b, r_b)\} \tag{3.10}$$

where $map(a)$ contains the representative nodes for a, defined as follows: (1) if a is a tree node, $map(a)$ contains the corresponding entry node; (2) if a is a chain node, $map(a)$ contains the two end nodes; and (3) if $a \in V'$, $map(a)$ is a itself. The intermediate query $\widetilde{\mathsf{dist}}^L(r_a, r_b)$ can be answered by Eq.(3.9), as $r_a, r_b \in V'$, according to the definition of $map()$.

There are two special cases to be handled separately.

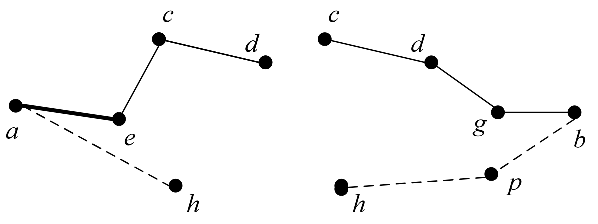
1. If a, b are tree nodes from the *same* graph incident tree, the query $\widetilde{\mathsf{dist}}^L(a, b)$ can be answered with the local landmark from the index on the original graph. For example, for query (e, f) in Figure 3.23, $\widetilde{\mathsf{dist}}^L(e, f) = \mathsf{dist}(e, d) + \mathsf{dist}(f, d)$, where d is the LCA of e and f.

2. If a, b are two chain nodes with the same end nodes, then $\widetilde{\mathsf{dist}}^L(a, b) = \min\{d, |\mathsf{dist}(a, r) - \mathsf{dist}(b, r)|\}$, where d is estimated by Eq.(3.10), and r is either one of the two end nodes. For example, for query (i, j) in Figure 3.23, there are two paths $P_1 = (i, j)$ and $P_2 = (i, h, n, k, j)$ between i and j. The distance $\widetilde{\mathsf{dist}}^L(i, j)$ is estimated by the shorter one among P_1 and P_2.

Our graph compression is lossless. Thus in all the above cases, the estimated distance based on the compressed graph will be the same as that based on the original graph.

Improving Accuracy by Local Search

In this subsection, we propose an online *local search* (LS) technique which performs a limited scope local search on the graph and may find a shortcut with a smaller distance than that based on LLS. Given a query (a, b), for each global landmark $l \in S$, we can find $\mathsf{LCA}_{T_l}(a, b)$ in T_l rooted at l. The shortest path between a query node and a local landmark $\mathsf{LCA}_{T_l}(a, b)$ can also be obtained from the corresponding SPT T_l. If we trace the shortest paths from a to all the LCAs (similarly from b to all the LCAs), we can form two *partial shortest path trees* rooted at a and b respectively, e.g., T_a and T_b in Figure 3.24 following Example 3.2. A leaf node in such trees must be an LCA; while it is also possible an LCA is an intermediate node, if it lies on the shortest path from a query node to another LCA, e.g., the intermediate node c in T_a in Figure 3.24 (a).



(a) T_a (b) T_b

Figure 3.24: Partial Shortest Path Trees

The local search expands a partial shortest path tree T by a width of c, i.e., for each node in T, its neighbors within c hops in the graph are included in the expanded tree T^c. For the two expanded trees T_a^c and T_b^c rooted at the query nodes, the common nodes of T_a^c and T_b^c act as bridges to connect the two query nodes. We will find a path connecting the two query nodes through a bridge with the smallest distance. If the distance is smaller than the estimation $\widetilde{\mathsf{dist}}^L(a, b)$ by LLS, we will report this local search distance as a more accurate estimation for the query (a, b). Figure 3.25 shows the 1-hop expanded trees T_a^1 and T_b^1, where the 1-hop neighbors of every tree node in T_a and T_b are included. The yellow shade illustrates (in an abstract way) that each node in the dashed path is also expanded to include its 1-hop neighbor. Based on the expanded trees there are four paths connecting a and b, i.e., (a, e, c, d, g, b), (a, e, f, g, b),

Algorithm 3.1: Local Search

```
Input: A query $(a, b)$ and the expansion width $c$.
Output: The shortest distance of a path.
1  $\mathsf{LCA} \leftarrow \{\mathsf{LCA}_{T_l}(a, b) | l \in S\}$;
2  $T_a \leftarrow \mathsf{partial\text{-}SPT}(a, \mathsf{LCA})$;
3  $T_b \leftarrow \mathsf{partial\text{-}SPT}(b, \mathsf{LCA})$;
4  $T_a^c \leftarrow \mathsf{Tree\text{-}Expansion}(T_a, c)$;
5  $T_b^c \leftarrow \mathsf{Tree\text{-}Expansion}(T_b, c)$;
6  $\mathsf{dist} \leftarrow \infty$;
7  for $r \in T_a^c \cap T_b^c$ do
8      if $\mathsf{dist}_{T_a^c}(a, r) + \mathsf{dist}_{T_b^c}(b, r) < \mathsf{dist}$ then
9          $\mathsf{dist} \leftarrow \mathsf{dist}_{T_a^c}(a, r) + \mathsf{dist}_{T_b^c}(b, r)$;
10 return dist;
```

$(a, e, p, \ldots, b)$ and $(a, \ldots, h, \ldots, p, \ldots, b)$. As (a, e, f, g, b) has the shortest distance between a and b, we return the distance as the answer.

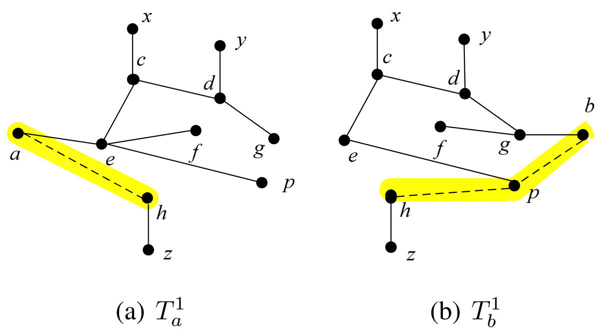


Figure 3.25: 1-Hop Expanded Trees T_a^1 and T_b^1

Algorithm 3.1 shows the pseudocode of the local search. Lines 2-3 build two partial shortest path trees rooted at a and b respectively to all the local landmarks. Lines 4-5 expand the two trees to include the neighbors within c hops for each tree node. $\mathsf{dist}_{T_a^c}(a, r)$ in line 8 represents the path length from a to r in the expanded tree T_a^c.

The setting of the search width c is a trade-off between the accuracy and the online query cost. If we set c to 2 or above, for graphs with high average degree, e.g., social networks, the online search space will explode and the query time will become too long. On the other hand, if we set $c = 0$ without edge expansion, it will be faster but much less accurate than the $c = 1$ case (note that LS with $c = 0$ is more accurate than LLS or the same, since it searches a shortcut between two query nodes by connecting two partial trees directly). Thus, in our experiment, we set $c = 1$ to achieve a good balance.

Comparing Local Search with TreeSketch: TreeSketch [32] is a sketch-based method for shortest distance/path estimation. It also uses online search to improve the accuracy. The main differences between TreeSketch and LS include different search order and stop condition. In TreeSketch, the sketch of node s denoted as T_s is a tree rooted at s. For a query $q = (s, d)$, TreeSketch performs a bi-directional search on T_s and T_d, and the expansion follows a breadth-first search order on each side. Let V_{BFS} and V_{RBFS} denote the sets of visited nodes from two sides respectively. Consider $u \in T_s$ and $v \in T_d$ that are two nodes under expansion in the current iteration. TreeSketch checks if there is an edge from u to a node in V_{RBFS} or from a node in V_{BFS} to v. If yes, then an s-d path is found and added to a queue Q. Denote the length of the shortest path in Q as l_{min}, the algorithm terminates if $\mathsf{dist}(s, u) + \mathsf{dist}(v, d) \geq l_{min}$. This early stop condition may miss a better shortcut and thus return a less optimal answer. Figure 3.26 is an example. l_1 and l_2 are two landmarks. For query (s, d), all the solid arrows are edges in $T_s \cup T_d$ while the dashed arrows are not. The solid curves are paths with one or more edges in T_s or T_d. During the search process, when b is visited, $V_{BFS} = \{j, a, f\}$ and $V_{RBFS} = \{e, c, b\}$, TreeSketch finds the first shortcut $p = p(s, a) \circ (a, b) \circ p(b, d) = (s, a, b, c, e, d)$, and updates $l_{min} = 5$. TreeSketch finds $\mathsf{dist}(s, f) + \mathsf{dist}(b, d) = 5 = l_{min}$ and then terminates the search. In contrast, LS can find a better shortcut $p^* = (s, j, f, h, d)$ with length 4 by local search, but p^* is missed by TreeSketch due to its early stop mechanism.

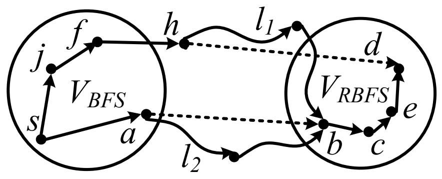


Figure 3.26: Comparing Local Search with TreeSketch

3.3.4. Local Landmark Scheme on Relational Database

Although the proposed memory based LLS and LS methods can estimate an approximate shortest distance between two nodes efficiently, the index size in $O(|S|n)$ increases linearly with the graph size. For very large graphs, the index may become too large to fit in the memory. This limitation motivates us to consider a scalable disk-based index. In this subsection, we propose to build a disk-based index on relational database (RDB) due to its powerful indexing and query optimization mechanisms. In the following, we will study how to design a database schema to store the index and how to use RDB features to optimize the query performance.

LLS on Relational Database

We first study how to implement the local landmark scheme on relational database, denoted as $\mathsf{LLS}_{\mathsf{db}}$. To distinguish, the memory based LLS described before is denoted as $\mathsf{LLS}_{\mathsf{mem}}$.

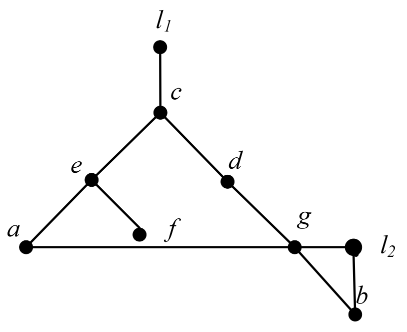
Database Schema For $\mathsf{LLS}_{\mathsf{db}}$

Recall the local landmark scheme estimates a shortest distance of a query (a, b) as $\widetilde{\mathsf{dist}}^{L}(a, b) = \mathsf{dist}(L_{ab}(S), a) + \mathsf{dist}(L_{ab}(S), b)$ in Eq.(3.6). If we store $\mathsf{dist}(L_{ab}(S), a)$ and $\mathsf{dist}(L_{ab}(S), b)$ in a relational table, they can be retrieved to answer a shortest distance query. Thus we create a table, called $\mathsf{Tbl}_{\mathsf{D}}$, with $Tbl_D\_schema = (s, t, d)$ where $s, t \in V$ are two nodes and $d = \mathsf{dist}(s, t)$ is the true shortest distance from s to t. (s, t) is designated as the primary key to support efficient selection and join operations on them.

We populate $\mathsf{Tbl}_{\mathsf{D}}$ to build a local landmark embedding as follows. On an SPT rooted at a global landmark l, consider a path $P(v,l) = (v, v_1, v_2, \ldots, v_t, l)$ for any $v \in V$, we calculate $\mathsf{dist}(v, v_i) = \mathsf{dist}(v, l) - \mathsf{dist}(v_i, l)$ and insert the tuple $\langle v, v_i, \mathsf{dist}(v, v_i)\rangle$ into $\mathsf{Tbl_D}$, for $i = 1, 2, \ldots, t$. Algorithm 3.2 shows how to populate $\mathsf{Tbl_D}$ using SPTs and the shortest distance $\mathsf{dist}(v,l)$, $\forall v \in V$ and $\forall l \in S$.

Algorithm 3.2: Constructing $\mathsf{Tbl_D}$

```
Input: SPTs for landmark set $S$ and $\mathsf{dist}(v,l), v \in V, l \in S$.
1 for $\forall v \in V, l \in S$ do
2     Obtain $P(v,l) = (v, v_1, v_2, \ldots, v_t, l)$ from SPT $T_l$;
3     for $i = 1, \ldots, t$ do
4         $\mathsf{dist}(v, v_i) \leftarrow \mathsf{dist}(v,l) - \mathsf{dist}(v_i, l)$;
5         Insert tuple $\langle v, v_i, \mathsf{dist}(v, v_i)\rangle$ into $\mathsf{Tbl_D}$;
```



s	t	d
a	e	1
a	c	2
a	l_1	3
a	g	1
a	l_2	2
b	g	1
b	d	2
b	c	3
b	l_1	4
b	l_2	1

Figure 3.27: Example for Building $\mathsf{Tbl_D}$

Example 3.3. Figure 3.27 shows a graph with two global landmarks l_1 and l_2. Based on the SPTs rooted at l_1 and l_2 respectively, we can build $\mathsf{Tbl_D}$ on the right, which only lists tuples with attribute $s = a$ or $s = b$ here.

$\mathsf{LLS_{db}}$ **Query** For $\mathsf{LLS_{db}}$, we express the shortest distance query in relational algebra as follows.

$$\begin{aligned}\widetilde{\mathsf{dist}}_{\mathsf{LLS_{db}}}(x, y) \quad &\leftarrow \quad \mathcal{G}_{\min(t_1.d+t_2.d)}(\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \\ &\qquad \bowtie_{t_1.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D}))) \end{aligned} \tag{3.11}$$

which is composed of a join operation and an aggregation operation. Firstly, it selects two groups of tuples where $s = x$ and $s = y$ and renames them as t_1 and t_2 respectively, then it joins t_1 and t_2 using the condition $t_1.t = t_2.t$. Such $t_1.t$ is a local landmark. In $\mathsf{LLS_{db}}$, any node can be a local landmark, as long as it satisfies the condition $t_1.t = t_2.t$. This is different from $\mathsf{LLS_{mem}}$ which restricts the local landmarks as LCAs of two query nodes. Finally, the aggregation operator $\mathcal{G}_{\min}$ finds the minimum $t_1.d + t_2.d$ over all joined tuples.

Accuracy Analysis $\mathsf{LLS_{db}}$ is more accurate, or at least the same accurate as $\mathsf{LLS_{mem}}$, i.e.,

$$\widetilde{\mathsf{dist}}_{\mathsf{LLS_{db}}}(x, y) \leq \widetilde{\mathsf{dist}}_{\mathsf{LLS_{mem}}}(x, y)$$

The proof can be sketched as follows. $\mathsf{LLS_{mem}}$ uses LCAs on SPTs as local landmarks. For any global landmark $l \in S$, we denote $\mathsf{LCA}_{T_l}(x, y)$ as r. Obviously, tuples $\langle x, r, \mathsf{dist}(x, r)\rangle$ and $\langle y, r, \mathsf{dist}(y, r)\rangle$ are in $\mathsf{Tbl_D}$ and they satisfy the $t_1.t = t_2.t$ condition. Thus the LCAs are a subset of local landmarks considered in $\mathsf{LLS_{db}}$. Under the aggregation operator $\mathcal{G}_{\min}$, we prove $\widetilde{\mathsf{dist}}_{\mathsf{LLS_{db}}}(x, y) \leq \widetilde{\mathsf{dist}}_{\mathsf{LLS_{mem}}}(x, y)$. Example 3.4 shows one such case.

Example 3.4. In Figure 3.27, there are two global landmarks l_1 and l_2. Given a query (a, b), $\mathsf{LCA}_{T_{l_1}}(a, b) = c$, and $\mathsf{LCA}_{T_{l_2}}(a, b) = l_2$. Therefore, by $\mathsf{LLS_{mem}}$ we have

$$\widetilde{\mathsf{dist}}_{\mathsf{LLS_{mem}}}(a, b) = \min\{\mathsf{dist}(a, c) + \mathsf{dist}(b, c), \mathsf{dist}(a, l_2) + \mathsf{dist}(b, l_2)\} = 3$$

In comparison, in $\mathsf{LLS_{db}}$, we have two tuples in $\mathsf{Tbl_D}$, $\langle a, g, 1\rangle$ and $\langle b, g, 1\rangle$, by joining which we have a more accurate estimation as

$$\widetilde{\mathsf{dist}}_{\mathsf{LLS_{db}}}(a, b) = 1 + 1 < 3 = \widetilde{\mathsf{dist}}_{\mathsf{LLS_{mem}}}(a, b)$$

g is a local landmark providing a shortcut between a, b.

For the memory based LS, if we set $c = 0$, it will report the same estimated distance as $\mathsf{LLS_{db}}$, since both of them essentially find shortcuts between two query nodes by joining two partial trees without edge expansion. From this perspective, it is not surprising that $\mathsf{LLS_{db}}$ achieves a better accuracy than $\mathsf{LLS_{mem}}$.

Local Search on Relational Database

We now study how to implement local search on relational database, denoted as $\mathsf{LS_{db}}$.

Database Schema For $\mathsf{LS_{db}}$ We create another table $\mathsf{Tbl_G}$ which stores the graph edges E and serves for online expansion. We define the schema $Tbl_G\_schema = (s, t, d)$ where $e(s, t) \in E$ represents an edge and $d = w(s, t)$ is the edge weight. (s, t) is designated as the primary key to support efficient selection and join operations on them. For a graph $G(V, E, w)$, we insert all edges in E and a self-loop for each node in V to $\mathsf{Tbl_G}$:

$$\mathsf{Tbl_G} \leftarrow \{\langle u, v, w(u, v)\rangle | (u, v) \in E\} \cup \{\langle v, v, 0\rangle | v \in V\}$$

The purpose of adding self-loops will be made clear shortly.

$\mathsf{LS_{db}}$ **Query** We define two slightly different local search queries: $\mathsf{unidirectional\text{-}expansion}$, denoted as $\mathsf{LS_{dbu}}$, and $\mathsf{bidirectional\text{-}expansion}$, denoted as $\mathsf{LS_{dbb}}$. We first express $\mathsf{LS_{dbu}}$ in relational algebra as follows.

$$\begin{aligned}\widetilde{\mathsf{dist}}_{\mathsf{LS_{dbu}}}(x, y) \quad \leftarrow \quad & \mathcal{G}_{\min(t_1.d+\mathsf{Tbl_G}.d+t_2.d)}(\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \\ & \bowtie_{t_1.t=\mathsf{Tbl_G}.s} \mathsf{Tbl_G} \\ & \bowtie_{\mathsf{Tbl_G}.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D}))) \qquad (3.12)\end{aligned}$$

It is composed of two join operations. The tuples selected by $\sigma_{s=x}(\mathsf{Tbl_D})$ correspond to a set of shortest paths originating from x to some graph nodes $t_1.t$. The first join operator expands these shortest paths from x by one edge with the condition $t_1.t = \mathsf{Tbl_G}.s$, and the second join operator connects the expanded paths with the shortest paths originating from y with the condition $\mathsf{Tbl_G}.t = t_2.t$. Conversely if we perform the second join operation before the first join, this corresponds to expanding the shortest paths originating from y and then connecting them with the shortest paths from x. The final result will be the same regardless of the join order. Here we do not explicitly specify a join order and leave it to DBMS. The aggregation operator $\mathcal{G}_{\min}$ finds the minimum $t_1.d + \mathsf{Tbl}_\mathsf{G}.d + t_2.d$ over all joined tuples.

Note that if t_1 has a tuple $\langle x, v, \mathsf{dist}(x, v)\rangle$ and t_2 has a tuple $\langle y, v, \mathsf{dist}(y, v)\rangle$, $v \in V$, then LS_dbu will join these two tuples through an intermediate tuple $\langle v, v, 0\rangle$ corresponding to a self-loop on v and generate a distance of $\mathsf{dist}(x, v) + 0 + \mathsf{dist}(y, v)$. This actually performs no edge expansion and is the same as LLS_db. Thus the purpose of including self-loops in Tbl_G is to allow both edge expansion and no expansion, with the hope to generate more accurate distance estimations.

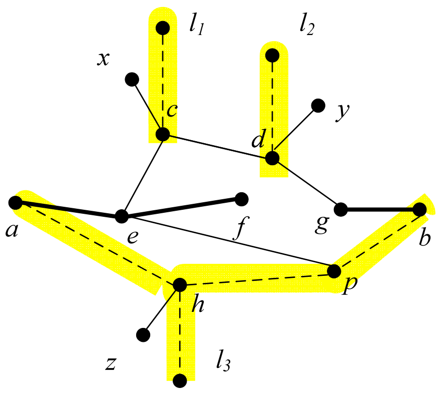
Next we define the bidirectional-expansion LS_dbb in relational algebra as follows.

$$\begin{aligned}\widetilde{\mathsf{dist}}_{\mathsf{LS}_\mathsf{dbb}}(x, y) \quad \leftarrow \quad & \mathcal{G}_{\min(t1.d+g1.d+g2.d+t2.d)} \\ & (\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl}_\mathsf{D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl}_\mathsf{G}) \\ & \bowtie_{g_1.t=g_2.s} \rho_{g_2}(\mathsf{Tbl}_\mathsf{G}) \\ & \bowtie_{g_2.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl}_\mathsf{D}))) \end{aligned} \tag{3.13}$$

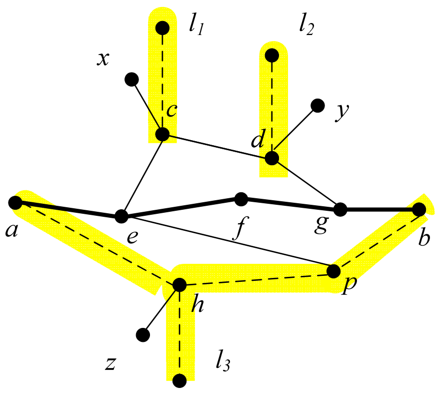
LS_dbb expands the shortest paths from both x and y by one edge respectively through the first and third join operators, and connects the expanded paths by the second join operator. It is similar to the memory based local search with the search width $c = 1$. We do not consider further edge expansions, as that would lead to an explosive number of joined tuples.

Here we use one example to illustrate the local search process unidirectional-expansion and bidirectional-expansion.

Example 3.5. LS_dbu performs unidirectional-expansion and evaluates all possible paths between a and b in Figure 3.28(a). LS_dbb, on the other hand, performs bidirectional-expansion and evaluates all possible paths in Figure 3.28(b). The yellow shade in figures illustrates (in an abstract way) that each node in the dashed path is also expanded to include its 1-hop neighbor. In this example, LS_dbb can find the true shortest path $P(a, b) = (a, e, f, g, b)$. The search space of LS_dbu is a subset of that of LS_dbb,



(a) $\mathsf{LS_{dbu}}$



(b) $\mathsf{LS_{dbb}}$

Figure 3.28: unidirectional-expansion and bidirectional-expansion

thus $\mathsf{LS_{dbu}}$ is less accurate than $\mathsf{LS_{dbb}}$ but with shorter response time.

$\mathsf{LS_{dbb}}$ contains three join operations. The built-in query optimization in RDB may generate a query plan by specifying a join order. It may reduce the intermediate joined results only, but cannot reduce the total number of final joined tuples before aggregation. The final joined results can be huge, thus cause an unacceptable query response time. Let us see an example as follows.

Example 3.6. Figure 3.29 shows three tables t_1, g_1 and $t_1 \bowtie_{t_1.t=g_1.s} g_1$. t_1 contains shortest paths between x and nodes $a_1, a_2, \ldots, a_p$, each of which can be joined with a tuple in g_1 leading to the same node m. Thus the join result contains p tuples denoting paths between x and m with different distances $t_1.d + g_1.d$ (the distance column $t_1.d + g_1.d$ is omitted in Figure 3.29 due to lack of space). Similarly, Figure 3.30 shows tables g_2, t_2 and $g_2 \bowtie_{g_2.t=t_2.t} t_2$. The join result contains q tuples denoting paths between m and y with different distances $g_2.d + t_2.d$ (the distance column $g_2.d + t_2.d$ is omitted in Figure 3.30 due to lack of space).

When we join these two intermediate results using the condition $g_1.t = g_2.s$, we totally get $p \times q$ tuples denoting different paths between x and y. By aggregation on $\min(t_1.d + g_1.d + g_2.d + t_2.d)$, only one tuple with the smallest distance will be returned. Thus all but one of these tuples are a waste of effort. We should try to reduce the intermediate result size before the final join.

s	t	d
x	a_1	d_1
x	a_2	d_2
...	...	...
x	a_p	d_p

t_1

s	t	d
a_1	m	d'_1
a_2	m	d'_2
...	...	...
a_p	m	d'_p

g_1

$t_1.s$	$t_1.t$	$g_1.t$
x	a_1	m
x	a_2	m
...	...	...
x	a_p	m

$t_1 \bowtie_{t_1.t=g_1.s} g_1$

Figure 3.29: Table t_1 Joins g_1

s	t	d
m	b_1	e_1
m	b_2	e_2
...	...	...
m	b_q	e_q

g_2

s	t	d
y	b_1	e'_1
y	b_2	e'_2
...	...	...
y	b_q	e'_q

t_2

$g_2.s$	$g_2.t$	$t_2.s$
m	b_1	y
m	b_2	y
...	...	...
m	b_q	y

$g_2 \bowtie_{g_2.t=t_2.t} t_2$

Figure 3.30: Table g_2 Joins t_2

Optimization of $\mathsf{LS_{dbb}}$: With careful analysis, we optimize $\mathsf{LS_{dbb}}$ as follows:

$$\begin{aligned}\widetilde{\mathsf{dist}}_{\mathsf{LS_{dbb}}}(x,y) \;\leftarrow\; & \mathcal{G}_{\min(d1+t2.d+g2.d)} \\ & ((_{g_1.t}\mathcal{G}_{\min(t_1.d+g_1.d)\ \textbf{as}\ d1} \\ & (\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl_G}))) \\ & \bowtie_{g_1.t=g_2.s} (\rho_{g_2}(\mathsf{Tbl_G}) \\ & \bowtie_{g_2.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D})))) \end{aligned} \tag{3.14}$$

The major optimization is the early evaluation on $\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl_G})$ by the inner aggregation operation $_{g_1.t}\mathcal{G}_{\min(t_1.d+g_1.d)}$. The aggregation operation groups the joined tuples by their destination $g_1.t$, and for each distinct $g_1.t$, the tuples with non-minimum distance on $t_1.d + g_1.d$ are eliminated, as these tuples cannot lead to a path with the minimum distance, when further joining with g_2 and t_2. In Figure

3.29, this means we only keep one tuple $\langle x, a_i, m, d_i + d'_i \rangle$ for m in the joined table where $d_i + d'_i$ is the minimum among all tuples. This tuple will join with tuples in table $g_2 \bowtie_{g_2.t=t_2.t} t_2$ in Figure 3.30 and produce only q joined tuples, instead of $p \times q$. Finally the outer aggregation operation will return the tuple with the minimum distance on $(t_1.d + g_1.d + g_2.d + t_2.d)$. This optimization greatly boosts the query efficiency by 5 to 70 times in our experiment.

Interestingly, we do not apply the same inner aggregation on the join results of $g_2 \bowtie_{g_2.t=t_2.t} t_2$. Actually we have tested that possibility and it turns out to be slower than the current version. The reason is when one side is aggregated, the join becomes an injection; then aggregating the other side will not reduce the number of tuples generated/evaluated but increase the overhead of inner aggregation evaluation.

Accuracy Analysis In terms of the distance estimation accuracy, we have

$$\widetilde{\mathsf{dist}}_{\mathsf{LS_{dbb}}}(x,y) \leq \widetilde{\mathsf{dist}}_{\mathsf{LS_{dbu}}}(x,y) \leq \widetilde{\mathsf{dist}}_{\mathsf{LLS_{db}}}(x,y)$$

The proof can be sketched as follows. $\mathsf{LS_{dbu}}$ can subsume $\mathsf{LLS_{db}}$ by joining two tuples $\langle x, v, \mathsf{dist}(x,v)\rangle$, $\langle y, v, \mathsf{dist}(y,v)\rangle$ in $\mathsf{Tbl_D}$, for an arbitrary $v \in V$, through a self-loop tuple $\langle v, v, 0\rangle$, which generates an estimated distance $\mathsf{dist}(x,v) + 0 + \mathsf{dist}(y,v)$. Similarly, $\mathsf{LS_{dbb}}$ subsumes $\mathsf{LLS_{db}}$ through joining the above two tuples with a self-loop twice, i.e., joining $\langle x, v, \mathsf{dist}(x,v)\rangle$, $\langle v, v, 0\rangle$, $\langle v, v, 0\rangle$ with $\langle y, v, \mathsf{dist}(y,v)\rangle$. $\mathsf{LS_{dbb}}$ subsumes $\mathsf{LS_{dbu}}$ through joining with a self-loop and a graph edge, i.e., joining $\langle x, v, \mathsf{dist}(x,v)\rangle$, $\langle v, v, 0\rangle$, $\langle v, u, \mathsf{dist}(v,u)\rangle$, with $\langle y, u, \mathsf{dist}(y,u)\rangle$. Based on the aggregation operator $\mathcal{G}_{\min}$, we prove the result. But in terms of query complexity, $\mathsf{LS_{dbb}}$ is the highest, and $\mathsf{LLS_{db}}$ is the lowest. The three RDB approaches $\mathsf{LLS_{db}}$, $\mathsf{LS_{dbu}}$ and $\mathsf{LS_{dbb}}$ are trade-offs between query time and accuracy. A user can choose one method based on his needs.

Table 3.3: Network Statistics

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V'\rvert$	$\lvert E'\rvert$	$\overline{\mathsf{dist}}$
Slashdot	77,360	905,468	36,012	752,478	4.1146
Google	875,713	5,105,039	449,341	4,621,002	7.4607
Youtube	1,157,827	4,945,382	313,479	4,082,046	5.3317
Flickr	1,846,198	22,613,981	493,525	18,470,294	5.5439
NYRN	264,346	733,846	164,843	532,264	27km
USARN	23,947,347	58,333,344	7,911,536	24,882,476	1522km

3.3.5. Experiment

We compare our query-dependent local landmark scheme with global landmark embedding. We present extensive experimental results in terms of accuracy, query efficiency and index size on six large networks. All algorithms were implemented in C++ and tested on a Windows server using one 2.67 GHz CPU and 128 GB memory.

Dataset Description

We use four social networks or webgraphs: Slashdot[1], Google Webgraph[2], Youtube [60], and Flickr [60], and two road networks: NYRN[3] and USARN[3]. Table 3.3 lists the network statistics. $|V|$ and $|E|$ represent the node and edge numbers in the original graph, while $|V'|$ and $|E'|$ represent the numbers in the compressed graph. As we can see, our proposed graph compression technique effectively reduces the node number by $38\% - 73\%$. Our embedding index is constructed on the compressed graph. We also sample $10,000$ node pairs in each network and show the average shortest distance $\overline{\mathsf{dist}}$.

[1]http://snap.stanford.edu/data/soc-Slashdot0902.html
[2]http://snap.stanford.edu/data/web-Google.html
[3]http://www.dis.uniroma1.it/~challenge9/download.shtml

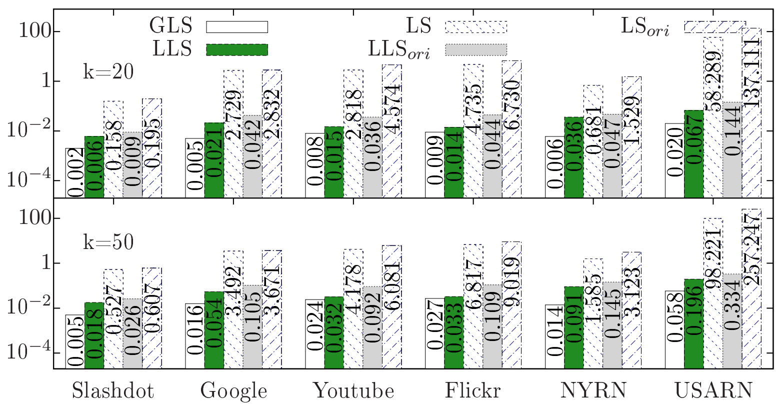


Figure 3.31: Online Query Time in Milliseconds

Comparison Methods and Metrics

We compare the following embedding methods: (1) Global Landmark Scheme (GLS), (2) Local Landmark Scheme (LLS) and (3) Local Search (LS) with $c = 1$. For global landmark selection, we use random selection and closeness centrality based selection [64]. We use two landmark set sizes $k = 20$ and $k = 50$ in our experiments.

We use the relative error

$$\frac{|\widetilde{\mathsf{dist}}(s,t) - \mathsf{dist}(s,t)|}{\mathsf{dist}(s,t)}$$

to evaluate the quality of an estimated distance for a query (s, t). As it is expensive to exhaustively test all node pairs in a large network, we randomly sample $10,000$ node pairs in each graph as queries and compute the average relative error on the sample set. In addition, we test the query processing time and the embedding index size.

In the following, we first compare the memory based implementations of different methods, and then the relational database based implementations.

Memory-based Implementations

Average Relative Error Table 3.4 shows the average relative error (AvgErr) of GLS, LLS and LS with different global landmark sets selected by Random and Centrality.

Table 3.4: Average Relative Error

		$k = 20$					
		SlashD	Google	Youtube	Flickr	NYRN	USARN
Random	GLS	0.6309	0.5072	0.6346	0.5131	0.1825	0.1121
	LLS	0.1423	0.0321	0.0637	0.0814	0.0246	0.0786
	LS	0	0.0046	0.0009	0.0001	0.0071	0.009
Centrality	GLS	0.152	0.0426	0.0595	0.0567	0.6458	1.5599
	LLS	0.1043	0.029	0.0489	0.0503	0.1536	0.4708
	LS	0.0001	0.0074	0.001	0.0003	0.1479	0.4703
		$k = 50$					
		SlashD	Google	Youtube	Flickr	NYRN	USARN
Random	GLS	0.4535	0.475	0.4549	0.4559	0.1188	0.0632
	LLS	0.0727	0.0142	0.0391	0.0444	0.0103	0.0241
	LS	0	0.0022	0.0003	0.0001	0.0042	0.003
Centrality	GLS	0.1385	0.0245	0.0461	0.0524	0.6133	0.7422
	LLS	0.0663	0.014	0.0334	0.0284	0.1533	0.4505
	LS	0	0.0037	0.0005	0	0.1455	0.4483

LLS reduces the AvgErr of GLS by a large margin in all cases. Under Random landmark selection strategy, the AvgErr of LLS is one order of magnitude smaller than that of GLS on most graphs; while under Centrality, LLS reduces the AvgErr by 40% compared with GLS on average. Furthermore, in most cases the AvgErr of LLS with $k = 20$ landmarks is even lower than that of GLS with $k = 50$ landmarks, and at the same time, LLS ($k = 20$) has a smaller embedding index size than GLS ($k = 50$) (see the GLS and LLS bars in Figure 3.3.5). This result demonstrates that selecting more global landmarks for GLS (e.g., $k = 50$) and using more index space do not necessarily achieve a better estimation accuracy than LLS ($k = 20$). Thus simply selecting more landmarks for GLS may not be an effective solution, as the main bottleneck of GLS is caused by the query-independent landmark embedding.

LS achieves the best performance in all cases. Its AvgErr is between 0 and the scale of 10^{-3} in most cases. In particular, in social networks the average distance is usually very small, according to the famous rule of "six degrees of separation". Thus by a local search with 1-hop expansion, the expanded trees rooted at both query nodes are very likely to intersect, which helps to find a very short path, or even the shortest

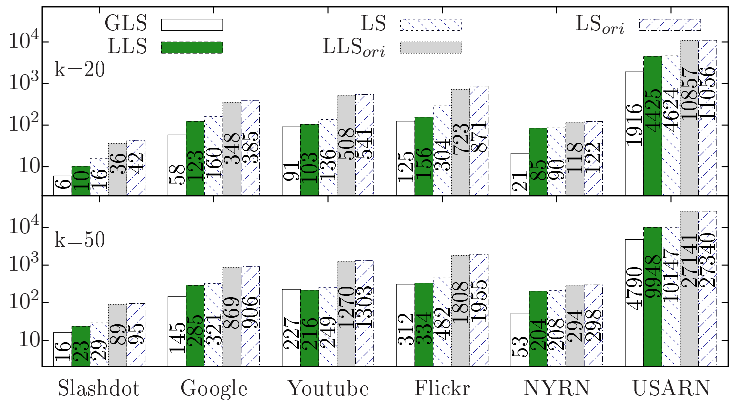


Figure 3.32: Index Size in MB

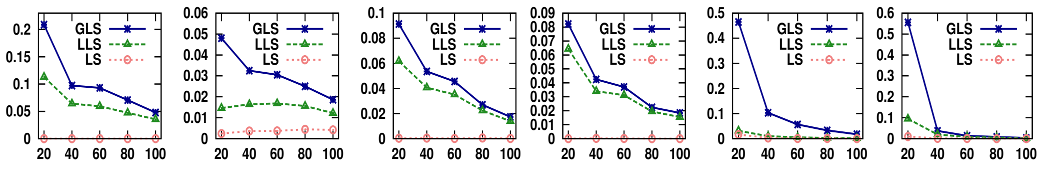
path, between the query nodes.

One point worth noting is that, the state-of-the-art techniques for computing shortest paths and shortest distances on road networks have achieved controlled error rate and low complexities, with the aid of coordinates [77, 75]. As LLS and LS are designed for general graphs, the performance improvement by our methods is more significant on social networks than on road networks.

Online Query Time Figure 3.3.5 shows the query time in milliseconds of different methods in log scale – GLS, LLS and LS on the compressed graph, LLS$_{ori}$ and LS$_{ori}$ on the original uncompressed graph. According to our analysis in Chapter 3.3.2, the online query complexity is $O(|S|)$ for both GLS and LLS. Even in the largest graph USARN with 24 million nodes, it only costs 0.196 milliseconds for LLS to process one query when $k = 50$. In most cases, LLS is $2 - 4$ times slower than GLS, which is a very small factor.

As LS performs online tree expansion and search in query processing, the query time largely depends on the network size. For example, it costs 0.158 milliseconds to process one query in Slashdot, but 58 milliseconds in USARN when $k = 20$, as the node number of USARN is about 310 times larger than that of Slashdot.

We can also observe that LLS reduces the query time of LLS$_{ori}$ by 23% – 70%. The main reason is that graph compression reduces the index size, thus increases the



(a) Slashdot (b) Google (c) Youtube (d) Flickr (e) NYRN (f) USARN

Figure 3.33: Average Relative Error on Queries with Shortest Distance in Different Ranges

locality of memory access and reduces the amortized time per memory access. Similarly, LS reduces the query time of LS$_{ori}$ by 32% on average, as the graph compression prevents LS from expanding the partial tree to tree/chain nodes unnecessarily.

Index Size Figure 3.3.5 shows the index size in MB of different methods in log scale – GLS, LLS and LS on the compressed graph, LLS$_{ori}$ and LS$_{ori}$ on the original uncompressed graph. The index size of LLS is about 2 times that of GLS, as LLS needs to store extra information including shortest path trees and LCA index tables. LS uses a little extra space compared with LLS, as LS needs to store the original graph in memory for edge expansion and local search. Nevertheless, we can see that our LLS and LS methods use moderate index sizes even for very large networks, e.g., an index of 10 GB (when $k = 50$) on USARN with 24 million nodes. We can also observe that the index size of LLS and LS is significantly smaller than that of LLS$_{ori}$ and LS$_{ori}$, and it is reduced by 63% on average, which is consistent with the graph compression ratio in Table 3.3.

In terms of the index construction time of LLS, it takes less than 1 minute in most cases, while the longest one takes 354.7 seconds on USARN when $k = 50$.

Distance Sensitive Relative Error

We evaluate the performance of GLS, LLS and LS on queries in different shortest distance ranges. For each network, we sort the $10,000$ sample queries in the increasing order of their actual shortest distances and find the 20th, 40th, 60th, 80th and 100th percentiles of the shortest distance. Based on this, we partition the $10,000$ queries into 5 intervals, each containing 20 percent of the queries. We evaluate the AvgErr in

Figure 3.33 for queries whose shortest distances fall into the five intervals respectively. For each network we adopt the best landmark selection heuristic, i.e., Centrality for social networks and Random for road networks. We set $k = 50$.

We observe that LLS outperforms GLS in all distance ranges on all networks. The improvement is most significant for queries whose shortest distances are within the 20th percentile. This demonstrates that LLS can provide very accurate estimation for nearby query nodes while GLS cannot. In particular, we observe that on the two road networks, the improvement of LLS over GLS within the 20th percentile is about 10 times or more, which is very substantial. This is because the road networks have large diameters. If two query nodes are nearby but distant from global landmarks, GLS will provide a very inaccurate distance estimation. In contrast, social networks usually have a fairly small diameter, which guarantees that the global landmarks will not be too far from the query nodes. Therefore, on the social networks, the improvement by LLS is around 2 times within the 20th percentile.

The performance of LS remains very stable in all distance ranges. The AvgErr of LS is zero or close to zero on most networks. We also observe that, on dense networks with large average degrees, e.g., Slashdot and Flickr, local search with neighbor expansion is particularly effective in finding the (nearly) shortest paths. The AvgErr of LS on Slashdot is zero in all distance ranges.

Relational Database based Implementations

In this experiment, we study the performance of local landmark scheme implemented on relational database. Specifically we compare $\mathsf{LLS}_{\mathsf{db}}$, $\mathsf{LS}_{\mathsf{dbu}}$ and $\mathsf{LS}_{\mathsf{dbb}}$ and report relative error, query time, index construction time and index size. 20 randomly selected global landmarks are used. We use Oracle Database 11g, Edition release 11.2.0.1.0 and connect to it through ODBC interface. The disk quota for Oracle system is 100 GB. We find in our experiments that the index size for USARN exceeds the 100 GB disk quota, so we only report results on the other five networks.

Average Relative Error

Figure 3.34 shows the average relative error of $\mathsf{LLS_{db}}$, $\mathsf{LS_{dbu}}$ and $\mathsf{LS_{dbb}}$ on five networks. $\mathsf{LS_{dbb}}$ has the most accurate estimation. It has zero error on SlashDot and Youtube, and average error at 10^{-4}–10^{-3} scale on the other networks. The online bidirectional search in $\mathsf{LS_{dbb}}$ reduces the error of $\mathsf{LLS_{db}}$ by 93% on social networks on average. $\mathsf{LS_{dbb}}$ also outperforms $\mathsf{LS_{dbu}}$. The reduction of relative error is the most remarkable on SlashDot, Google, and Flickr, i.e., 100%, 76%, and 99%, respectively. In addition, $\mathsf{LS_{dbu}}$ has the second best performance. It outperforms $\mathsf{LLS_{db}}$ with a reduction of relative error by 76% on social networks on average. Note that the error on NYRN is the same for all three methods. This shows that on a network with a large diameter, local search within 1 or 2-hop neighborhood does not help find a shortcut. But on social networks with much smaller diameters, local search can reduce the relative error substantially.

In Chapter 3.3.4 we have proved that $\mathsf{LLS_{db}}$ has better precision than $\mathsf{LLS_{mem}}$. When we compare the average error of $\mathsf{LLS_{db}}$ and $\mathsf{LLS_{mem}}$ (in Table 3.4, under 20 random global landmarks), we find $\mathsf{LLS_{db}}$ reduces the relative error by 31% on average. In particular, on NYRN dataset, the average error of $\mathsf{LLS_{db}}$ is only $\frac{1}{3}$ that of $\mathsf{LLS_{mem}}$. This result verifies that $\mathsf{LLS_{db}}$ is more accurate than $\mathsf{LLS_{mem}}$.

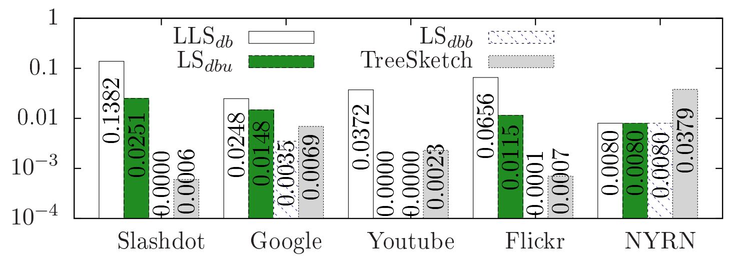


Figure 3.34: Average Relative Error of RDB Algorithms

Online Query Time We report the query time of the three approaches in Figure 3.35 in milliseconds. $\mathsf{LLS_{db}}$ uses the least query time. The query time of $\mathsf{LS_{dbu}}$ is 6.67 times that of $\mathsf{LLS_{db}}$ on average. As a disk-based solution with fairly good precision, $\mathsf{LS_{dbu}}$ only uses 13.5 ms query time on average on the five networks, which is very

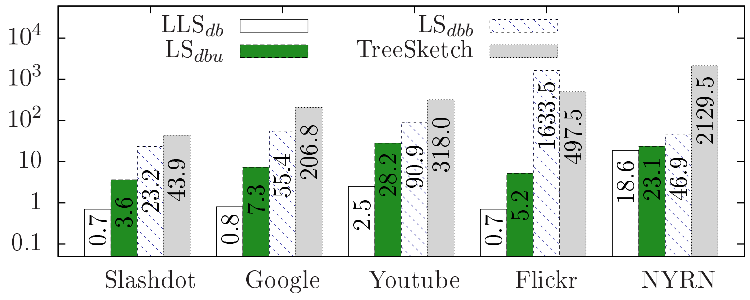


Figure 3.35: Online Query Time (ms) of RDB Algorithms

cost effective. As for $\mathsf{LS_{dbb}}$, it spends 8 times longer query time than $\mathsf{LS_{dbu}}$ on most networks. Still it spends less than 90.9 ms on 4 out of 5 graphs for query processing, which is also quite small for a disk-based algorithm.

Index Size Figure 3.36 reports the index size on the disk in MB. $\mathsf{LS_{dbu}}$ and $\mathsf{LS_{dbb}}$ use the same disk space since both of them use tables $\mathsf{Tbl_D}$ and $\mathsf{Tbl_G}$, thus we use $\mathsf{LS_{db}}$ to represent both of them. $\mathsf{LS_{db}}$ uses slightly more space than $\mathsf{LLS_{db}}$ since it has to store the $\mathsf{Tbl_G}$ table. The index size of all social networks are within 1 GB in most cases. However, the index size of NYRN is as large as 8.9 GB although it is a small network with only 264K nodes. This is because the diameter of NYRN is large while that of social networks is a small value (less than 8 on the networks in our experiment). The index time of $\mathsf{LLS_{db}}$ and $\mathsf{LS_{db}}$ shows a similar trend with their index size, as almost all the index time is spent on database insertion operations on $\mathsf{Tbl_D}$ and $\mathsf{Tbl_G}$.

Comparison with TreeSketch

In this experiment, we compare $\mathsf{LS_{dbb}}$ with TreeSketch [32], a sketch-based method implemented in RDF graph database provided by the authors. Figure 3.34 shows that $\mathsf{LS_{dbb}}$ reduces the average error of TreeSketch by a large margin. $\mathsf{LS_{dbb}}$ has zero error on SlashDot and Youtube. On the other three networks, the average error of $\mathsf{LS_{dbb}}$ is 2–7 times smaller than that of TreeSketch. Figure 3.35 shows that the query time of $\mathsf{LS_{dbb}}$ is 2–45 times shorter than that of TreeSketch on four out of five graphs. Furthermore, Figure 3.36 shows the index time and index size of both $\mathsf{LS_{dbb}}$ (denoted as $\mathsf{LS_{db}}$) and TreeSketch, from which we can see $\mathsf{LS_{dbb}}$ uses much smaller index size

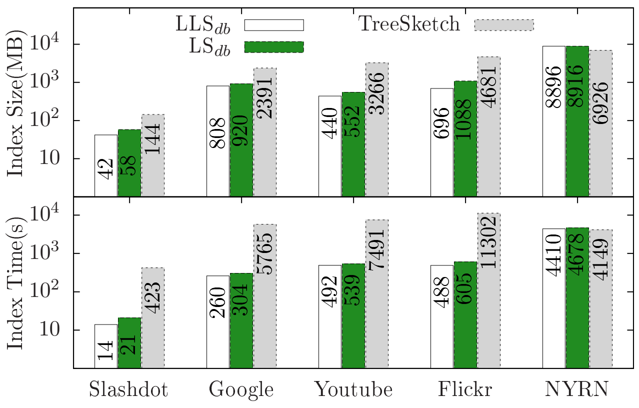


Figure 3.36: Index Size and Index Time of RDB Algorithms

and shorter index time on most graphs than TreeSketch. One reason for the smaller index size is that we apply our graph compression technique in LS$_{dbb}$, but TreeSketch does not have the compression. We also find the implementation of TreeSketch incurs an unnecessary index overhead due to the way RDF3x constructs all possible index configurations which are not really needed for the shortest path estimation. In conclusion, LS$_{dbb}$ outperforms TreeSketch in all aspects.

3.4. Summary

In this chapter, we devise two landmark embedding schemes for a shortest distance query, an *error bounded landmark scheme* and a *local landmark scheme*. The former scheme can guarantee an error bound for an estimated distance. Besides, a partition based method is also proposed which can reduce the offline embedding time by two orders of magnitude with a slightly increase of estimation error. The latter scheme can significantly improve the distance estimation accuracy without increasing the offline embedding or the online query complexity. In addition, it is adapted to the relational database for better scalability. Extensive experimental results on large-scale social networks and road networks demonstrate the effectiveness and efficiency of the proposed landmark schemes.

CHAPTER 4

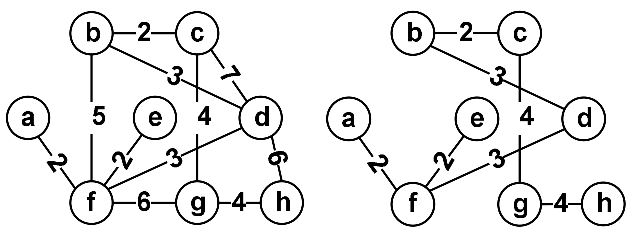
QUERYING WEIGHT CONSTRAINT REACHABILITY

4.1. Problem Definition

4.1.1. Edge Weight Constraint

We first consider an edge-weighted undirected graph $G = (V, E, \Sigma, w)$, where V is the set of vertices, E is the set of edges, $\Sigma \subset \mathbb{R}$ is the set of real-value edge weights and $w : E \mapsto \Sigma$ is a function that assigns each edge $e \in E$ to a real-value weight $w(e) \in \Sigma$. A path P between vertices u and v is denoted as $P(u, v) = (u, v_1, \ldots, v_{l-1}, v)$, where $\{u, v_1, \ldots, v_{l-1}, v\} \subseteq V$ and $\{(u, v_1), \ldots, (v_{l-1}, v)\} \subseteq E$. We say e belongs to P, denoted as $e \in P$, if e is an edge on P. The edge weight constraint reachability query is defined as follows.

Definition 4.1 (Edge Weight Constraint Reachability (EWCR))**.** Given a graph $G(V, E, \Sigma, w)$, an EWCR query is in the form of $q = (a, b, c)$, where $a, b \in V$ and c is a range constraint on edge weight, e.g., $\geq x$, $\leq y$, or within $[x, y]$. q asks whether there is a path $P(a, b)$ between vertices a and b such that $\forall e \in P(a, b)$, $w(e)$ satisfies the constraint c, e.g., $w(e) \geq x$, or $w(e) \leq y$, or $w(e) \in [x, y]$, where $x, y \in \mathbb{R}$. If yes, we say a and b are reachable w.r.t. the edge weight constraint c.



(a) $G(V, E, \Sigma, w)$ (b) The MST T

Figure 4.1: An Example Graph G and Its MST T

Example 4.1. We use a running example throughout this chapter. An undirected graph G is shown in Figure 4.1(a), where each edge has a real-value weight. To answer an EWCR query $q = (a, g, \leq 4)$, we can find a path $P(a, g) = (a, f, d, b, c, g)$ in G such that $\forall e \in P(a, g)$, $w(e) \leq 4$. Thus vertices a and g are reachable w.r.t. the constraint.

4.1.2. Node Weight Constraint

The WCR query also applies to node-weighted graphs, denoted as $G(V, E, \Sigma, w_n)$, where $w_n : V \mapsto \Sigma$ is a function that assigns each node $v \in V$ to a real-value weight $w_n(v) \in \Sigma$. The node weight constraint reachability query is defined as follows.

Definition 4.2 (Node Weight Constraint Reachability (NWCR))**.** Given a graph $G(V, E, \Sigma, w_n)$, an NWCR query is in the form of $q = (a, b, c)$, where $a, b \in V$ and c is a range constraint on node weight, e.g., $\geq x$, $\leq y$, or within $[x, y]$. q asks whether there is a path $P(a, b)$ between vertices a and b such that $\forall v \in P(a, b)$, $w_n(v)$ satisfies the constraint c, e.g., $w_n(v) \geq x$, or $w_n(v) \leq y$, or $w_n(v) \in [x, y]$, where $x, y \in \mathbb{R}$. If yes, we say a and b are reachable w.r.t. the node weight constraint c.

Reducing NWCR **to** EWCR**:** An NWCR query on a node-weighted graph $G(V, E, \Sigma, w_n)$ can be reduced to an EWCR query, if we transform G to an edge-weighted graph $G'(V', E', \Sigma, w')$ as follows. For each edge in G, $e(a, b) \in E$, we create two weighted edges in E', i.e., $e(a, v_{ab}) \in E'$ and $e(v_{ab}, b) \in E'$ with $w'(e(a, v_{ab})) = w_n(a)$ and $w'(e(v_{ab}, b)) = w_n(b)$. Here v_{ab} is a dummy node introduced in V'. This transformation incurs a small constant-factor overhead on the edge set as $|E'| = 2|E|$. Then an NWCR query on G can be equivalently answered by an EWCR query on G'. Hence, we mainly focus on the EWCR query on edge-weighted graphs. For simplicity, in the following we use WCR to denote the edge weight constraint reachability problem.

More generally, our problem setting is not limited to real-value edge or node weights, actually, it can be extended to any form of edge/node labels as long as they have a *total order*, for example, string labels with a total lexicographic order, and multidimensional features on edges/nodes given a function which maps a multidimensional feature to a *total ordered* one-dimensional value, e.g., Hilbert curve [55]. Such graphs are commonplace in various scientific areas including bioinformatics and cheminformatics.

4.1.3. Two Basic Solutions

There are two basic approaches to answer the WCR query. One is online DFS or BFS search using the weight constraint to confine the search space. Starting from vertex a, we follow DFS or BFS order to recursively visit all adjacent vertices through edges whose weights satisfy the constraint, until we reach vertex b or have searched all reachable vertices without reaching b. The query time is $O(|V| + |E|)$, which is impractical for online query processing in a realtime system.

The other approach is to pre-compute the weight constraint reachability for all pairs $a, b \in V$. The query time is $O(1)$, but the space complexity is $O(|V|^2)$ for the constraint $\geq x$ or $\leq y$, or $O(|\Sigma||V|^2)$ for the constraint $[x, y]$, which severely limits the scalability.

The above two solutions are two extremes in terms of the query time complexity and index space complexity. In the following we will design novel index structures for efficient WCR query processing in large-scale networks.

4.2. An Efficient Memory Algorithm

The user-specified edge weight constraint c can have various forms, such as a half-bounded interval, e.g., $\geq x$, $\leq y$, or a bounded interval, e.g., within $[x, y]$. It is not hard to see $\geq x$ and $\leq y$ are symmetric, thus without loss of generality we assume the constraint c has the form of $\leq y$. We will show later that our proposed algorithms can be easily extended to handle the bounded interval constraint $[x, y]$. We first show a property in WCR for query processing.

4.2.1. Properties of WCR

Given a graph G and a WCR query $q(a, b, \leq y)$, for a path $P(a, b)$ in G, we denote the maximum edge weight on $P(a, b)$ as $\overline{P}(a, b) = \max_{e \in P(a,b)} w(e)$. If for *every* path $P(a, b)$ in G, $\overline{P}(a, b) > y$ holds, then we can conclude that a, b are not reachable w.r.t. the constraint $\leq y$. But it is too expensive to enumerate all possible paths $P(a, b)$ in G and check whether $\overline{P}(a, b) > y$ holds or not.

Let us consider the cut property [20] of minimum spanning tree (MST) of a graph, which states that *for any cut C in the graph, if the weight of an edge $e \in C$ is smaller than the weights of any other edges in C, then this edge e belongs to all* MST*s of the graph.* Let T denote an MST of G. For a vertex pair (a, b), there is a unique tree path $P_T(a, b)$ between a and b on T. Let $\overline{P_T}(a, b) = \max_{e \in P_T(a,b)} w(e)$ denote the maximum edge weight on $P_T(a, b)$. Based on the cut property, we can derive the following lemma.

Lemma 4.1. *Given a graph $G(V, E, \Sigma, w)$, its* MST *T and an arbitrary vertex pair (a, b), for any $P(a, b)$ in G, $\overline{P_T}(a, b) \leq \overline{P}(a, b)$ holds, i.e., $\overline{P_T}(a, b) \leq \min_{P(a,b)}\{\overline{P}(a, b)\}$.*

Proof. For an arbitrary vertex pair (a, b), an a-b cut C_{ab} is a set of edges, the removal of which causes a and b to be in two disjoint components of G. Given an arbitrary path $P(a, b)$ in G and an arbitrary a-b cut C_{ab}, the intersection of $P(a, b)$ and C_{ab} must be non-empty, i.e., $P(a,b) \cap C_{ab} \neq \emptyset$. Denote the minimum weight of a cut C_{ab} as $\underline{C_{ab}} = \min\{w(e)|e \in C_{ab}\}$. Then for any $e' \in P(a,b) \cap C_{ab}$, we have

$$\underline{C_{ab}} \leq w(e') \leq \overline{P}(a,b). \tag{4.1}$$

Let e_{max} denote the edge in $P_T(a,b)$ with the largest weight, i.e., $e_{max} = \arg\max_{e \in P_T(a,b)} w(e)$. We can divide T into two disjoint components T_a and T_b by removing e_{max}. Here T_a and T_b represent the connected components in T containing a and b respectively. Then we obtain an a-b cut C^*_{ab} of G, $C^*_{ab} = \{e(u,v) \in E|u \in T_a, v \in T_b\}$ and $e_{max} \in C^*_{ab}$. Based on the cut property of MST, e_{max} must be the edge with the smallest weight in C^*_{ab}, i.e.,

$$w(e_{max}) = \underline{C^*_{ab}}, \tag{4.2}$$

as e_{max} belongs to T and no other edges in C^*_{ab} belong to T.
Combining (4.1) and (4.2), we prove

$$\overline{P_T}(a,b) = w(e_{max}) = \underline{C^*_{ab}} \leq \overline{P}(a,b)$$

for any path $P(a,b)$ in G. □

From Lemma 4.1 we can derive the following theorem to answer a WCR query.

Theorem 4.1. *Two vertices a and b are reachable w.r.t. the weight constraint $\leq y$ in a graph $G \Leftrightarrow \overline{P_T}(a,b) \leq y$ where T is the* MST *of G.*

Proof. 1. $\Rightarrow$: As a and b are reachable w.r.t. the weight constraint $\leq y$ in the graph G through a path, denoted as $P(a,b)$, we have $\overline{P}(a,b) \leq y$ holds. By Lemma 4.1, we have $\overline{P_T}(a,b) \leq \overline{P}(a,b) \leq y$.

2. $\Leftarrow$: As $\overline{P_T}(a,b) \leq y$, we know $w(e) \leq y$ for each edge $e \in P_T(a,b)$. Thus, a and b are reachable w.r.t. the weight constraint $\leq y$ through path $P_T(a,b)$ in G. □

Based on Theorem 4.1, a WCR query $q = (a, b, \leq y)$ can be processed as follows. We find the unique MST path $P_T(a,b)$ between a and b on T and compute the largest edge weight $\overline{P_T}(a, b)$. a and b are reachable w.r.t. the weight constraint if and only if $\overline{P_T}(a, b) \leq y$.

Example 4.2. For graph G in Figure 4.1(a), the MST T is shown in Figure 4.1(b). To answer a WCR query $q = (a, g, \leq 4)$, we find the unique tree path $P_T(a, g) = (a, f, d, b, c, g)$, such that $\forall e \in P_T(a, g)$, $w(e) \leq 4$. Thus vertices a and g are reachable w.r.t. the constraint.

An MST can be built in $O(|E|)$ time using Kruskal's algorithm [54] (with the union-find technique [20]). All edges in E can be sorted beforehand in $O(|E|)$ time using radix sort, since the edge weights are from a finite set. A straightforward approach takes $O(|V|)$ time to find the path $P_T(a, b)$ on T to answer a WCR query. Theorem 4.1 serves as a building block for constructing an efficient index to answer a WCR query.

4.2.2. Novel Edge Based Indexing

In this subsection, we aim to design a novel index which can answer a WCR query in $O(1)$ time. According to Theorem 4.1, it is equivalent to solving the following problem: *Given an* MST T*, compute* $\overline{P_T}(a, b)$ *for any two vertices* a *and* b *in* T *in* $O(1)$ *time.*

Let e_{max} be the edge with the maximum edge weight w_{max} in T. If there are more than one edge with the same maximum weight w_{max}, we pick one of them arbitrarily. We have the following observations.

1. After removing e_{max}, T becomes two disjoint subtrees T_1 and T_2.
2. For any vertices a in T_1 and b in T_2, $\overline{P_T}(a, b) = w_{max}$.
3. For any two vertices a and b in T_1 (or in T_2), $\overline{P_T}(a, b)$ can be similarly determined in T_1 (or in T_2).

Based on the above observations, for an MST T, we define its edge-based index tree as follows.

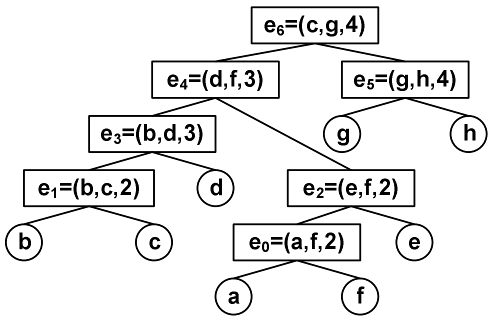


Figure 4.2: The Edge-Based Index Tree $\mathcal{T}$ of MST T

Definition 4.3 (Edge-based Index Tree)**.** For an MST T, its edge-based index tree, denoted as $\mathcal{T}[T]$, is a vertex labeled binary tree, such that: If T contains only one vertex, $\mathcal{T}[T]$ is a tree with a single vertex labeled 0. Otherwise, let e_{max} be an edge with the maximum edge weight w_{max} in T. Suppose after removing e_{max}, T becomes two disjoint subtrees T_1 and T_2. $\mathcal{T}[T]$ is recursively defined as follows.

- The root of $\mathcal{T}[T]$ is labeled w_{max}.
- The left subtree of $\mathcal{T}[T]$ is $\mathcal{T}[T_1]$.
- The right subtree of $\mathcal{T}[T]$ is $\mathcal{T}[T_2]$.

For any MST T, from the definition of $\mathcal{T}[T]$, each node $v \in V(T)$ corresponds to a unique leaf node in $\mathcal{T}[T]$ with a label 0, and each edge $e \in E(T)$ corresponds to a unique internal node in $\mathcal{T}[T]$ with a label $w(e)$. We use $L(v)$ to denote the label of any node v in $\mathcal{T}[T]$. If the context is obvious, we use $\mathcal{T}$ to denote $\mathcal{T}[T]$.

Example 4.3. Figure 4.2 shows the edge-based index tree $\mathcal{T}$ for the MST T shown in Figure 4.1(b). A leaf node in circle corresponds to a vertex $v \in V(T)$ and the letter in the circle denotes the vertex id v. The label of each leaf node in $\mathcal{T}$ is 0 and is not shown in the figure. An internal node with a triple (u, v, w) in rectangle corresponds to an edge $e(u, v) \in E(T)$ with $w = w(e(u, v))$. The label of an internal node in $\mathcal{T}$ is w. $\mathcal{T}$ organizes all 7 edges in T hierarchically.

Essentially $\mathcal{T}$ is a delicate reorganization of all edges in MST T, and it supports computing $\overline{P_T}(a,b)$ in $O(1)$ time with little space overhead. $\mathcal{T}$ is stored in the memory as our index named Edge-Index. Next, we discuss query processing using Edge-Index, followed by the construction algorithm of Edge-Index.

Query Processing: The edge-based index tree $\mathcal{T}$ has the following property: for an internal node $v \in V(\mathcal{T})$, its label $L(v) \leq L(r)$ where $r \in V(\mathcal{T})$ is any ancestor of v in $\mathcal{T}$. We have the following lemma.

Lemma 4.2. *Given an* MST T *of a graph* $G(V, E, \Sigma, w)$ *and its edge-based index tree* $\mathcal{T}$*,* $\forall a, b \in V$*, we denote the lowest common ancestor of* a *and* b *in* $\mathcal{T}$ *as* $\mathsf{LCA}_{\mathcal{T}}(a,b)$*. Then we have*

$$\overline{P_T}(a,b) = L(\mathsf{LCA}_{\mathcal{T}}(a,b))$$

Proof. Let us consider the process to construct the edge-based index tree $\mathcal{T}$: We remove edges from the MST T one by one in the decreasing order of the edge weight. After removing a certain edge, the subtree that contains the edge becomes two disjoint subtrees and the removed edge becomes the root of a subtree in $\mathcal{T}$.

1. Let e be the first edge, whose removal separates a and b in two disjoint subtrees of T. It means e is the first node created in $\mathcal{T}$ that separates a and b, or in other words, $e = \mathsf{LCA}_{\mathcal{T}}(a,b)$.

2. Let us consider the time before removing e and all edges with weights $> w(e)$ are removed. At that time, a and b are still in the same connected subtree in T, and e has the largest weight in the remaining edges. It means $\overline{P_T}(a,b) \leq w(e)$.

3. After removing e in T, a and b are disconnected, which means e is in $P_T(a,b)$. Thus $\overline{P_T}(a,b) \geq w(e)$.

From items 1–3, we can prove

$$\overline{P_T}(a,b) = w(e) = L(\mathsf{LCA}_{\mathcal{T}}(a,b)).$$

□

Based on Lemma 4.2, the WCR query $q = (a, b, \leq y)$ can be processed as follows. On the edge-based index tree $\mathcal{T}$, we find the lowest common ancestor $\mathsf{LCA}_{\mathcal{T}}(a, b)$ of a and b. If $\overline{P_T}(a, b) = L(\mathsf{LCA}_{\mathcal{T}}(a, b)) \leq y$, then a and b are reachable w.r.t. the constraint; otherwise, they are not. The LCA query on an index tree $\mathcal{T}$ can be answered in $O(1)$ time with an $O(|V|)$ size index. [8] provides the technical details of the indexing and query processing, according to which, the $O(|V|)$ size index can be constructed in $O(|V|)$ time.

Example 4.4. With the edge-based index tree $\mathcal{T}$ in Figure 4.2, to answer the WCR query $q = (a, g, \leq 4)$, one has to find $\mathsf{LCA}_{\mathcal{T}}(a, g)$, which is $e_6 = (c, g, 4)$. As $L(e_6) = 4 \leq 4$, a and g are reachable w.r.t. the constraint.

Index Construction: Definition 4.3 gives a straightforward way to build the edge-based index tree $\mathcal{T}$ for an MST T in a top-down fashion. The index construction is shown in Algorithm 4.1 and is self-explanatory. Line 3 needs $O(|V|)$ time to find e_{max} with the maximum weight, and it is processed for $|V| - 1$ times using recursion. The total time complexity of Algorithm 4.1 is $O(|V|^2)$ for constructing Edge-Index. When the graph contains hundreds of millions of vertices, this time complexity is unacceptable.

Thus we propose a novel linear-time method (in Algorithm 4.2) to build the edge-based index tree $\mathcal{T}$ from the MST T in *a bottom-up fashion*, in a way similar to Kruskal's algorithm [54].

In Algorithm 4.2, the edge-based index tree $\mathcal{T}$ is built with a function $R : V(\mathcal{T}) \mapsto V(\mathcal{T})$ which maps a node to its root. Initially, R maps each node to the node itself as the root (line 2–3). So all nodes in $\mathcal{T}$ are isolated. In each iteration, we pick an MST edge $e(a, b) \in E(T)$ in the ascending order of $w(e)$, and create an internal node with a label $w(e)$ in $\mathcal{T}$ (line 5–6). In line 7–8, Find-Root returns the current root of a and b in $\mathcal{T}$ respectively. Line 9–10 unions the two subtrees where a and b reside into one, by linking the two root nodes to the new internal node created in line 6. After inserting all edges $e \in E(T)$ as internal nodes in $\mathcal{T}$, the edge-based index tree $\mathcal{T}$ is constructed.

Algorithm 4.2 adopts the classical union-find algorithm [20]. The Find-Root

Algorithm 4.1: Edge-Index-Construct-Naive(T)

```
Input: An MST $T$.
Output: An edge-based index tree $\mathcal{T}$ for $T$.
1 if $T$ contains a single node then
2     return a tree of a single node labeled $0$;
3 $e_{max} \leftarrow$ an edge in $T$ with the maximum weight $w_{max}$;
4 remove $e_{max}$ from $T$ to generate two trees $T_1$ and $T_2$;
5 $\mathcal{T} \leftarrow$ a tree with root labeled $w_{max}$;
6 $\mathcal{T}.left \leftarrow$ Edge-Index-Construct-Naive($T_1$);
7 $\mathcal{T}.right \leftarrow$ Edge-Index-Construct-Naive($T_2$);
8 return $\mathcal{T}$;
```

procedure returns the current root of a node in $\mathcal{T}$. It adopts the path compression technique [20], thus the amortized time complexity for each Find-Root operation is $O(\alpha(|V|))$. Here $\alpha(n)$ is the inverse Ackermann function which is a very slowly growing function, and can be considered as a small constant. As an indicator, $\alpha(2^{2^{2^{65536}}} - 3) = 4$. As we totally perform $O(|V|)$ Find-Root operations, the time for all Find-Root operations is $O(|V|)$. In addition, sorting all edges in $E(T)$ can be done in $O(|V|)$ time using radix sort since all edge weights are from a finite set. Hence the overall time complexity of building an edge-based index tree $\mathcal{T}$ is $O(|V|)$.

Example 4.5. The edge-based index tree $\mathcal{T}$ in Figure 4.2 is constructed from T in Figure 4.1(b) as follows. Initially all leaf nodes are isolated and the root of each node is itself. We first sort the edges in $E(T)$ in the ascending order of their weights to be the edge sequence $e_0, \ldots, e_6$. In the first step, we create an internal node corresponding to $e_0(a, f, 2)$, and link nodes a and f as two children of the e_0 node. Next, we create an internal node corresponding to $e_1(b, c, 2)$ and link nodes b and c to it. In the third step, we create an internal node corresponding to $e_2(e, f, 2)$ and link node e and the current root of node f, i.e., node $e_0(a, f, 2)$ to the e_2 node. This process iterates until we insert all 7 edges $e_0, \ldots, e_6$ as internal nodes into $\mathcal{T}$.

Algorithm 4.2: Edge-Index-Construct(T)

```
   Input: An MST $T$.
   Output: An edge-based index tree $\mathcal{T}$ for $T$.
1  $\mathcal{T} \leftarrow \emptyset$;
2  $R(v) \leftarrow v, \forall v \in V(T)$;
3  $R(e) \leftarrow e, \forall e \in E(T)$;
4  create a leaf node $v$ labeled 0 in $\mathcal{T}$, $\forall v \in V(T)$;
5  for $e(a, b) \in E(T)$ in ascending order of $w(e)$ do
6      create an internal node $e$ labeled $w(e)$ in $\mathcal{T}$;
7      $e.left \leftarrow$ Find-Root($a$);
8      $e.right \leftarrow$ Find-Root($b$);
9      $R(e.left) \leftarrow e$;
10     $R(e.right) \leftarrow e$;
11 return $\mathcal{T}$;
12 Procedure Find-Root($v$)
13 if $R(v) = v$ then
14     return $v$;
15 $R(v) \leftarrow$ Find-Root($R(v)$);
16 return $R(v)$;
```

Lemma 4.3. *Constructing* Edge-Index *takes* $O(|E|)$ *time and* $O(|V|)$ *space. Using* Edge-Index*, the query time is* $O(1)$.

Proof. We first build an MST from G in $O(|E|)$ time, then build the edge-based index tree and the LCA index in $O(|V|)$ time. Thus the time complexity for constructing Edge-Index is $O(|E| + |V|)$, or simplified as $O(|E|)$, as $|V| \leq |E| + 1$ usually holds, assuming G is a connected graph. The spaces for both the edge-based index tree and the LCA index are $O(|V|)$, thus the space complexity is $O(|V|)$. The query time is $O(1)$ for finding $\text{LCA}_{\mathcal{T}}(a, b)$ on $\mathcal{T}$. □

4.2.3. Extension to Other Constraint Formats

In the previous discussions we assume the edge weight constraint has the form of $\leq y$. If the user-specified weight constraint is $\geq x$, Edge-Index can be applied similarly, since $\geq x$ and $\leq y$ are symmetric. The only change is to build a maximum spanning tree T' of G, instead of a minimum spanning tree. All the complexity results apply to the $\geq x$ constraint.

When the weight constraint is a bounded interval $[x, y]$, we can show that Edge-Index can also be easily extended to handle this more general constraint.

When the Weight Constraint is $[x, y]$

In this subsection, we describe how to extend Edge-Index to handle the constraint $[x, y]$ respectively.

The y constraint can be handled in the same way. To satisfy the x constraint as well, we can simply remove all $e \in E$ with $w(e) < x$. For this purpose, we define the l-Graph.

Definition 4.4 (l-Graph)**.** Given $l \in \mathbb{R}$, we define the l-Graph $G_l(V, E_l, \Sigma_l, w)$ as a subgraph of $G(V, E, \Sigma, w)$, such that $E_l = \{e|e \in E, w(e) \geq l\}$ and $\Sigma_l = \{l'|l' \in \Sigma, l' \geq l\}$.

Index Construction: For each $l \in \Sigma$, we first construct the MST T_l from the l-Graph G_l. A T_l may be a forest, due to the removal of edges with $w(e) < l$. It is trivial to handle this case by adding a virtual root. In the following, we assume T_l is connected for each l. For each MST T_l, $\forall l \in \Sigma$, we invoke Algorithm 4.2 to build an edge-based index tree $\mathcal{T}_l$ and build the LCA index for $\mathcal{T}_l$. Both the index tree $\mathcal{T}_l$, $\forall l \in \Sigma$, and the LCA index are kept in the memory.

Query Processing: Given a query $q = (a, b, [x, y])$, we find $\mathcal{T}_l$ where $l = \min_{l' \in \Sigma}\{l' \geq x\}$. Then we compute $\mathsf{LCA}_{\mathcal{T}_l}(a, b)$ on $\mathcal{T}_l$. a and b are reachable w.r.t. $[x, y]$ if and only if $\overline{P_{T_l}}(a, b) = L(\mathsf{LCA}_{\mathcal{T}_l}(a, b)) \leq y$. Algorithm 4.3 lists the pseudo code for query processing by Edge-Index to handle the constraint $[x, y]$.

Algorithm 4.3: Query-Processing-Edge-Index($\mathcal{T}_l, q$)

```
  Input: Index Trees $\mathcal{T}_l$, $\forall l \in \Sigma$, and a WCR $q(a, b, [x, y])$.
  Output: Whether $a$ and $b$ are reachable w.r.t. $[x, y]$.
1 $l \leftarrow \min_{l' \in \Sigma}\{l' \geq x\}$;
2 $r \leftarrow \mathsf{LCA}_{\mathcal{T}_l}(a, b)$;
3 return $L(r) \leq y$;
```

Lemma 4.4. *To handle a bounded interval constraint* $[x, y]$*, building* Edge-Index *takes* $O(|\Sigma||E|)$ *time and* $O(|\Sigma||V|)$ *space. Using* Edge-Index*, the query time is* $O(1)$.

We can see from Lemma 4.4, to handle a bounded constraint $[x, y]$, the index construction time and space complexities increase by a factor of $|\Sigma|$, but the query time complexity is the same as that for the constraint $\geq x$ or $\leq y$.

4.3. An I/O-Efficient Index

The indexing scheme Edge-Index assumes the index can entirely fit into the main memory, with a space complexity of $O(|\Sigma||V|)$ to handle the weight constraint $[x, y]$. When $|\Sigma|$ or $|V|$ is large which is very common for emerging massive graphs containing hundreds of millions of vertices, the index size may exceed the memory limit. In this section we propose an I/O-efficient algorithm which builds compact disk-resident index for query processing with low I/O cost. We use the basic *external memory model* [89] which moves B data items continuously between external and internal memory as one I/O communication.

A straightforward implementation of the disk-based algorithm is to store all the $|\Sigma|$ MSTs on the disk. Given a query $q = (a, b, [x, y])$, the MST T_l where $l = \min_{l' \in \Sigma}\{l' \geq x\}$, is fetched into the memory to answer q. The disk-based index size is $O(|\Sigma||V|)$. However, it needs $O(|V|/B)$ I/O for query processing where B is the page size, as it fetches the whole MST T_l into the memory. Thus indexing MSTs directly on the disk is not I/O-efficient for query processing. Another approach is to store the edge-based index tree $\mathcal{T}_l$ as well as its LCA index, $\forall l \in \Sigma$ on the disk. However, answering queries using the disk-based LCA index is not I/O-efficient because of the complex structure used in the LCA index [8]. Our goal is to design a compact disk-based index based on the MSTs to reduce the I/O cost in query processing.

4.3.1. Vertex Coding

For efficient storage and indexing, coding is a commonly used technique. To design an I/O-efficient disk index, we have an intuitive vertex coding scheme on an MST T as follows. We pick an arbitrary vertex $r \in V(T)$ as the root of T, thus T becomes a rooted MST. Given two vertices a and b, we denote the lowest common ancestor of a and b on such a tree as r_{ab}. Since r_{ab} lies on the tree path between a and b, we have

$$\overline{P_T}(a,b) = \max\{\overline{P_T}(a,r_{ab}), \overline{P_T}(b,r_{ab})\}.$$

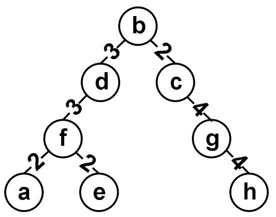
where $P_T(a, r_{ab})$ denotes the tree path between a and r_{ab}.

Based on the above equation, we generate a code for every vertex $v \in V(T)$. In the rooted MST T, we define the level of the root as 0, and the level of a child node increases that of its parent by 1. For any vertex v at level l of T, we find its ancestors r_i at level i, $0 \leq i \leq l-1$. We also pre-compute the maximum edge weight $\overline{P_T}(v, r_i)$ on the path $P_T(v, r_i)$ from v to r_i on T. The code of a vertex v, $\mathsf{code}(v)$, is

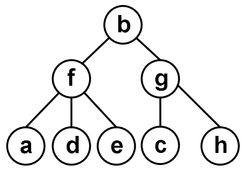
$$\begin{aligned}\mathsf{code}(v) = \{(r_i, \overline{P_T}(v,r_i)) \quad | \quad & r_i \text{ is } v\text{'s ancestor at level } i, \\ & 0 \leq i \leq l-1\}\end{aligned}$$

The code is stored on the disk as index.

To answer a query $q = (a, b, [x, y])$, we retrieve the pages containing $\mathsf{code}(a)$ and $\mathsf{code}(b)$, from both of which we find the lowest common ancestor r_{ab} of a and b in the rooted MST. Thus $\overline{P_T}(a,b) = \max\{\overline{P_T}(a,r_{ab}), \overline{P_T}(b,r_{ab})\}$ can be determined by the two codes with no extra overhead. The I/O cost for a query is $O(h/B)$ on retrieving $\mathsf{code}(a)$ and $\mathsf{code}(b)$, where h is the tree height of the rooted MST (h is also the maximum number of ancestors). The code based index uses $O(|V|h)$ space for one MST, or $O(|\Sigma||V|h)$ for $|\Sigma|$ MSTs.



(a) Rooted MST



(b) Balanced Tree

Figure 4.3: Rooted MST and Its Balanced Tree

Example 4.6. Figure 4.3(a) shows a rooted MST of T where vertex b is an arbitrarily picked root. To answer a query $q = (a, g, [2, 4])$, we retrieve

$$\mathsf{code}(a) = \{(b, \overline{P_T}(a, b)), (d, \overline{P_T}(a, d)), (f, \overline{P_T}(a, f))\},$$

$$\mathsf{code}(g) = \{(b, \overline{P_T}(g, b)), (c, \overline{P_T}(g, c))\}.$$

We can find $\mathsf{LCA}(a, g) = b$. So

$$\overline{P_T}(a, g) = \max\{\overline{P_T}(a, b), \overline{P_T}(g, b)\} = 4.$$

The above coding method is not I/O efficient, because an MST may appear in arbitrary shape and thus the height h of an MST is $O(|V|)$ in the worst case, which causes $O(|V|/B)$ I/O cost for query processing and $O(|\Sigma||V|^2)$ index size in the worst case, and is too expensive. To address this issue, we design a novel tree re-balancing technique to reorganize the MST into a balanced rooted tree, such that the height has an upper bound of $\log_2 |V|$. With the balanced tree, our disk-based algorithm only needs four I/Os for query processing and $O(|\Sigma||V| \log |V|)$ index size on the disk.

In the following, we will first introduce the tree re-balancing technique in Chapter 4.3.2, and then propose the disk-based index construction method in Chapter 4.3.3 and query processing algorithm in Chapter 4.3.4.

4.3.2. MST Re-balancing

In the vertex coding scheme, the key to reduce the I/O cost in query processing and the disk-based index size is to make the height h of a rooted MST T as small as possible. However, when T is a chain, even if we pick the center node of T as its root, its height h is still as large as $\frac{|V|}{2}$. Thus we need to re-balance T in order to make h small. Suppose we select a certain vertex $r \in V(T)$ as the root of T. Underneath the root r, T has $p \geq 1$ disjoint subtrees $T_1, T_2, \ldots, T_p$. For each subtree T_i, we select a vertex $r_i \in V(T_i)$ as the root of T_i for the balance purpose. This re-balancing process is recursively done in a top-down fashion to the leaf nodes. Although the re-balanced tree is no longer an MST, the vertex coding based index and query processing can still be applied based on the following property.

Property 4.1. Once the root r of T is fixed, no matter how the subtrees $T_1, T_2, \ldots, T_p$ under r are re-balanced, for any two vertices $a \in V(T_i)$ and $b \in V(T_j)$ for $i \neq j$, their LCA in the re-balanced tree is still r, as a and b are from disjoint subtrees. Furthermore, r must lie on $P_T(a, b)$. The maximum weights $\overline{P_T}(a, r)$ and $\overline{P_T}(b, r)$ are pre-computed based on the MST paths $P_T(a, r)$ and $P_T(b, r)$ in the original unbalanced MST. $\overline{P_T}(a, b) = \max\{\overline{P_T}(a, r), \overline{P_T}(b, r)\}$ can be computed in the same way regardless of how the subtrees T_1, $T_2, \ldots, T_p$ are re-balanced.

The main purpose of the balanced tree is to reduce the tree height, thus reduce the code length and the disk-based index size. For an MST T, we define its balanced tree $\mathcal{B}[T]$.

Definition 4.5 (Balanced Tree)**.** For an MST T, its balanced tree, denoted as $\mathcal{B}[T]$, is a rooted tree such that

- The root node r of $\mathcal{B}[T]$ corresponds to a node in T.
- The height of $\mathcal{B}[T]$ is at most $\log_2 |V(T)|$.
- Underneath the root r, T has $p \geq 1$ disjoint subtrees $T_1, T_2, \ldots, T_p$, then the subtrees under root r in $\mathcal{B}[T]$ are $\mathcal{B}[T_1], \mathcal{B}[T_2], \ldots, \mathcal{B}[T_p]$.

```
Algorithm 4.4: Find-Median-Node($T$)
    Input: An MST $T$.
    Output: The median node $v_m$ of $T$.
1   Traverse $T$ from an arbitrarily picked root $r'$;
2   $v_m \leftarrow$ the lowest node $u$ with subtree rooted at $u$ satisfying $|V(T_u)| > \lfloor \frac{|V(T)|}{2} \rfloor$;
3   return $v_m$;
```

If the context is obvious, we will use $\mathcal{B}$ to denote $\mathcal{B}[T]$.

Example 4.7. Figure 4.3(b) shows the balanced tree $\mathcal{B}$ for the unbalanced rooted MST in Figure 4.3(a). We pick node b as the tree root at level 0, and nodes f and g as the roots of the left and right subtrees at level 1. The tree height $h = 2 < \log_2 8 = 3$, as $|V(T)| = 8$. For vertices a and g, we can see that their lowest common ancestor in $\mathcal{B}$ is still vertex b, although the left and right subtrees are re-balanced.

Before introducing our algorithm to construct the balanced tree, we first study a special type of node in an MST T called *median node* which is defined as follows.

Definition 4.6 (Median Node)**.** Given an MST T, a node $v_m \in V(T)$ is a median node of T, if for each neighbor of v_m, i.e., $\forall v' \in \{v|(v_m, v) \in E(T)\}$, $|V(T_{v'})| \leq \lfloor \frac{|V(T)|}{2} \rfloor$ holds, where $T_{v'}$ is the incident subtree of v_m rooted at a neighbor node v' and $|V(T_{v'})|$ is the number of nodes in $T_{v'}$.

We will select v_m as the root of a balanced tree $\mathcal{B}$. However, does the median node always exist? If yes, how to find it?

Algorithm 4.4 gives a constructive proof that such a median node always exists – the lowest node u with the subtree size satisfying $S_u = |V(T_u)| > \lfloor \frac{|V(T)|}{2} \rfloor$ is the median node. We start from an arbitrarily picked root $r' \in V(T)$ to traverse the tree and find the median node. By 'lowest', we guarantee for each child node u_c of u, $|V(T_{u_c})| \leq \lfloor \frac{|V(T)|}{2} \rfloor$ holds. We denote u's parent as u_p. After we re-root the tree on u,

Algorithm 4.5: Balanced-Tree-Construct(T)

```
Input: An MST $T$.
Output: A balanced tree $\mathcal{B}$ for $T$.
1 if $|V(T)| = 1$ then
2     return a tree with the only node in $T$;
3 $r \leftarrow$ Find-Median-Node($T$);
4 $\mathcal{B} \leftarrow$ a tree of a single node $r$;
5 consider disjoint subtrees $T_1, T_2, \ldots, T_p$ under $r$ in $T$;
6 for $i = 1$ to $p$ do
7     $\mathcal{B}_i \leftarrow$ Balanced-Tree-Construct($T_i$);
8     add $\mathcal{B}_i$ to be a subtree under $r$ in $\mathcal{B}$;
9 return $\mathcal{B}$;
```

the size of subtree rooted at node u_p is

$$\begin{aligned} |V(T_{u_p})| &= |V(T)| - S_u \\ &\leq |V(T)| - \lfloor\frac{|V(T)|}{2}\rfloor - 1 \\ &\leq \lfloor\frac{|V(T)|}{2}\rfloor \end{aligned}$$

Therefore, we prove for each neighbor v of u, the subtree rooted at v has a size satisfying $|V(T_v)| \leq \lfloor\frac{|V(T)|}{2}\rfloor$. Thus u is a median node by definition. Algorithm 4.4 costs $O(|V(T)|)$ time to traverse the tree and find the median node.

Given an MST T, we choose the median node as the root, and construct a balanced tree recursively based on Definition 4.5. The algorithm to construct the balanced tree $\mathcal{B}$ for T is shown in Algorithm 4.5 and is self-explanatory. We have the following theorem.

Theorem 4.2. *Given an* MST T*, the height of the balanced tree* $\mathcal{B}[T]$ *built is at most* $\log_2 |V(T)|$.

Proof. Let $h(\mathcal{B})$ be the height of the tree $\mathcal{B}$, and $h(n)$ be the maximum height of the

Table 4.1: Complexity Results for Bounded Interval Constraint $[x, y]$

Methods	**Index Time**	**Index Size**	**Query Time**
Edge-Index (memory-based)	$O(\lvert\Sigma\rvert\lvert E\rvert)$	$O(\lvert\Sigma\rvert\lvert V\rvert)$	$O(1)$
Balanced-Index (disk-based)	$O(\lvert\Sigma\rvert\lvert E\rvert + \lvert\Sigma\rvert\lvert V\rvert \log \lvert V\rvert)$	$O(\lvert\Sigma\rvert\lvert V\rvert \log \lvert V\rvert)$	4 I/Os

balanced tree $\mathcal{B}[T]$ of any tree T with n nodes. Obviously, $h(n)$ is a nondecreasing function. In Algorithm 4.5, we find a median node r in T, whose removal from T generates p disjoint trees $T_1, T_2, \ldots, T_p$. Since r is a median node, we have $|V(T_i)| \leq \lfloor\frac{|V(T)|}{2}\rfloor, \forall 1 \leq i \leq p$. From the construction of $\mathcal{B}$, we know

$$\begin{aligned} h(\mathcal{B}[T]) &= \max_{1\leq i\leq p} h(\mathcal{B}[T_i]) + 1 \\ &\leq \max_{1\leq i\leq p} h(\lfloor\frac{|V(T)|}{2}\rfloor) + 1 = h(\lfloor\frac{|V(T)|}{2}\rfloor) + 1 \end{aligned}$$

Thus,

$$\begin{aligned} h(n) &= \max_{\forall T, s.t. |V(T)|=n} h(\mathcal{B}[T]) \\ &\leq h(\lfloor\frac{n}{2}\rfloor) + 1 \\ &\leq h(\lfloor\frac{n}{2^2}\rfloor) + 2 \\ &\leq \cdots \leq h(1) + \log_2 n = \log_2 n \end{aligned}$$

Hence we prove $h(\mathcal{B}[T]) \leq h(|V(T)|) \leq \log_2 |V(T)|$. □

Lemma 4.5. *The time complexity for Algorithm 4.5 to construct the balanced tree $\mathcal{B}$ for tree T is $O(|V(T)| \log |V(T)|)$.*

Proof. There are at most $\log_2 |V(T)|$ levels in $\mathcal{B}$, and in each level, finding median nodes for all subtrees on the level takes at most $O(|V(T)|)$ time. The total time complexity is $O(|V(T)| \log |V(T)|)$. □

Algorithm 4.6: Balanced-Index-Construct(G)

```
Input: A graph $G(V, E, \Sigma, w)$.
Output: Balanced-Index for $G$.
1  $\mathcal{I} \leftarrow$ an empty external code index;
2  $\mathcal{O} \leftarrow$ an empty external offset index;
3  for all $l \in \Sigma$ do
4      $T_l \leftarrow$ MST for $G_l$;
5      $\mathcal{B}_l \leftarrow$ Balanced-Tree-Construct($T_l$);
6      for all $v \in V(\mathcal{B}_l)$ do
7          code($v$) $\leftarrow \emptyset$;
8          for all $v$'s ancestor $u$ in a top-down fashion do
9              $P_{T_l}(v, u) \leftarrow$ the path from $v$ to $u$ on $T_l$;
10             code($v$).Append($(u, \overline{P_{T_l}}(v, u))$);
11         $\mathcal{O}$.Append($\mathcal{I}.offset$);
12         $\mathcal{I}$.Append(code($v$));
```

4.3.3. Disk-Based Index Construction

Our disk-based Balanced-Index includes two parts, namely, a code index $\mathcal{I}$ and an offset index $\mathcal{O}$. $\mathcal{I}$ stores all the codes code(v), $\forall v \in T_l$ and $\forall l \in \Sigma$. For each node v, code(v) is a list of $(key, value)$ pairs, where key is the node id for each ancestor of v in $\mathcal{B}[T_l]$ from the root to v, and $value$ is the maximum edge weight on the path from v to key in T_l, i.e., $\overline{P_{T_l}}(v, key)$. $\mathcal{O}$ stores the offsets for all codes in $\mathcal{I}$, because codes are of different sizes, as nodes at different levels in $\mathcal{B}[T_l]$ have different number of ancestors.

Algorithm 4.6 constructs the Balanced-Index from a graph G. We first initialize $\mathcal{I}$ and $\mathcal{O}$ to be two empty lists. Then for each weight $l \in \Sigma$, we construct the balanced tree $\mathcal{B}_l$ from the MST T_l. For each node in $\mathcal{B}_l$, we calculate its code as described above. Note that the maximum edge weight $\overline{P_{T_l}}(v, u)$ on the path from v to v's ancestor u is calculated in T_l, not in $\mathcal{B}_l$. Finally we append the code and the offset to $\mathcal{I}$ and $\mathcal{O}$ respectively.

Lemma 4.6. *Using Algorithm 4.6,* Balanced-Index *for graph G can be constructed using $O(|\Sigma||E| + |\Sigma||V|\log|V|)$ time and $O(|\Sigma||V|\log|V|)$ disk space. The I/O cost to store the index is $O(\frac{|\Sigma||V|\log|V|}{B})$ where B is the page size.*

Proof. We first build $|\Sigma|$ MSTs in $O(|\Sigma||E|)$ time, and $|\Sigma|$ balanced trees in $O(|\Sigma||V|\log|V|)$ time. This uses $O(|V|)$ memory space for processing one MST and balanced tree at a time. We also need $O(|\Sigma||V|\log|V|)$ time and disk space to calculate and store all the codes from the balanced trees, because each balanced tree has $|V|$ codes and each code has at most $\log_2|V|$ $(key, value)$ pairs. Since creating $\mathcal{I}$ and $\mathcal{O}$ uses sequential I/Os, the total I/O cost is $O(\frac{|\Sigma||V|\log|V|}{B})$. □

4.3.4. Query Processing

We show how to process queries using the disk-based offset index $\mathcal{O}$ and code index $\mathcal{I}$ in Algorithm 4.7. Given a query $q = (a, b, [x, y])$, find $l = \min_{l' \in \Sigma}\{l' \geq x\}$. Suppose l is ranked r_l in Σ, and the id of node a in graph G is id_a. Let $p = r_l \times |V| + id_a$, we can get the offset of code(a) in $\mathcal{I}$ by retrieving the p-th element from index $\mathcal{O}$ with one I/O. Using the offset, we can retrieve code(a) from index $\mathcal{I}$, which contains at most $\log_2|V|$ $(key, value)$ pairs. This operation needs one I/O since one page is enough to hold $\log_2|V|$ pairs in a code for a very large $|V|$, i.e., in the scale of $O(2^B)$, where B is the page size. We set $B = 4096$ bytes in our implementation. Similarly, we retrieve code(b) from index $\mathcal{I}$. We compare the i-th element in code(a) with the i-th element in code(b) one by one in a top-down fashion, until they are not referring to the same node. The last node with code$(a)[i].key =$ code$(b)[i].key$ corresponds to the LCA of a and b in $\mathcal{B}_l$. We have

$$\overline{P_{T_l}}(a, b) = \max\{\mathsf{code}(a)[i].value, \mathsf{code}(b)[i].value\}$$

Thus we return true if $\overline{P_{T_l}}(a, b) \leq y$, and return false otherwise. We totally need four I/Os to retrieve the index entries for a and b to answer a WCR query. The offset index

Algorithm 4.7: Query-Processing-Balanced-Index($\mathcal{I}, \mathcal{O}, q$)

```
    Input: The code index $\mathcal{I}$ and the offset index $\mathcal{O}$, and a WCR query
           $q(a, b, [x, y])$.
    Output: Whether $a$ and $b$ are reachable w.r.t. $[x, y]$.
1   $l \leftarrow \min_{l' \in \Sigma}\{l' \geq x\}$;
2   $o_a \leftarrow \mathcal{O}$.Get-Offset($l, a$); code($a$) $\leftarrow \mathcal{I}$.Get-Code($o_a$);
3   $o_b \leftarrow \mathcal{O}$.Get-Offset($l, b$); code($b$) $\leftarrow \mathcal{I}$.Get-Code($o_b$);
4   $i \leftarrow \max\{j |$code$(a)[j].key =$ code$(b)[j].key\}$;
5   return $\max\{$code$(a)[i].value,$ code$(b)[i].value\} \leq y$;
```

$\mathcal{O}$ contains $|\Sigma||V|$ offset values, and is typically small enough to fit in the memory. Thus if $\mathcal{O}$ is in the memory, we only need two I/Os for query processing.

Lemma 4.7. *Answering a* WCR *query with* Balanced-Index *by Algorithm 4.7 takes* $O(\log |V|)$ *time and four I/Os.*

Proof. The I/O cost is on retrieving the offsets from $\mathcal{O}$ and the codes from $\mathcal{I}$. We need one I/O to retrieve each offset. The number of $(key, value)$ pairs in a code is at most $\log_2 |V|$ and is small enough to fit into one page, thus we can use one I/O to retrieve the code for each node. Thus we totally need four I/Os. After retrieving the codes into memory, in the worst case, we need to traverse all elements in the two codes once to find the LCA of a and b in $\mathcal{B}_l$, which needs $O(\log |V|)$ time. □

The disk-based algorithm can be applied directly to solve the half-bounded constraint $\geq x$ and $\leq y$. The only difference is that, we build only one balanced tree $\mathcal{B}$ from the MST of the graph G. Thus the index construction time is $O(|E|+|V| \log |V|)$ and the disk index size is $O(|V| \log |V|)$. The query time is the same as in Lemma 4.7. Finally, Table 4.1 summarizes the complexities of our proposed algorithms.

4.4. Experiments

In this section we perform extensive experimental studies on real and synthetic datasets. We systematically test our memory algorithm Edge-Index and the I/O-efficient algorithm Balanced-Index. We report three performance measures, index construction time (IT), index size (IS), and query time (QT). All our algorithms are implemented in C++, and our experiments are performed on a machine with a 2.67GHz CPU and 12GB memory.

Besides the two algorithms we proposed, we also test the following baseline methods for comparison.

1. Naive-Search Methods: We take three basic search approaches, depth-first search (DFS), breadth-first search (BFS), and bi-directional search (BIS) [24], as memory-based baselines. Given a query $q = (a, b, [x, y])$, to find a valid path between a and b, Naive-Search only visits edges which satisfy the $[x, y]$ constraint. Being different only on the searching order, they share the same index time $O(|E| \log |E|)$ (for pre-sorting all graph edges according to the *start node id* of the edges, if the input is not sorted), index size $O(|V| + |E|)$ (for storing the original graph G), and worst-case query time $O(|V| + |E|)$.

   We also use them as disk-based baselines by adapting them to external memory, denoted as Ext-DFS, Ext-BFS, and Ext-BIS respectively. In the preprocessing phase, we need to pre-sort all edges in memory in $O(|E| \log |E|)$ time, and then sequentially store the adjacency lists on disk with $O(|E|/B)$ I/Os. The disk-based index takes $O(|E|)$ size. In the online phase, when extending a node v, it needs $O(1 + degree(v)/B)$ I/Os to fetch v's neighbors from disk. Thus the total number of I/Os for query processing is $O(\sum_{v \in V}(1 + degree(v)/B)) = O(|V| + |E|/B)$ I/Os in the worst case.

2. MST-Index: MST-Index is a memory-based baseline, which builds $|\Sigma|$ MSTs in memory, denoted as T_l, $\forall l \in \Sigma$. Given a query $q = (a, b, [x, y])$, MST-Index

Table 4.2: Real Network Statistics

Network	$\|V\|$	$\|E\|$	$\|\Sigma\|$
Facebook (Day)	63,731	440,384	862
Facebook (Hour)	63,731	440,384	19,657
USARN	23,947,347	29,166,672	12

finds $l = \min_{l' \in \Sigma}\{l' \geq x\}$ and computes $\overline{P_{T_l}}(a, b)$ on T_l. The index time and space complexities of MST-Index are the same as those of Edge-Index, but its query time is $O(|V|)$.

3. External-MST: External-MST is a disk-based baseline, which stores $|\Sigma|$ MSTs on disk. Given a query $q = (a, b, [x, y])$, External-MST finds $l = \min_{l' \in \Sigma}\{l' \geq x\}$ and fetches the MST T_l into the memory. Then it computes $\overline{P_{T_l}}(a, b)$ on T_l. It uses $O(|V|/B)$ I/Os for query processing.

4. Sampling-Tree [42] and 2-Label-Hop [92]: These two approaches handle label-constraint reachability (LCR) on directed graphs, where the labels appearing on the path from a node u to another node v should be a subset of a user provided label set. In terms of problem hardness, LCR is more difficult than WCR. But since LCR is the closest to our WCR problem in the literature, we adapt Sampling-Tree and 2-Label-Hop to answer WCR query on undirected graphs for performance comparison.

4.4.1. Experiments on Real Datasets

In this experiment, we evaluate the performance of different methods on two real world datasets. The first is the Facebook New Orleans network[1] [88] over a period of two years. A node represents a user and an undirected edge denotes a user-to-user friendship link. Each edge has a weight denoting the UNIX timestamp with the time of link establishment. We generalize the timestamps into two granularities, hour and day. The two resulting networks are denoted as Facebook (Hour) and Facebook (Day). The

[1]http://socialnetworks.mpi-sws.org/data-wosn2009.html

Table 4.3: Memory-based Algorithms on Real Dataset Results (IT in Seconds, IS in GB, QT in Microseconds)

	Naive-Search					MST-Index			Edge-Index		
	IT	IS	QT			IT	IS	QT	IT	IS	QT
			DFS	BFS	BIS						
Facebook(Day)	0.4	0.01	1,098	1,429	2,324	27.9	0.66	1	48.9	2.20	1
Facebook(Hour)	0.4	0.01	1,500	1,377	1,860	-	$> 12G$	-	-	$> 12G$	-
USARN	33.7	0.89	32,462	30,868	31,325	119.0	3.45	1,382	469.9	11.49	4

Table 4.4: Disk-based Algorithms on Real Dataset Results (IT in Seconds, IS in GB, QT in Microseconds)

	Naive-Search					External-MST			Balanced-Index		
	IT	IS	QT			IT	IS	QT	IT	IS	QT
			Ext-DFS	Ext-BFS	Ext-BIS						
Facebook(Day)	0.6	0.01	31,368	48,152	45,405	41.8	0.66	772	125.6	2.98	11
Facebook(Hour)	0.5	0.01	35,325	57,366	47,533	922.3	15.03	749	3336.3	68.43	11
USARN	48.8	0.89	294,521	64,471	29,385	146.9	3.45	422,810	1425.9	6.19	18

second graph is the USA road network[2], a representative of very large-scale networks. A node represents an intersection or endpoint while an edge represents a road segment. We generate 12 weights from 10 to 120 with a step size of 10, and randomly assign a weight to each edge to represent the road speed limit. Table 4.2 lists the statistics of these real networks.

We generate and test 10,000 queries for each real network. All the queries have the constraint format of $[x, y]$, where x, y are randomly picked from Σ. Table 4.3 and Table 4.4 show the index time (in seconds), index size (in GB) and average query time (in microseconds) of memory-based and disk-based algorithms respectively. Sampling-Tree and 2-Label-Hop cannot finish index construction on any of these datasets within 12 hours. The comparisons are as follows.

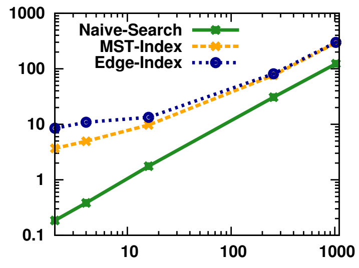
Memory-based Algorithms (Table 4.3). The query time of Edge-Index and MST-Index are the same on Facebook (Day), but on USARN with 24 million nodes,

[2]http://www.dis.uniroma1.it/∼challenge9/download.shtml

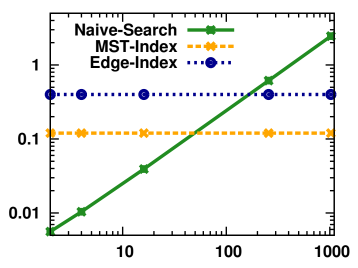
the query time of MST-Index increases dramatically to 1382 microseconds, which is 345 times slower than Edge-Index. This is because the query time complexity of MST-Index is $O(|V|)$. The index time of Edge-Index is 1.75–3.95 times that of MST-Index, and the index size of Edge-Index is about 3 times larger, because Edge-Index needs to build edge-based index trees and LCA index. Both methods run out of memory on Facebook (Hour) on indexing due to the large number of weights, $|\Sigma| = 19,657$. Edge-Index uses 13.9–122.3 times longer index time and 12.9–220.0 times larger index size than Naive-Search due to the $|\Sigma|$ factor. But its query time is within 4 microseconds, *three orders of magnitude* faster than that of Naive-Search, which takes 1,098 to 32,462 microseconds.

Disk-based Algorithms (Table 4.4). The query time of Balanced-Index is very stable on all networks, taking 11 or 18 microseconds. The query time of External-MST is orders of magnitude longer than that of Balanced-Index, especially on USARN with 24 million vertices, as it takes $O(|V|/B)$ I/Os to fetch an MST. The index size of Balanced-Index is 2.85–4.3 times larger than that of External-MST and the index time is 3–10 times longer. Both methods do not suffer from the memory limit as they build disk-based index. Balanced-Index's index size is 6.95–6843 times larger than that of Naive-Search, since the index size of Balanced-Index is linear with $|\Sigma|$ whereas that of Naive-Search is $O(|E|)$, not affected by $|\Sigma|$. But the query time of Balanced-Index is within 18 microseconds in all cases, 2,851–16,362 times faster than that of Naive-Search, which takes 29,385 to 294,521 microseconds.

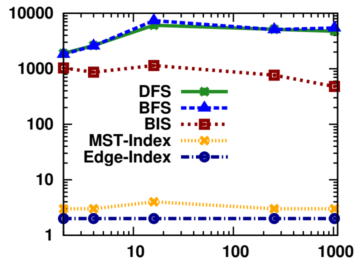
Summary. Our memory method Edge-Index has a very low and stable query time. But the memory index size can be a bottleneck for Edge-Index (also for baseline MST-Index). Naive-Search only stores the original graph which is compact, but its query time is three orders of magnitude longer. In fact, all the baseline methods, Naive-Search, MST-Index and External-MST have a long query time especially when $|V|$ is large. In contrast, our disk-based Balanced-Index is very scalable and the query time remains very low and stable regardless of the network size, the edge density, or the distinct weight number. The choice of algorithm depends upon the network size



(a) Index Time (seconds)

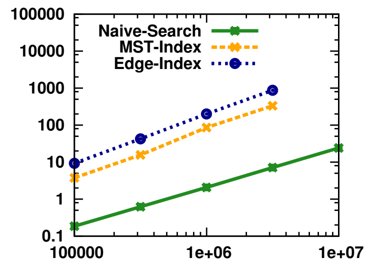


(b) Index Size (GB)

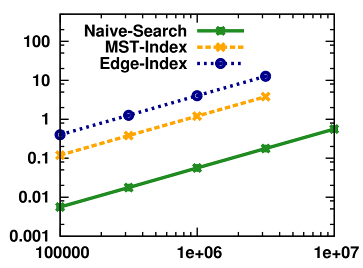


(c) Query Time (microseconds)

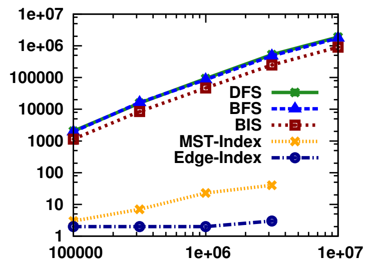
Figure 4.4: Memory-based algorithms on synthetic datasets, varying $|E|/|V|$ from 2 to 1024, $|V| = 10^5$, $|\Sigma| = 100$



(a) Index Time (seconds)



(b) Index Size (GB)



(c) Query Time (microseconds)

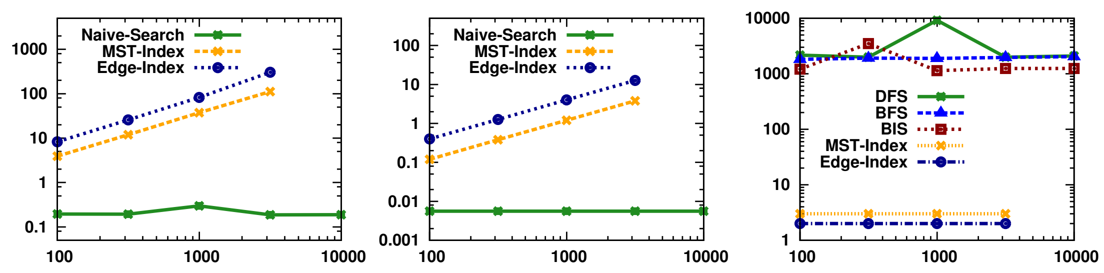
Figure 4.5: Memory-based algorithms on synthetic datasets, varying $|V|$ from 10^5 to 10^7, $|E|/|V| = 2$, $|\Sigma| = 100$

and the available memory.

4.4.2. Memory-based Algorithms on Synthetic Datasets

To test the parameter sensitivity of different methods, we generate a collection of random graphs based on *Erdös-Rényi* model [46] by varying three parameters $|E|/|V|$, $|V|$ and $|\Sigma|$. $|E|/|V|$ is the density of the graph. The default values are $|E|/|V| = 2$, $|V| = 10^5$ and $|\Sigma| = 100$. We assign a random weight to each edge in the graphs. The edge weights follow a uniform distribution.

We first test our memory algorithm Edge-Index. For comparison, we also test the baselines, main memory Naive-Search (DFS, BFS and BIS) and MST-Index. We test $10,000$ random queries on each graph, and report index construction time (in seconds),



(a) Index Time (seconds) (b) Index Size (GB) (c) Query Time (microseconds)

Figure 4.6: Memory-based algorithms on synthetic datasets, varying $|\Sigma|$ from 10^2 to 10^4, $|E|/|V| = 2$, $|V| = 10^5$

index size (in GB) and the average query time (in microseconds). All the queries have the constraint format of $[x, y]$, where x, y are random real numbers from Σ.

Varying Density: In this experiment, we vary the edge density $|E|/|V|$ from 2 to 1024 in log scale, and fix $|V| = 10^5$ and $|\Sigma| = 100$. Figures 4.4(a)-(c) show the index construction time, index size and query time in log scale, respectively.

The index time of Edge-Index and MST-Index increases with the density $|E|/|V|$, or equivalently with $|E|$ as $|V|$ is fixed, since they both take $O(|\Sigma||E|)$ time to build MSTs. In addition, Edge-Index needs to build edge-based index trees and LCA index in $O(|\Sigma||V|)$ time, thus its index time is longer. But the margin between Edge-Index and MST-Index decreases with the increase of $|E|$, as the MST construction time dominates LCA index building time given a large $|E|$. The index time of Naive-Search is linear with $|E|$ since it is dominated by the loading cost of the graph, $|E|/B$ I/Os, which is much larger than $O(|E| \log |E|)$ sorting cost.

The index size of Edge-Index and MST-Index is not affected by the density. MST-Index uses 0.12 GB space for indexing MSTs, each of which contains $|V|$ vertices; Edge-Index uses 0.40 GB to store the edge-based index trees, each of which contains $\leq 2|V|$ vertices, as well as the LCA index. In contrast, the index size of Naive-Search increases linearly with $|E|$. When $|E|/|V| = 1024$, Naive-Search uses 6 times larger index size than Edge-Index.

Edge-Index takes 2 microseconds for query processing via an $O(1)$ LCA operation.

Thus its query time is not affected by the density. The query time of MST-Index is around 3–4 microseconds when the density changes. As it needs to search the MST online for the path between two query nodes, the query time depends on the MST structure and how distant two query nodes are on the MST. The query time of Edge-Index is 915–3,682 times faster than that of DFS and BFS on graphs with different density. Although BIS slightly improves DFS and BFS (by 2–10 times) by reducing the searching space, it is still at least two orders of magnitude slower than Edge-Index.

Varying Vertex Number: In this experiment, we vary $|V|$ from 10^5 to 10^7 in log scale, and fix $|E|/|V| = 2$ and $|\Sigma| = 100$. Figures 4.5(a)-(c) show the index construction time, index size and query time in log scale, respectively.

The index time of all algorithms increases with $|V|$, as $|E|$ increases with $|V|$ given a fixed density. Since Edge-Index needs to build edge-based index trees and LCA index, its index time is about 2.6 times that of MST-Index, and 93 times longer than that of Naive-Search. The index size of all methods increases linearly with $|V|$. Edge-Index uses index size about 3.3 times more than MST-Index, and 71 times more than Naive-Search. When $|V| = 10^7$, the index size of both Edge-Index and MST-Index exceeds the 12 GB memory limit, but the index size of Naive-Search is much smaller by keeping the original graph only. The query time of Edge-Index remains 2 microseconds when $|V|$ increases, whereas that of MST-Index increases linearly with $|V|$ as the query time is $O(|V|)$. When $|V| = 10^{6.5}$, the query time of MST-Index is 20 times that of Edge-Index, which is a very big difference. The query time of Naive-Search increases linearly with $|V|$ and is three to five orders of magnitude longer than Edge-Index. When $|V| = 10^7$, it takes 1.5 seconds on average to answer one query.

Varying Distinct Weight Number: In this experiment, we vary $|\Sigma|$ from 10^2 to 10^4 in log scale, and fix $|E|/|V| = 2$ and $|V| = 10^5$. Figures 4.6(a)-(c) show the index construction time, index size and query time in log scale, respectively.

The index time and size increase linearly with $|\Sigma|$ for both Edge-Index and

Table 4.5: Online Performance of Memory-based Algorithms

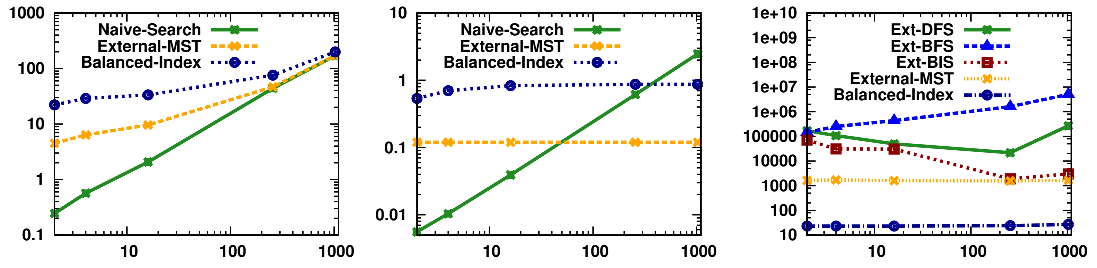
	Dependency	Response Time
Naive-Search	Linear with $\|V\|$	$481 – 1,892,083\ \mu s$
MST-Index	Linear with $\|V\|$	$3 – 57\ \mu s$
Edge-Index	None	$2\ \mu s$

MST-Index. The index time of Edge-Index is 2.4 times that of MST-Index, and the index size of Edge-Index is 3.3 times larger. Both methods run out of memory when $|\Sigma| = 10^4$. The query time of Edge-Index remains 2 microseconds, while that of MST-Index remains 3 microseconds. This demonstrates that our query processing is not affected by the weights appearing in the graph or in the query. The index time, index size and query time of Naive-Search remain stable when increasing $|\Sigma|$. Its query time is three orders of magnitude longer than that of Edge-Index and MST-Index.

Summary. Edge-Index takes only 2 microseconds query time in all networks which is very stable and fast. The query time difference between Edge-Index and MST-Index increases with $|V|$ linearly, thus can be very significant when $|V|$ is large. Edge-Index uses 2–3 times more index time and space than MST-Index, which is a small overhead. When $|V| = 10^7$ or $|\Sigma| = 10^4$, both Edge-Index and MST-Index run out of the 12 GB memory limit. Naive-Search uses much smaller index time and size. But its query time is at least three orders of magnitude longer than that of Edge-Index, moreover, it increases linearly with $|V|$ and $|E|$. This is prohibitive for an online system. The online performance is summarized in Table 4.5.

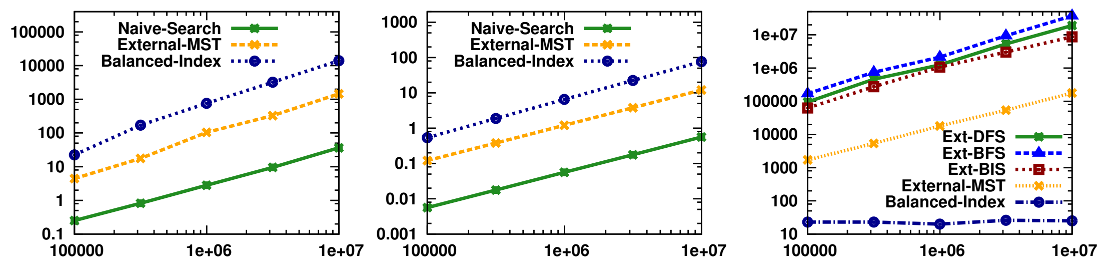
4.4.3. Disk-based Algorithms on Synthetic Data

We test our disk-based algorithm Balanced-Index on the identical set of random graphs with that of our memory-based approaches. For comparison, we also test the following baseline methods, Naive-Search (Ext-DFS, Ext-BFS and Ext-BIS) and External-MST. Naive-Search stores sorted adjacency lists on disk and External-MST stores $|\Sigma|$ MSTs on disk. In query processing, Naive-Search traverses along edges which satisfy the



(a) Index Time (seconds) (b) Index Size (GB) (c) Query Time (microseconds)

Figure 4.7: Disk-based algorithms on synthetic datasets, varying $|E|/|V|$ from 2 to 1024, $|V| = 10^5$, $|\Sigma| = 100$



(a) Index Time (seconds) (b) Index Size (GB) (c) Query Time (microseconds)

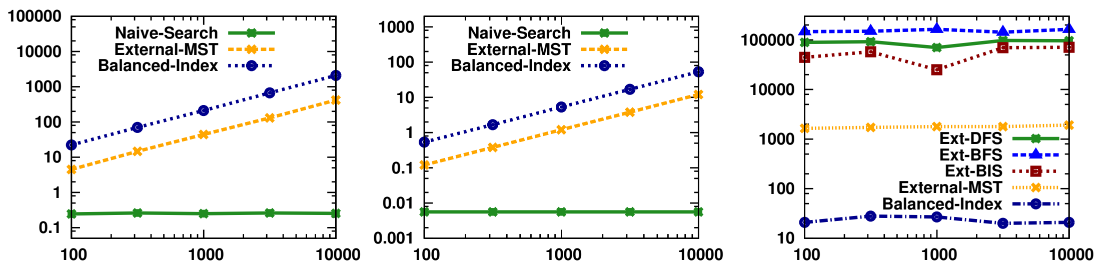
Figure 4.8: Disk-based algorithms on synthetic datasets, varying $|V|$ from 10^5 to 10^7, $|E|/|V| = 2$, $|\Sigma| = 100$

$[x, y]$ constraint, while External-MST loads an MST T_l where $l = \min_{l' \in \Sigma}\{l' \geq x\}$ to answer a WCR query.

Varying Density: In this experiment, we vary the edge density $|E|/|V|$ from 2 to 1024 in log scale, and fix $|V|$ and $|\Sigma|$. Figures 4.7(a)-(c) show the performance measures.

The index time of Balanced-Index is about 4 times that of External-MST when the density is small, i.e., $|E|/|V| = 2 - 16$, but the margin decreases with the density increase. Balanced-Index uses 32 times longer index time than Naive-Search on average, but the margin decreases dramatically to less than 2 when $|E|/|V| \geq 100$.

The index size of External-MST is 0.12 GB under different density values. The index of Balanced-Index is about 4.5–7.3 times larger, as the vertex coding based index takes $O(|\Sigma||V| \log |V|)$ space. Naive-Search's index size increases linearly with $|E|$,



(a) Index Time (seconds) (b) Index Size (GB) (c) Query Time (microseconds)

Figure 4.9: Disk-based algorithms on synthetic datasets, varying $|\Sigma|$ from 10^2 to 10^4, $|E|/|V| = 2$, $|V| = 10^5$

and is 2.8 times larger than that of External-MST when $|E|/|V| = 1024$.

There is a very large margin between the query time of External-MST and Balanced-Index. External-MST takes 1635 microseconds, while Balanced-Index takes 23 microseconds, which is 71 times faster. The query time of External-MST and Balanced-Index is not affected by the density. In contrast, the query time of Naive-Search varies with different densities. Ext-BIS is the most efficient among the three variants. On average, the query time of Naive-Search is three to four orders of magnitude longer than Balanced-Index.

Varying Vertex Number: In this experiment, we vary $|V|$ from 10^5 to 10^7 in log scale, and fix $|E|/|V|$ and $|\Sigma|$. Figures 4.8(a)-(c) show the performance measures.

The index time of Balanced-Index is about 5–10 times longer than that of External-MST, and its index size is about 4.5–6.4 times larger. The index time and index size of all the three methods increase near linearly with $|V|$. The query time of Balanced-Index remains 23 microseconds when $|V|$ increases. In contrast, the query time of External-MST and Naive-Search increases linearly with $|V|$, because External-MST takes $O(|V|/B)$ I/Os to load an MST, and Naive-Search uses $O(|V| + |E|/B)$ I/Os to search online. When $|V| = 10^7$, External-MST takes 176,901 microseconds to answer a query, 7,691 times slower than Balanced-Index. Naive-Search is even slower, with 2,722–764,514 times longer query time than Balanced-Index.

Varying Distinct Weight Number: In this experiment, we vary $|\Sigma|$ from 10^2 to 10^4 in log scale, and fix $|E|/|V|$ and $|V|$. Figures 4.9(a)-(c) show the performance measures.

The index time and size increase linearly with $|\Sigma|$ for both External-MST and Balanced-Index. The index time of Balanced-Index is about 5 times longer than that of External-MST, while the index size of Balanced-Index is about 4.5 times larger. The query time of both methods remains very stable as $|\Sigma|$ increases. The query time of Balanced-Index is about 23 microseconds while that of External-MST is 1770 microseconds, which is 77 times slower. The index time and index size of Naive-Search is independent of $|\Sigma|$, which remain 0.25 seconds and 5.6 MB respectively. Naive-Search's query time does not vary much with $|\Sigma|$ either. For comparison, DFS, BFS, and BIS take 89,000, 153,000 and 53,000 microseconds respectively for query processing on average, whereas Balanced-Index only takes 23 microseconds, which is three orders of magnitude faster.

Summary. Balanced-Index scales to very large networks while at the same time providing 23 microseconds response time. It uses 5–10 times more index time and space than External-MST, which is a small overhead. Naive-Search has a low index time and size in most cases. But when the density $|E|/|V|$ is large, the index size of Naive-Search can be much larger than that of Balanced-Index. The index size of Balanced-Index is independent of $|E|/|V|$ or $|E|$. The query time of Naive-Search is three to four orders of magnitude longer than that of Balanced-Index. The online performance of disk-based approaches is summarized in Table 4.6.

Table 4.6: Online Performance of Disk-based Algorithms

	Dependency	Response Time
Naive-Search	Linear with $\|V\|$	$1,883$ – $37,277,978$ μs
External-MST	Linear with $\|V\|$	$1,573$ – $176,901$ μs
Balanced-Index	None	23 μs

Table 4.7: Edge-Index vs. Sampling-Tree and 2-Label-Hop, Uniform Weights (IT in Secs, IS in MB, QT in Microseconds)

	Edge-Index			Sampling-Tree			2-Label-Hop		
$\|E\|/\|V\|$	IT	IS	QT	IT	IS	QT	IT	IS	QT
1	0.01	0.2	1	72	0.1	86	24	4.55	22
1.5	0.01	0.2	1	339	0.1	106	225	9.74	18
2	0.01	0.2	1	627	0.3	193	574	10.88	20
2.5	0.01	0.2	1	737	0.4	158	1757	12.41	13
3	0.01	0.2	1	953	0.6	135	3773	13.81	13
$\|V\|$	IT	IS	QT	IT	IS	QT	IT	IS	QT
1000	0.01	0.2	1	339	0.1	106	226	9.45	20
2000	0.01	0.4	1	1500	0.3	362	1994	43.59	13
3000	0.01	0.6	1	3630	0.6	454	6299	94.88	12
4000	0.01	0.8	1	14432	1.4	565	17390	182.89	14
5000	0.02	1	1	$> 12h$	-	-	34019	277.36	13
$\|\Sigma\|$	IT	IS	QT	IT	IS	QT	IT	IS	QT
5	0.01	0.2	1	339	0.1	106	228	9.59	23
10	0.01	0.4	1	$> 12h$	-	-	28140	102.83	21
15	0.01	0.6	1	$> 12h$	-	-	$> 12h$	-	-
20	0.02	0.8	1	$> 12h$	-	-	$> 12h$	-	-
25	0.02	1	1	$> 12h$	-	-	$> 12h$	-	-

4.4.4. Comparing Edge-Index with Sampling-Tree and 2-Label-Hop

We compare our solution with Sampling-Tree [42] and 2-Label-Hop [92] in this experiment. Sampling-Tree and 2-Label-Hop are memory algorithms to solve the label-constraint reachability on directed graphs. In terms of problem hardness, LCR is much more difficult than WCR. So adapting LCR solutions to answer WCR queries is unnecessarily complicated, thus is not very fair. Nevertheless, as LCR is the closest problem in the literature to WCR, we compare Edge-Index with Sampling-Tree and 2-Label-Hop in terms of index time, index size and average query time. We adapt both methods to handle undirected graphs. Given a query $q(a, b, [x, y])$, we rewrite it into the query format of Sampling-Tree and 2-Label-Hop by forming a categorical label set $A = \{l \in \Sigma | x \leq l \leq y\}$.

Table 4.8: Edge-Index vs. Sampling-Tree and 2-Label-Hop, Power Law Weights (IT in Secs, IS in MB, QT in Microseconds)

		Edge-Index			Sampling-Tree			2-Label-Hop		
$\|E\|/\|V\|$	$\|\Sigma\|$	IT	IS	QT	IT	IS	QT	IT	IS	QT
1	14	0.02	1	1	125	0.1	311	12	2.90	20
1.5	17	0.02	1.2	1	154	0.4	716	49	3.41	17
2	20	0.02	1.4	1	2376	1.8	837	78	2.96	18
2.5	22	0.02	1.5	1	18078	6.2	663	114	2.68	12
3	24	0.03	1.7	1	$> 12h$	-	-	148	2.63	12
$\|V\|$	$\|\Sigma\|$	IT	IS	QT	IT	IS	QT	IT	IS	QT
1000	17	0.02	1.2	1	154	0.4	716	44	3.45	15
2000	24	0.04	3.4	1	681	1.4	1252	404	15.22	12
3000	30	0.07	6.2	1	2579	5.5	1280	1114	31.61	12
4000	34	0.09	9.6	1	4309	10	3275	3110	61.45	12
5000	38	0.13	13.4	1	6633	17.4	3308	6106	96.03	13

Random Graphs with Uniform Weight Distribution. In this experiment, we generate a set of random graphs by varying the parameters $|E|/|V|$, $|V|$ and $|\Sigma|$. The default values are $|E|/|V| = 1.5$, $|V| = 1000$ and $|\Sigma| = 5$. The edge weights follow a uniform distribution. We report the performance in Table 4.7.

We vary the density $|E|/|V|$ from 1.0 to 3.0 with a step size of 0.5, and fix $|V| = 1000$ and $|\Sigma| = 5$. The index time, index size and query time of Edge-Index are not affected by the density. In contrast, the index time of Sampling-Tree and 2-Label-Hop increases with the density and is four to five orders of magnitude longer than that of Edge-Index. The index size of 2-Label-Hop is 50 times larger than that of Edge-Index on average. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index, and the query time of 2-Label-Hop is about 20 times longer.

When we vary $|V|$ and $|\Sigma|$, the index time and query time of Edge-Index remain stable, while the index size increases linearly with $|V|$ and $|\Sigma|$. In contrast, the index construction time of Sampling-Tree and 2-Label-Hop increases dramatically. When $|V| \geq 5000$ or $|\Sigma| \geq 10$, Sampling-Tree cannot finish index construction within 12 hours. 2-Label-Hop cannot finish index construction within 12 hours when $|\Sigma| \geq 15$. The index size of 2-Label-Hop is 150 times larger than that of Edge-Index. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index, and the query time of 2-Label-Hop is 12–23 times longer.

Random Graphs with Power Law Weight Distribution. In this experiment, we test random graphs with weights following a power law distribution with the parameter $\alpha = 2$ [61], the same setting as in [42]. According to the power law weight distribution, only a few weights appear frequently, while the majority of weights appear infrequently. The default values of the parameters are $|E|/|V| = 1.5$ and $|V| = 1000$. Under a power law distribution, the number of distinct weight values that actually appear in a network depends on $|E|$, thus we do not vary $|\Sigma|$ here. Table 4.8 shows the performance when we vary $|E|/|V|$ and $|V|$. The actual weight number $|\Sigma|$ is listed in the second column of Table 4.8.

First we vary the density $|E|/|V|$ from 1.0 to 3.0 with a step size of 0.5, and fix $|V| = 1000$. We observe that the index time and index size of Edge-Index increase slightly, and the query time remains 1 microsecond. In contrast, the index time of Sampling-Tree and 2-Label-Hop is three to five orders of magnitude longer. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index and the query time of 2-Label-Hop is 16 times longer. When $|E|/|V| = 3.0$, Sampling-Tree cannot finish index construction within 12 hours.

When we vary the vertex number $|V|$, the index time and index size of Edge-Index increase by at most 6.5 and 11 times respectively, as $|V|$, $|E|$ and $|\Sigma|$ all increase. The query time remains 1 microsecond. The index time of Sampling-Tree and 2-Label-Hop is three to four orders of magnitude longer than Edge-Index. Both index time and index size of Sampling-Tree and 2-Label-Hop increase with $|V|$. The query time of Sampling-Tree is three orders of magnitude longer than that of Edge-Index and the query time of 2-Label-Hop is 13 times longer.

Summary. In the above experiments the query time of Edge-Index remains 1 microsecond in all cases, which is one to three orders of magnitude faster than that of Sampling-Tree and 2-Label-Hop. In addition, both Sampling-Tree and 2-Label-Hop suffer from the index construction efficiency. In [42], the indexing time of Sampling-Tree is $O(n|V||E|\binom{|\Sigma|}{|\Sigma|/2} + n/n_0(|E| + |V|\log|V|))$, which increases exponentially with the label set size $|\Sigma|$, and also increases with $|V|$ and $|E|$. 2-Label-Hop [92] needs to pre-compute the local transitive closure in index construction. That is why Sampling-Tree and 2-Label-Hop cannot finish index construction within 12 hours in many cases, even for very small scale, e.g., $|V| = 1000$ and $|\Sigma| = 15$. This demonstrates that both Sampling-Tree and 2-Label-Hop are not efficient to answer the WCR query.

4.5. Summary

In this chapter we study a type of reachability query called *weight constraint reachability* WCR on undirected graphs with real-value edge or node weights. WCR is very common and has a wide range of real-world applications. We design two novel index structures for the memory and disk scenarios respectively. To answer a WCR query, we can guarantee $O(1)$ query time with the memory-based index Edge-Index and $O(1)$ I/O cost (exactly four I/Os) with the disk-based index Balanced-Index. Experimental results on real and synthetic graphs demonstrate that both the memory and disk-based approaches answer a query in microseconds with very compact index and efficient index construction. The disk-based algorithm is highly scalable in large networks and I/O-efficient in query processing.

CHAPTER 5

QUERYING TOP K-NEAREST KEYWORD

5.1. Problem Definition

We model a weighted undirected graph as $G(V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges in G. We use V and E to denote $V(G)$ and $E(G)$ if the context is obvious. Each edge $(u, v) \in E$ has a positive weight, denoted as $\mathsf{weight}(u, v)$. A path $p = (v_1, v_2, \cdots, v_l)$ is a sequence of l nodes in V such that for each $v_i(1 \leq i < l)$, $(v_i, v_{i+1}) \in E$. The weight of a path is the total weight of all edges on the path. For any two nodes $u \in V$ and $v \in V$, the distance of u and v on G, $\mathsf{dist}(u, v)$, is the minimum weight of all paths from u to v in G. Each node $v \in V$ contains a set of zero or more keywords which is denoted as $\mathsf{doc}(v)$. The union of keywords for all nodes in G is denoted as $\mathsf{doc}(V)$. Note that $\mathsf{doc}(V)$ is a multiset and $|\mathsf{doc}(V)| = \sum_{v \in V} |\mathsf{doc}(v)|$. We use $V_\lambda \subseteq V$ to denote the set of nodes carrying keyword λ in V.

Definition 5.1. Given a graph $G(V, E)$, a top-k nearest keyword ($\mathsf{k\text{-}NK}$) query is a triple $Q = (q, \lambda, k)$, where $q \in V$ is a query node in G, λ is a keyword, and k is a positive integer. Given a query Q, a node $v \in V$ is a keyword node w.r.t. Q if v contains keyword λ, i.e., $v \in V_\lambda$. The result is a set of k keyword nodes, denoted as $R = \{v_1, v_2, \cdots, v_k\} \subseteq V_\lambda$, and there does not exist a node $u \in V_\lambda \setminus R$ such that

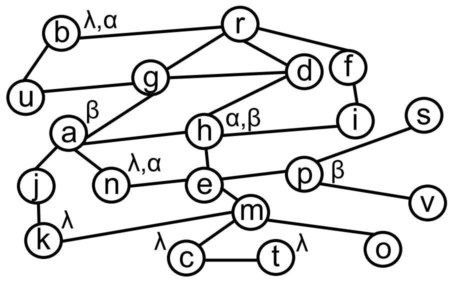


Figure 5.1: A Graph G with Keywords

$\mathsf{dist}(q,u) < \max_{v\in R}\mathsf{dist}(q,v)$. To further report the distance in the top-k result, we can use the form $R = \{v_1 : \mathsf{dist}(q,v_1), v_2 : \mathsf{dist}(q,v_2), \cdots, v_k : \mathsf{dist}(q,v_k)\}$.

In this chapter, we aim at answering a k-NK query $Q = (q,\lambda,k)$ on a graph G. For simplicity, we assume that there is only one keyword λ in the query. We will discuss how to answer a query containing multiple keywords with AND and OR semantics.

Example 5.1. Fig. 5.1 shows a graph G. Assume that the weight of each edge is 1. For a k-NK query $Q = (f,\lambda,3)$, the keyword node set is $V_\lambda = \{b,c,k,n,t\}$. The result of Q is $R = \{b : 2, n : 4, k : 5\}$ since $\mathsf{dist}(f,b) = 2$, $\mathsf{dist}(f,n) = 4$, and $\mathsf{dist}(f,k) = 5$.

5.2. Existing Solutions

A straightforward approach to answering a k-NK query $Q = (q,\lambda,k)$ on G is to use Dijkstra's algorithm to search from the query node q and output k nearest keyword nodes in nondecreasing order of their distances to q. The time complexity is $O(|E| + |V|\cdot\log|V|)$. Obviously, Dijkstra's algorithm is inefficient when the size of the graph is large or the keyword nodes are far away from q.

In the literature, [5] and [83] design different indexing schemes to process (top-k) nearest keyword queries on a graph or a tree. We introduce the two methods in the following two subsections.

5.2.1. Approximate k-NK on a Graph

Bahmani and Goel [5] find an approximate answer to a $\mathsf{k\text{-}NK}$ query in a graph based on a distance oracle [78].

Distance Oracle: Distance oracle is a technique for estimating the distance of two nodes in a graph [78]. Given a graph G, a distance oracle is a Voronoi partition of $V(G)$ determined by a set of randomly selected *center nodes*. More specifically, given a number n_c, we randomly select n_c nodes from $V(G)$ as the center nodes to construct a distance oracle $\mathcal{O}$. Then the partition is constructed by assigning each node $v \in V(G)$ to its nearest center node, denoted as $\mathsf{wit}_{\mathcal{O}}(v)$, which is called the witness node of v w.r.t. $\mathcal{O}$. If v is a center node, $\mathsf{wit}_{\mathcal{O}}(v) = v$. For each node $v \in V(G)$, the shortest distance from v to its witness node, i.e., $\mathsf{dist}(v, \mathsf{wit}_{\mathcal{O}}(v))$, is precomputed. After constructing $\mathcal{O}$, given two nodes u and v in G, if u and v are in the same partition in $\mathcal{O}$, i.e., $\mathsf{wit}_{\mathcal{O}}(u) = \mathsf{wit}_{\mathcal{O}}(v)$, we compute the estimated distance, called *witness distance*, as $\overline{\mathsf{dist}}_{\mathcal{O}}(u, v) = \mathsf{dist}(u, \mathsf{wit}_{\mathcal{O}}(u)) + \mathsf{dist}(v, \mathsf{wit}_{\mathcal{O}}(v))$. If u and v are not in the same partition in $\mathcal{O}$, $\overline{\mathsf{dist}}_{\mathcal{O}}(u, v) = +\infty$.

One distance oracle is usually not enough for distance estimation in a graph G. It cannot estimate the distance of two nodes in different partitions. Even for two nodes in the same partition, the estimation may have a large error. Therefore, a set of $r = p \times \log |V|$ distance oracles $\{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r\}$ are constructed, where p can be considered as a constant[1]. The algorithm is processed in $\log |V|$ phases. In phase i ($0 \le i < \log |V|$), p distance oracles are constructed where each distance oracle contains 2^i randomly selected center nodes. Given r distance oracles, the distance of two nodes u and v in G can be estimated as an upper bound $\overline{\mathsf{dist}}(u, v) = \min_{1 \le i \le r} \overline{\mathsf{dist}}_{\mathcal{O}_i}(u, v)$.

The time complexity to compute the estimated distance $\overline{\mathsf{dist}}(u, v)$ for any two nodes u and v in a graph G is $O(\log |V|)$. The distance oracles consume $O(|V| \cdot \log |V|)$ space. Das Sarma et al. [78] prove that when $p = \Theta(|V|^{1/\log |V|})$, the estimated distance can be bounded by $\mathsf{dist}(u, v) \le \overline{\mathsf{dist}}(u, v) \le (2 \log_2 |V| - 1) \cdot \mathsf{dist}(u, v)$

[1]In [78], the set $\{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r\}$ is defined as a distance oracle.

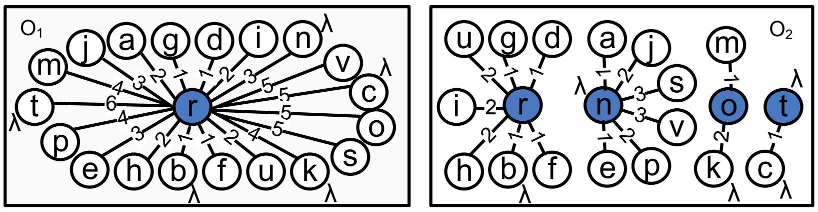


Figure 5.2: Two Distance Oracles $\mathcal{O}_1$ and $\mathcal{O}_2$

with a high probability.

Example 5.2. Fig. 5.2 shows two distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ for the graph shown in Fig. 5.1. There is one center node r in $\mathcal{O}_1$, and four center nodes r, n, o and t in $\mathcal{O}_2$. The distance of nodes j and s is estimated as $\overline{\mathsf{dist}}(j,s) = \min\{\overline{\mathsf{dist}}_{\mathcal{O}_1}(j,s), \overline{\mathsf{dist}}_{\mathcal{O}_2}(j,s)\} = \min\{\mathsf{dist}(j,r)+\mathsf{dist}(s,r), \mathsf{dist}(j,n)+\mathsf{dist}(s,n)\} = 5$.

Answering k-NK with Distance Oracle: [5] designs a Partitioned Multi-Indexing (PMI) scheme which uses a set of distance oracles to answer a k-NK query in a graph. For each partition in a distance oracle $\mathcal{O}_i$, an inverted list is constructed for each keyword in the partition. Specifically, for a partition with a center node c and a keyword λ, the inverted list contains all nodes in the partition that contain keyword λ ranked in nondecreasing order of their distances to c. Given a k-NK query $Q = (q, \lambda, k)$ and a distance oracle $\mathcal{O}_i$, the algorithm first finds the partition that q belongs to in $\mathcal{O}_i$. The result w.r.t. $\mathcal{O}_i$ is the first k elements in the inverted list for λ in the partition, denoted as $R_{\mathcal{O}_i} = \{u_1 : \mathsf{dist}(c,u_1) + \mathsf{dist}(c,q), u_2 : \mathsf{dist}(c,u_2) + \mathsf{dist}(c,q), \cdots, u_k : \mathsf{dist}(c,u_k) + \mathsf{dist}(c,q)\}$. The final result R is computed by merging the nodes in each $R_{\mathcal{O}_i}$ and maintaining k nodes with the shortest distances to q. The query time complexity is $O(k \cdot \log |V|)$. We illustrate the algorithm using the following example.

Example 5.3. Consider the graph in Fig. 5.1 and two distance oracles in Fig. 5.2. For keyword λ, the inverted list for the partition centered at node r in $\mathcal{O}_1$ has 5 elements $\{b : 1, n : 3, k : 4, c : 5, t : 6\}$. The inverted list for the partition centered at node o in $\mathcal{O}_2$ has 1 element $\{k : 2\}$. Given a k-NK query $Q = (m, \lambda, 2)$, from $\mathcal{O}_1$, we can get a result $R_{\mathcal{O}_1} = \{b : 1 + \mathsf{dist}(r,m), n : 3 + \mathsf{dist}(r,m)\} = \{b : 5, n : 7\}$, and from $\mathcal{O}_2$, we can get a result $R_{\mathcal{O}_2} = \{k : 2 + \mathsf{dist}(o, m)\} = \{k : 3\}$. By merging $R_{\mathcal{O}_1}$ and $R_{\mathcal{O}_2}$, the final answer is $R = \{k : 3, b : 5\}$. The exact answer is $R = \{c : 1, k : 1\}$ according to Fig. 5.1.

Limitation: Although in theory, the witness distance used by [5] can be bounded by a factor of $2\log_2|V| - 1$ of the exact distance with a high probability, in practice, however, we find the distance estimation error can be quite large. For example, for the graph G in Fig. 5.1 and two distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ in Fig. 5.2, for two nodes s and v, the witness distance in $\mathcal{O}_1$ is $\overline{\mathsf{dist}}_{\mathcal{O}_1}(s, v) = \mathsf{dist}(s, r) + \mathsf{dist}(v, r) = 10$, and that in $\mathcal{O}_2$ is $\overline{\mathsf{dist}}_{\mathcal{O}_2}(s, v) = \mathsf{dist}(s, n) + \mathsf{dist}(v, n) = 6$. However, the exact distance is $\mathsf{dist}(s, v) = 2$ in G, which is much smaller than both $\overline{\mathsf{dist}}_{\mathcal{O}_1}(s, v)$ and $\overline{\mathsf{dist}}_{\mathcal{O}_2}(s, v)$. The inaccurate distance estimation can greatly distort the ranking of the nodes carrying the query keyword, and thus lead to a low result quality, as illustrated in Example 5.3.

5.2.2. Exact 1-NK on a Tree

Tao et al. [83] compute the exact answer to a 1-NK query on a tree $T(V, E)$. Given a query $Q = (q, \lambda, 1)$, the result is the nearest node in T that contains keyword λ, denoted as $\mathsf{NN}(q, \lambda)$. The basic idea is as follows. We label a node v with the sequence number of v in the preorder traversal of T. For a certain keyword λ, all nodes with the preorder label in the interval $[1, |V|]$ can be partitioned into several disjointed intervals, such that any node v in the same interval shares an identical $\mathsf{NN}(v, \lambda)$. The partition is called tree Voronoi partition of λ, denoted as $\mathsf{TVP}(\lambda)$. By precomputing $\mathsf{TVP}(\lambda)$ for all keywords λ on the tree, a query $Q = (q, \lambda, 1)$ can be answered in $O(\log|V_\lambda|)$ time using a binary search in $\mathsf{TVP}(\lambda)$.

In order to compute $\mathsf{TVP}(\lambda)$ for all keywords λ in T efficiently, two new data structures, namely, Compact Tree $\mathsf{CT}(\lambda)$ and Extended Compact Tree $\mathsf{ECT}(\lambda)$, are proposed in [83].

Definition 5.2. (Compact Tree and Extended Compact Tree) For a tree T and a keyword λ, a compact tree $\mathsf{CT}(\lambda)$ is a tree that keeps only two types of nodes in

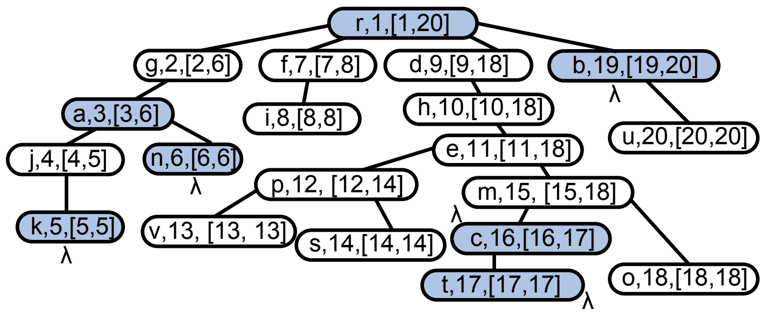
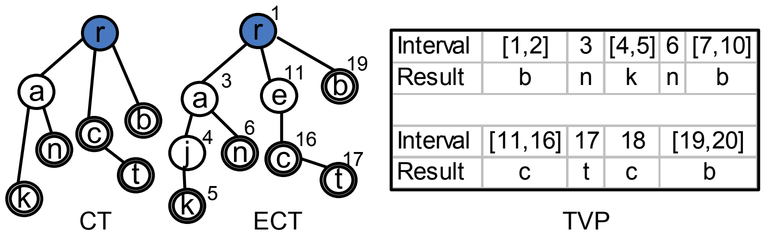


Figure 5.3: A Tree T with Preorder and Interval on Each Node



Interval	[1,2]	3	[4,5]	6	[7,10]
Result	b	n	k	n	b

Interval	[11,16]	17	18	[19,20]
Result	c	t	c	b

Figure 5.4: CT(λ), ECT(λ) and TVP(λ) for Keyword λ

T: a keyword node that contains keyword λ, and a node that has at least two direct subtrees containing nodes carrying keyword λ. In the preorder traversal of T, for two successive nodes u and v, if NN$(u, \lambda) \neq$ NN(v, λ), v is called a change node. An extended compact tree ECT(λ) is a tree constructed by adding all change nodes into the compact tree CT(λ).

Using ECT(λ), TVP(λ) can be constructed easily. In [83], the authors prove that the total size of all compact trees and all extended compact trees for all keywords in the tree $T(V, E)$ is bounded by $O(|\text{doc}(V)|)$. The time to compute all compact trees and all extended compact trees for all keywords in the tree $T(V, E)$ is bounded by $O(|\text{doc}(V)| \cdot \log |V|)$.

Example 5.4. Fig. 5.3 shows a tree with the preorder label from 1 to 20 on its nodes. For keyword λ, there are 5 keyword nodes b, c, k, n, t. For node s, NN$(s, \lambda) = c$. The compact tree of λ, CT(λ), is shown on the left part of Fig. 5.4. Node r is in CT(λ) because r has three direct subtrees with nodes carrying keyword λ. e is not in CT(λ) because e is not a keyword node and e has only one direct subtree rooted at m with nodes carrying keyword λ. The extended compact tree of λ, $\mathsf{ECT}(\lambda)$, is shown in the middle part of Fig. 5.4 with the preorder label marked beside each node. Node e is in $\mathsf{ECT}(\lambda)$, because for its parent node h, $\mathsf{NN}(h, \lambda) = b \neq \mathsf{NN}(e, \lambda) = c$. The tree Voronoi partition of λ, $\mathsf{TVP}(\lambda)$, is shown on the right part of Fig. 5.4. For node s with preorder label 14, it is in the interval $[11, 16]$, thus $\mathsf{NN}(s, \lambda) = c$ as listed in $\mathsf{TVP}(\lambda)$.

5.3. Solution Overview

Answering k-NK on a Graph using Tree Distance: To address the drawback of witness distance, in this chapter, we propose to use *tree distance* in processing a k-NK query. We observe that for a partition of a distance oracle, we can construct a shortest path tree rooted at the center node of the partition. Since a tree contains more structural information than a star, using tree distance will be more accurate than using witness distance for estimating the distance of two nodes. For a distance oracle $\mathcal{O}_i$, let the set of trees constructed in $\mathcal{O}_i$ be T_i. T_i can be considered as a tree by adding a virtual root and several virtual edges with weight $+\infty$ that connect the new virtual root to every root node in T_i respectively. Let the k-NK result on tree T be R_T. Suppose we have an algorithm to compute R_T on a tree T, we can solve the k-NK problem in a graph by merging R_{T_i} for each tree T_i, $1 \leq i \leq r$. Obviously, such a result will be more accurate than the result by [5]. The following example illustrates the k-NK query processing based on tree distance.

Example 5.5. For the distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ shown in Fig. 5.2, the corresponding shortest path trees T_1 and T_2 are shown in Fig. 5.5. For T_1, there is only 1 tree rooted at r because there is only 1 partition in $\mathcal{O}_1$. For T_2, there are 4 trees rooted at nodes n, o, r, t respectively, because there are 4 partitions in $\mathcal{O}_2$. In each tree, the path from any node to the root node is a shortest path in the original graph. For two nodes s and v, their tree distance is 2 in both T_1 and T_2, the same as the exact distance $\mathsf{dist}(s, v)$ in G. For a k-NK query $Q = (m, \lambda, 2)$, we have $R_{T_1} = \{c : 1, t : 2\}$, and $R_{T_2} = \{k : 1\}$. By merging R_{T_1} and R_{T_2}, we get $R = \{c : 1, k : 1\}$. Such a result is much better than the result in Example 5.3 computed using witness distance for the same query.

With the tree distance formulation, the key operation in answering a k-NK query on a graph is to answer the k-NK query on a tree. Therefore, we start with processing a k-NK query on a tree.

Answering k-NK on a Tree: We show that it is nontrivial to answer a k-NK query on a tree efficiently even if k is bounded. Our first attempt is to extend the existing 1-NK solution on a tree $T(V, E)$ in [83]. Recall that in [83], for a certain keyword λ, the range $[1, |V|]$ is partitioned into several disjoint intervals, and nodes with the preorder label in an identical interval share the same 1-NK result. When $k \geq 2$, each interval needs to be further partitioned to ensure that all nodes with the preorder label in the same interval share an identical k-NK result. The number of intervals increases exponentially w.r.t. the number of keyword nodes on the tree until it reaches $|V|$ for a keyword λ. Clearly, using such an approach, the index size is too large in practice even for a small k. Our second attempt is that, for each node v on the tree $T(V, E)$ and each keyword λ, we precompute its $\overline{k}$ nearest nodes that contain λ. When processing a query $Q = (q, \lambda, k)$ with $k \leq \overline{k}$, we can simply retrieve the precomputed result on node q and output the first k nodes directly. Such an approach is impractical because for each keyword λ, we need $O(\overline{k} \cdot |V|)$ space to store the precomputed results.

In the following, we first introduce two algorithms for answering exact k-NK on a tree $T(V, E)$. Our first algorithm tree-boundk can only handle bounded k values with query processing time $O(k + \log |V_\lambda|)$ and index size $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ for all keywords where $\overline{k}$ is an upper bound value of k. Our second algorithm tree-pivot can handle an arbitrary k with query processing time $O(k \cdot \log |V|)$ and index size $O(|\mathsf{doc}(V)| \cdot \log |V|)$ for all keywords which is independent of k. We then show our algorithm for approximate k-NK on a graph by merging results on a bounded number of trees. We propose a global storage technique to further reduce the index size and the query time on a graph. Finally we show how to extend our method to handle a query with multiple keywords.

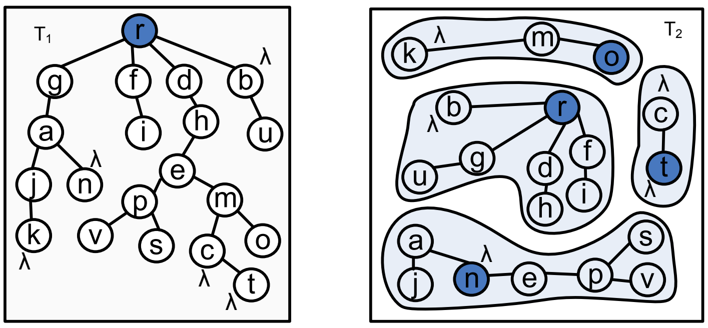


Figure 5.5: Shortest Path Trees T_1 and T_2

5.4. K-NK on a Tree for a Small K

In this subsection, we study how to answer a k-NK query $Q = (q, \lambda, k)$ on a tree $T(V, E)$. We first consider a common scenario when users are interested in a small number of answer nodes bounded by a small constant $\overline{k}$, i.e., $k \leq \overline{k}$. Recall that for a keyword λ, its compact tree $\mathsf{CT}(\lambda)$ keeps all the structural information of λ on the tree T. Our idea is to precompute the top-$\overline{k}$ results for every keyword λ and every node on $\mathsf{CT}(\lambda)$. Since the total size of all compact trees is bounded by $O(|\mathsf{doc}(V)|)$, the total space to store the top-$\overline{k}$ results of nodes on all compact trees is bounded by $O(\overline{k} \cdot |\mathsf{doc}(V)|)$. Given a query $Q = (q, \lambda, k)$, if q is on $\mathsf{CT}(\lambda)$, we can simply report the precomputed answer on $\mathsf{CT}(\lambda)$. If q is not on $\mathsf{CT}(\lambda)$, we need to find a way to construct the answer using the precomputed results as well as the structure of $\mathsf{CT}(\lambda)$ and T. In the following, we first introduce how to answer a k-NK query using $\mathsf{CT}(\lambda)$, followed by discussions on the construction of the index.

5.4.1. Query Processing

For a keyword λ, and each node v in the compact tree $\mathsf{CT}(\lambda)$, we use a *candidate list* $\mathsf{cand}_\lambda(v)$ to denote the precomputed k-NK results for $k = \overline{k}$ on node v ranked in nondecreasing order of their distances to v, in the form of $\mathsf{cand}_\lambda(v) = \{v_1 : \mathsf{dist}(v, v_1), v_2 : \mathsf{dist}(v, v_2), \cdots, v_{\overline{k}} : \mathsf{dist}(v, v_{\overline{k}})\}$ where $\mathsf{dist}(v, v_1) \leq \mathsf{dist}(v, v_2) \leq \cdots \leq \mathsf{dist}(v, v_{\overline{k}})$. Given a query $Q = (q, \lambda, k)$ on a tree $T(V, E)$ where $k \leq \overline{k}$, if q

Algorithm 5.1: tree-boundk (Q,T)

```
Input: A k-NK query $Q = (q, \lambda, k)$, and a tree $T$.
Output: Answer for $Q$ on $T$.
1 $R \leftarrow \emptyset$;
2 $(u, u') \leftarrow$ the entry edge of $q$ on $\mathsf{CT}(\lambda)$;
3 $R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(u) \oplus \mathsf{dist}(q, u))$;
4 $R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(u') \oplus \mathsf{dist}(q, u'))$;
5 return $R$;
```

is in $\mathsf{CT}(\lambda)$, we can simply report the first k elements in $\mathsf{cand}_\lambda(q)$ as the answer. The difficult case is when q is not in $\mathsf{CT}(\lambda)$. In order to answer such a query, we define an *entry edge* to be the edge in $\mathsf{CT}(\lambda)$ that is nearest to q. Intuitively, the entry edge plays a role of connecting the query node q to the compact tree $\mathsf{CT}(\lambda)$. The formal definition of entry edge is as follows.

Definition 5.3. (**Entry Node and Entry Edge**) Given a compact tree $\mathsf{CT}(\lambda)$, for each edge (u, u') on $\mathsf{CT}(\lambda)$ with u' being a child node of u, (u, u') represents a unique path from u to u' on the original tree T. For any node v on T, we say v **sticks to** $\mathsf{CT}(\lambda)$, denoted as $v \in_s \mathsf{CT}(\lambda)$, if and only if there exists an edge (u, u') on $\mathsf{CT}(\lambda)$ such that v is on the path from u to u' on T, otherwise v does not stick to $\mathsf{CT}(\lambda)$, denoted as $v \notin_s \mathsf{CT}(\lambda)$. For a node q on T, let v be the first node on the path from q to the root node of T such that $v \in_s \mathsf{CT}(\lambda)$. v is called the Entry Node of q w.r.t. λ, denoted as $\mathsf{EN}_\lambda(q)$. The corresponding edge (u, u') on $\mathsf{CT}(\lambda)$ is called the Entry Edge of q w.r.t. λ, denoted as $\mathsf{EE}_\lambda(q)$.

Note that for a node q and a keyword λ, $\mathsf{EE}_\lambda(q)$ is an edge on the compact tree $\mathsf{CT}(\lambda)$, and $\mathsf{EN}_\lambda(q)$ is a node on the original tree T. We use an example to illustrate the entry node and entry edge.

Example 5.6. For the tree T shown in Fig. 5.3 and keyword λ, the compact tree $\mathsf{CT}(\lambda)$ is shown on the left part of Fig. 5.4. For ease of illustration, we also mark the nodes in $\mathsf{CT}(\lambda)$ dark on the tree T in Fig. 5.3. For edge (r, c) in $\mathsf{CT}(\lambda)$, $h \in_s \mathsf{CT}(\lambda)$ because

h is on the path from r to c in T. $p \notin_s \mathsf{CT}(\lambda)$ since p is not on the tree path of any $\mathsf{CT}(\lambda)$ edge. For node v, its entry node is $\mathsf{EN}_\lambda(v) = e$, as e is the first node on the path (v, p, e, h, d, r) such that $e \in_s \mathsf{CT}(\lambda)$. The entry edge for v is $\mathsf{EE}_\lambda(v) = (r, c)$ since the entry node e for v is on the path from r to c in T. The entry nodes and entry edges for some other nodes in T are listed in the following table.

Node	g	j	d	e	p	u
EN_λ	g	j	d	e	e	b
EE_λ	(r, a)	(a, k)	(r, c)	(r, c)	(r, c)	(r, b)

The Algorithm: Given a tree $T(V, E)$, for keyword λ, all keyword nodes are contained in $\mathsf{CT}(\lambda)$. For any node $q \in V$, the path from q to any keyword node will go through the entry node $\mathsf{EN}_\lambda(q)$. Based on such property, the result of a query $Q = (q, \lambda, k)$ is identical with the result of the query $Q' = (\mathsf{EN}_\lambda(q), \lambda, k)$. However, $\mathsf{EN}_\lambda(q)$ may not be on $\mathsf{CT}(\lambda)$, thus the result of Q' is not necessarily precomputed. Let $(u, u') = \mathsf{EE}_\lambda(q)$, since $\mathsf{EN}_\lambda(q)$ is on the path from u to u' on the tree T, the path from $\mathsf{EN}_\lambda(q)$ to any keyword node in T will go through either u or u'. Thus, the answer for Q' can be constructed by merging the precomputed candidate lists $\mathsf{cand}_\lambda(u)$ and $\mathsf{cand}_\lambda(u')$ on $\mathsf{CT}(\lambda)$.

Our algorithm for processing a query $Q = (q, \lambda, k)$ on a tree T is shown in Algorithm 5.1. We assume that the compact tree $\mathsf{CT}(\lambda)$ for each keyword λ and the list $\mathsf{cand}_\lambda(u)$ for every node u on $\mathsf{CT}(\lambda)$ have been computed. After initializing the result R in line 1, we find the entry edge (u, u') for q on $\mathsf{CT}(\lambda)$ (line 2). We add a distance $\mathsf{dist}(q, u)$ to every node in $\mathsf{cand}_\lambda(u)$ using the $\oplus$ operator, to reflect the distance from q to a keyword node via u. We then merge the new result into R using the $\otimes_k$ operator (line 3). Similarly we apply the two operators to $\mathsf{cand}_\lambda(u')$ with the distance $\mathsf{dist}(q, u')$ (line 4). We will describe the operators $\oplus$ and $\otimes_k$ later. We use the following example to illustrate the algorithm.

Example 5.7. Given the tree T shown in Fig. 5.3 and $\mathsf{CT}(\lambda)$ on the left part of Fig. 5.4, for a query $Q = (o, \lambda, 2)$, the entry edge $\mathsf{EE}_\lambda(o) = (r, c)$. Suppose the lists $\mathsf{cand}_\lambda(r) =$

```
Algorithm 5.2: operator $R \oplus \delta$
  Input: Candidate list $R = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$, distance $\delta$.
  Output: A candidate list by adding $\delta$ to all distances in $R$.
1 $R' \leftarrow \emptyset$;
2 for $i = 1$ to $|R|$ do
3     $R' \leftarrow R' \bigcup \{u_i : d_{u_i} + \delta\}$;
4 return $R'$;
```

$\{b : 1, n : 3\}$ and $\mathsf{cand}_\lambda(c) = \{c : 0, t : 1\}$ are precomputed. By adding $\mathsf{dist}(o, r) = 5$ to $\mathsf{cand}_\lambda(r)$, and adding $\mathsf{dist}(o, c) = 2$ to $\mathsf{cand}_\lambda(c)$, we get the new lists $\{b : 6, n : 8\}$ for r and $\{c : 2, t : 3\}$ for c. We merge the two lists and get the final result $R = \{c : 2, t : 3\}$.

The efficiency of Algorithm 5.1 depends on three operations. The first operation is to find the entry edge for any node on T (line 2). The second operation is to calculate the distance of any two nodes on T, e.g., $\mathsf{dist}(q, u)$ and $\mathsf{dist}(q, u')$ (line 3-4). The third operation is to merge two sorted lists into a new one using operators $\oplus$ and $\otimes_k$ (line 3-4). Next, we discuss the three operations separately.

Finding the Entry Edge: Given a keyword λ, for any node v on a tree $T(V, E)$, our idea of finding the entry edge $\mathsf{EE}_\lambda(v)$ of v is similar to the idea of finding the 1-NK answer using the tree Voronoi partition $\mathsf{TVP}(\lambda)$ in [83]. For the range $[1, |V|]$, we partition it into several disjoint intervals, such that nodes with the preorder label in the same interval share an identical entry edge. We call such partition an *entry edge partition* for λ, denoted as $\mathsf{EEP}(\lambda)$. Given $\mathsf{EEP}(\lambda)$, $\mathsf{EE}_\lambda(v)$ can be computed easily using a binary search in $\mathsf{EEP}(\lambda)$ in $O(\log |V_\lambda|)$ time. In the next subsection, we show how to build $\mathsf{EEP}(\lambda)$ for all keywords efficiently and prove that the total size of $\mathsf{EEP}(\lambda)$ for all keywords in T is bounded by $O(\mathsf{doc}|V|)$.

Computing Tree Distance: Given a tree $T(V, E)$ with root r, suppose the distance from r to every node in T has been precomputed. For any two nodes u and v on T, we denote $\mathsf{LCA}(u, v)$ as their lowest common ancestor. The distance of u and v can be computed as $\mathsf{dist}(u, v) = \mathsf{dist}(r, u) + \mathsf{dist}(r, v) - 2\mathsf{dist}(r, \mathsf{LCA}(u, v))$. Using the techniques in [9], $\mathsf{LCA}(u, v)$ can be found in $O(1)$ time using $O(|V|)$ index space. Thus $\mathsf{dist}(u, v)$ for any two nodes u and v on T can be computed in $O(1)$ time using $O(|V|)$ index space.

Merging Results: The results are merged using two operators $\oplus$ and $\otimes_k$. Algorithm 5.2 shows the operator $\oplus$, which takes a candidate list R and a distance δ as input, and outputs a candidate list by adding δ to all distances in R. The time complexity for the $\oplus$ operator is $O(|R|)$. Algorithm 5.3 shows the operator $\otimes_k$, which takes two candidate lists R_1 and R_2 sorted in nondecreasing order of the distances, and a value k as input, and outputs the merged candidate list R. R contains at most k elements sorted in nondecreasing order of the distances. R can be constructed by visiting each element in R_1 and R_2 at most once. The time complexity for the $\otimes_k$ operator is $O(\min\{|R_1| + |R_2|, k\})$. The $\otimes_k$ and $\oplus$ operators satisfy the commutative, associative and distributive laws as follows.

(Commutative Law) $R_1 \otimes_k R_2 = R_2 \otimes_k R_1$.

(Associative Law) $(R_1 \otimes_k R_2) \otimes_k R_3 = R_1 \otimes_k (R_2 \otimes_k R_3)$.

(Distributive Law) $(R_1 \otimes_k R_2) \oplus d = (R_1 \oplus d) \otimes_k (R_2 \oplus d)$.

Theorem 5.1. *Algorithm 5.1 computes the exact* k-NK *answer for a query* $Q = (q, \lambda, k)$ *on a tree* $T(V, E)$ *in* $O(k + \log |V_\lambda|)$ *time.*

Algorithm 5.1 uses the novel idea of entry edge, and elegantly extends the 1-NK method [83] to handle k-NK ($k > 1$) with the same query time complexity, except for an extra linear cost $O(k)$ indispensable for reporting the results.

Given the tree T, the compact tree $\mathsf{CT}(\lambda)$ for every keyword λ in T can be constructed using the algorithm in [83]. Two more indexes need to be constructed. The first index is the *entry edge partition* $\mathsf{EEP}(\lambda)$ for every keyword λ, to find the entry edge for any node on T. The second index is the *candidate list* $\mathsf{cand}_\lambda(v)$ for every node on $\mathsf{CT}(\lambda)$ for each keyword λ. In the following, we introduce how to construct the two indexes separately.

Algorithm 5.3: operator $R_1 \otimes_k R_2$

```
Input: Two sorted candidate lists $R_1 = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$
       $R_2 = \{v_1 : d_{v_1}, v_2 : d_{v_2}, \cdots\}$, and result size $k$.
Output: The merged candidate list.
1 $R \leftarrow \emptyset; i \leftarrow 1; j \leftarrow 1;$
2 while $(i < |R_1|$ or $j < |R_2|)$ and $|R| \leq k$ do
3     if $i < |R_1|$ and $(d_{u_i} \leq d_{v_j}$ or $j \geq |R_2|)$ then
4         if $u_i \notin R$ then $R \leftarrow R \bigcup \{u_i : d_{u_i}\}$;
5         $i \leftarrow i + 1;$
6     else if $j < |R_2|$ and $(d_{v_j} \leq d_{u_i}$ or $i \geq |R_1|)$ then
7         if $v_j \notin R$ then $R \leftarrow R \bigcup \{v_j : d_{v_j}\}$;
8         $j \leftarrow j + 1;$
9 return $R$;
```

5.4.2. Construction of Entry Edge Partition

Given a tree $T(V, E)$, for each keyword λ, sharing the similar idea with the tree Voronoi partition $\mathsf{TVP}(\lambda)$, we construct an entry edge partition $\mathsf{EEP}(\lambda)$, which divides $[1, |V|]$ into several disjoint intervals, such that nodes in V with preorder in the same interval share an identical entry edge on $\mathsf{CT}(\lambda)$. In order to construct the entry edge partition, for each edge (u, u') on $\mathsf{CT}(\lambda)$, we label (u, u') with an interval according to the following definition.

Definition 5.4. (Labeled Compact Tree) Given a tree T, a node v on T has an interval $[s_v, t_v]$ where s_v is the preorder label of v on T and t_v is the maximum preorder label for all nodes in the subtree rooted at v. Given a compact tree $\mathsf{CT}(\lambda)$, for any edge (u, u') on $\mathsf{CT}(\lambda)$, let the branching node of (u, u') be the first node along the path from u to u' on T, and denote it as u_b. We label edge (u, u') with the interval of u_b.

The label of every edge on a compact tree $\mathsf{CT}(\lambda)$ can be computed easily when constructing $\mathsf{CT}(\lambda)$. Given any node v on a tree T and an edge (u, u') on a compact tree $\mathsf{CT}(\lambda)$, denote the branching node of (u, u') as u_b, then v is in the subtree rooted at u_b if and only if the preorder label of v on T is in the interval of u_b, which is identical with the label of edge (u, u'). For ease of presentation, for each labeled compact tree $\mathsf{CT}(\lambda)$, we add a virtual root ϕ and an edge from ϕ to the original root of $\mathsf{CT}(\lambda)$. We use the following example to illustrate the labeled compact tree.

Example 5.8. For the tree T shown in Fig. 5.3, we mark the preorder and the interval of each node on the tree. For the node h, its interval is $[10, 18]$ because the preorder of h on T is 10 and the maximum preorder for all nodes on the subtree rooted at h is 18. The labeled compact tree $\mathsf{CT}(\lambda)$ for keyword λ is shown on the left part of Fig. 5.6. For the edge (r, c) on $\mathsf{CT}(\lambda)$, its branching node is d because d is the first node along the path (r, d, h, e, m, c) on T. The label of edge (r, c) is the interval of node d, which is $[9, 18]$.

For a compact tree $\mathsf{CT}(\lambda)$ of tree T and a keyword λ, suppose (u, u') on $\mathsf{CT}(\lambda)$ is an entry edge of a node v on tree T, i.e., $\mathsf{EE}_\lambda(v) = (u, u')$. The preorder of v is in the interval of (u, u'), because the interval of (u, u') contains all nodes under the subtree rooted at the branching node of (u, u'). Based on such an observation, by excluding the intervals of all edges under the subtree rooted at u' in $\mathsf{CT}(\lambda)$ from the interval of (u, u'), nodes with preorder in the remaining intervals will use (u, u') as the entry edge. For example, in the compact tree $\mathsf{CT}(\lambda)$ shown in Fig. 5.6, the edge (ϕ, r) has an interval $[1, 20]$. r has three branches with intervals $[2, 6]$, $[9, 18]$ and $[19, 20]$ respectively. By excluding the three intervals from $[1, 20]$, two intervals $[1, 1]$ and $[7, 8]$ are left. Thus nodes with preorder in either of the two intervals $[1, 1]$ and $[7, 8]$ share the same entry edge (ϕ, r). For edge (r, c) with interval $[9, 18]$, by excluding interval $[17, 17]$ of the only branch of c, nodes with preorder in either of the two intervals $[9, 16]$ and $[18, 18]$ share the same entry edge (r, c).

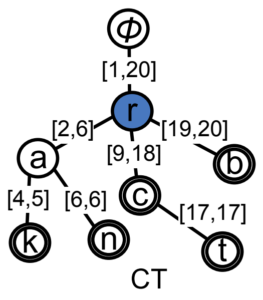
Algorithm 5.4 shows the construction of the entry edge partition $\mathsf{EEP}(\lambda)$ on $\mathsf{CT}(\lambda)$ for a keyword λ. After initializing $\mathsf{EEP}(\lambda)$ (line 2), the main operation is a recursive procedure partition (line 3), to partition the interval $[1, |V|]$ to several disjoint intervals. Each entry in $\mathsf{EEP}(\lambda)$ is in the form of $([s, t], (u, u'))$ denoting that nodes with the

Algorithm 5.4: EEP-construct (T,CT(λ))

```
Input: A tree $T(V, E)$ and a labelled compact tree CT($\lambda$).
Output: Entry edge partition EEP($\lambda$).
1  $r \leftarrow$ the original root of CT($\lambda$);
2  EEP($\lambda$) $\leftarrow \emptyset$;
3  partition(EEP($\lambda$), $[1, |V|]$, $(\phi, r)$, CT($\lambda$));
4  return EEP($\lambda$);
5  Procedure partition(EEP($\lambda$), interval $[s, t]$, edge $(u, u')$, CT($\lambda$))
6  foreach subnode $u''$ of $u'$ on CT($\lambda$) in increasing preorder do
7      $[s', t'] \leftarrow$ interval of $(u', u'')$;
8      if $s < s'$ then add $([s, s' - 1], (u, u'))$ to EEP($\lambda$);
9      partition(EEP($\lambda$), $[s', t']$, $(u', u'')$, CT($\lambda$));
10     $s \leftarrow t' + 1$;
11 if $s \leq t$ then add $([s, t], (u, u'))$ to EEP($\lambda$);
```

preorder label in the interval $[s, t]$ share the same entry edge (u, u'). For an edge (u, u') with interval $[s, t]$, the procedure processes every child node u'' of u' on CT(λ) in increasing preorder of u'' (line 6). For each edge (u', u'') with interval $[s', t']$, the interval $[s, t]$ is partitioned into three parts: $[s, s' - 1]$, $[s', t']$ and $[t' + 1, t]$. The first part is added to EEP(λ) with the entry edge (u, u') if it is not empty (line 8). The second part is processed recursively for edge (u', u'') (line 9), and the third part is left to be further partitioned by other child nodes of u' by simply setting s to be $t' + 1$ (line 10). After processing all child nodes of u', if $[s, t]$ is still not empty, we add $[s, t]$ to EEP(λ) with the entry edge (u, u') (line 11).

The time complexity of Algorithm 5.4 is $O(|V(\mathsf{CT}(\lambda))|)$ since every node on CT(λ) is visited once. For each edge (u, u') on CT(λ), at most two intervals are added into EEP(λ). One is added before invoking partition for edge (u, u') (line 8) and the other is added at the end of partition for (u, u') (line 11). Thus the total number of intervals in EEP(λ) is no more than $2 \times |V(\mathsf{CT}(\lambda))|$.



Interval	[1, 1]	[2, 3]	[4, 5]	[6, 6]	[7, 8]
EntryEdge	(Φ,r)	(r,a)	(a,k)	(a,n)	(Φ,r)
Interval	[9,16]	[17,17]	[18,18]	[19,20]	
EntryEdge	(r,c)	(c,t)	(r,c)	(r,b)	

Entry Edge Partition

Figure 5.6: Labeled Compact Tree and Entry Edge Partition

Example 5.9. For the labeled compact tree $\mathsf{CT}(\lambda)$ shown in Fig. 5.6, when invoking $\mathsf{partition}(\mathsf{EEP}(\lambda), [1, 20], (\phi, r), \mathsf{CT}(\lambda))$, we process the three child nodes a, c, b of r in order. We first process edge (r, a) with interval $[2, 6]$, which divides the interval $[1, 20]$ into three parts: $[1, 1]$, $[2, 6]$, and $[7, 20]$. $[1, 1]$ is added into $\mathsf{EEP}(\lambda)$ with the entry edge (ϕ, r). $[2, 6]$ is processed recursively by invoking $\mathsf{partition}(\mathsf{EEP}(\lambda), [2, 6], (r, a), \mathsf{CT}(\lambda))$, and $[7, 20]$ is processed by the other two child nodes c and b similarly. $\mathsf{EEP}(\lambda)$ is shown on the right part of Fig. 5.6.

Theorem 5.2. *For a tree $T(V, E)$ with the compact trees for all keywords constructed, the edge entry partition $\mathsf{EEP}(\lambda)$ for all keywords can be constructed in $O(|\mathsf{doc}(V)|)$ time and stored in $O(|\mathsf{doc}(V)|)$ space.*

5.4.3. Construction of Candidate List

Given a compact tree $\mathsf{CT}(\lambda)$ for a tree T and a keyword λ, we need to compute the candidate list $\mathsf{cand}_\lambda(v)$ for every node v on $\mathsf{CT}(\lambda)$. Since $\mathsf{CT}(\lambda)$ keeps the structural information of all keyword nodes in T, it is sufficient to search only on $\mathsf{CT}(\lambda)$ to calculate $\mathsf{cand}_\lambda(v)$. A naive solution is to compute each $\mathsf{cand}_\lambda(v)$ separately on $\mathsf{CT}(\lambda)$. This approach may take $O(|V(\mathsf{CT}(\lambda))|)$ time to calculate $\mathsf{cand}_\lambda(v)$ for a node v, thus $O(|V(\mathsf{CT}(\lambda))|^2)$ time to calculate all candidate lists in $\mathsf{CT}(\lambda)$ for one keyword λ, which is too slow.

In order to save the computational cost, we design a novel method to update the candidate list of a node using those of its nearby nodes on the tree $\mathsf{CT}(\lambda)$. Note

Algorithm 5.5: cand-construct (T,$\mathsf{CT}(\lambda)$, $\overline{k}$)

```
Input: A tree $T$, a compact tree $\mathsf{CT}(\lambda)$, and the upper bound of $k$, $\overline{k}$.
Output: $\mathsf{cand}_\lambda(v)$ for each $v$ on $\mathsf{CT}(\lambda)$.
1 $\mathsf{cand}_\lambda(v) \leftarrow \emptyset$ for each node $v$ on $\mathsf{CT}(\lambda)$;
2 $\mathsf{cand}_\lambda(v) \leftarrow \{v : 0\}$ for each node $v$ on $\mathsf{CT}(\lambda)$ that contains $\lambda$;
3 foreach $v$ on $\mathsf{CT}(\lambda)$ in a bottom-up fashion do
4     $u \leftarrow$ the parent node of $v$ on $\mathsf{CT}(\lambda)$;
5     $\mathsf{cand}_\lambda(u) \leftarrow \mathsf{cand}_\lambda(u) \otimes_{\overline{k}} (\mathsf{cand}_\lambda(v) \oplus \mathsf{dist}(u, v))$;
6 foreach $v$ on $\mathsf{CT}(\lambda)$ in a top-down fashion do
7     $u \leftarrow$ the parent node of $v$ on $\mathsf{CT}(\lambda)$;
8     $\mathsf{cand}_\lambda(v) \leftarrow \mathsf{cand}_\lambda(v) \otimes_{\overline{k}} (\mathsf{cand}_\lambda(u) \oplus \mathsf{dist}(u, v))$;
```

that in $\mathsf{CT}(\lambda)$, the path between two nodes u, v is unique: from node u to the lowest common ancestor of u and v, $\mathsf{LCA}(u, v)$, and then from $\mathsf{LCA}(u, v)$ to v. Based on this observation, we can follow the path to propagate the candidate list on u to v. Using this idea, we just need to traverse the tree $\mathsf{CT}(\lambda)$ twice to build the candidate lists for all nodes on $\mathsf{CT}(\lambda)$. The first traversal on $\mathsf{CT}(\lambda)$ is a bottom-up one, such that the candidate list on each node is propagated to all its ancestors on $\mathsf{CT}(\lambda)$. The second traversal on $\mathsf{CT}(\lambda)$ is a top-down one, such that the candidate list on each node is further propagated to all its descendants.

Algorithm 5.5 shows the construction of the candidate lists on $\mathsf{CT}(\lambda)$. We first initialize the candidate list for each keyword node to be the node itself and initialize the candidate list for each non-keyword node to be $\emptyset$ (line 1-2). We then traverse $\mathsf{CT}(\lambda)$ in a bottom-up fashion, e.g., using postorder traversal. For each node v traversed, we merge $\mathsf{cand}_\lambda(v)$ into that of its parent node u by adding a distance $\mathsf{dist}(u, v)$ to the list $\mathsf{cand}_\lambda(v)$ (line 3-5). At last, we traverse $\mathsf{CT}(\lambda)$ in a top-down fashion, e.g., using preorder traversal. For each node v traversed, we merge the list of v's parent node u, $\mathsf{cand}_\lambda(u)$, into that of v by adding a distance $\mathsf{dist}(u, v)$ to the list $\mathsf{cand}_\lambda(u)$ (line 6-8). Since the $\otimes_{\overline{k}}$ operator takes $O(\overline{k})$ time, the time complexity of Algorithm 5.5 is

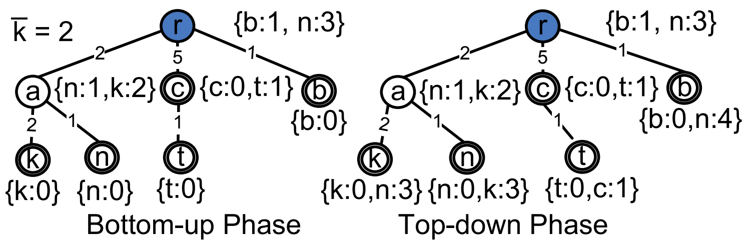


Figure 5.7: Constructing Candidate Lists

$O(\overline{k} \cdot |V(\mathsf{CT}(\lambda))|)$ using $O(\overline{k} \cdot |V(\mathsf{CT}(\lambda))|)$ space.

Example 5.10. Fig. 5.7 shows the candidate lists after the bottom-up phase and the top-down phase for the compact tree $\mathsf{CT}(\lambda)$ shown on the left part of Fig. 5.4. Initially, the candidate list for t is $\{t : 0\}$ and the candidate list for c is $\{c : 0\}$. Since c is a parent node of t, in the bottom-up phase, the list of t is propagated and merged into that of c by adding a distance $\mathsf{dist}(c, t) = 1$, thus $\mathsf{cand}_\lambda(c) = \{c : 0, t : 1\}$ after the bottom-up phase. In the top-down phase, the list of c is propagated and merged into that of t, thus $\mathsf{cand}_\lambda(t) = \{t : 0, c : 1\}$ after the top-down phase.

Theorem 5.3. *Given a tree T, an upper bound of k, $\overline{k}$, and $\mathsf{CT}(\lambda)$ for all keywords λ, the candidate lists $\mathsf{cand}_\lambda(v)$ for all keywords λ and all nodes v on $\mathsf{CT}(\lambda)$ can be constructed in $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ time and stored in $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ space.*

5.5. K-NK on a Tree for a Large K

Algorithm 5.1 can only process a k-NK query $Q = (q, \lambda, k)$ with a bounded k, i.e., $k \leq \overline{k}$, on a tree T. If k can be arbitrarily large, the index size cannot be bounded. In this subsection, we will remove the restriction on k and introduce an algorithm to handle a k-NK query for an arbitrary k, with an index size independent of k.

5.5.1. A Basic Pivot Approach

Recall that for a node u that contains keyword λ and an arbitrary node v in a tree T, the path from v to u is unique on T, and can be divided into two segments: the first segment is from v to their lowest common ancestor $\mathsf{LCA}(u, v)$, and the second segment is from $\mathsf{LCA}(u, v)$ to u. Our basic idea is to compute the first segment online and precompute the results regarding the second segment offline. Thus, in the precomputing phase, instead of propagating a keyword node u to all nodes in T to update their candidate lists, we just need to propagate u to its ancestors in T. In the query processing phase, we do not search the whole tree to get the answer for a query, but instead, we just need to merge the precomputed candidates along the path from the query node to the root node of the tree T. Using this method, the size of the index to keep the candidate nodes can be largely reduced at the expense of longer query processing time.

We use $\mathsf{depth}(T)$ to denote the depth of tree T, and $\mathsf{depth}(u, T)$ to denote the depth of node u on tree T. For any two nodes u and v on T, u is a pivot of v if and only if u is an ancestor of v on T. For each node v, we denote the set of pivots of v on T as $\mathsf{PV}(v, T)$. We have $|\mathsf{PV}(v, T)| = \mathsf{depth}(v, T)$. Given a keyword λ, for each node u on tree T, we use the candidate list $\mathsf{cand}_\lambda(u)$ to denote the set of nodes that contain keyword λ on the subtree rooted at u on tree T, sorted in nondecreasing order of their distances to u. The candidate list is in the form of $\mathsf{cand}_\lambda(u) = \{u_1 : \mathsf{dist}_T(u, u_1), u_2 : \mathsf{dist}_T(u, u_2), \cdots\}$ where $\mathsf{dist}_T(u, u_1) \leq \mathsf{dist}_T(u, u_2) \leq \cdots$. In order to handle an arbitrary k, the size of $\mathsf{cand}_\lambda(u)$ is not bounded by any predefined $\overline{k}$. Clearly, a node $v \in \mathsf{cand}_\lambda(u)$ if and only if v contains keyword λ and $u \in \mathsf{PV}(v, T)$. In other words, a keyword node v only appears in the candidate lists of its pivots. As a result, for any keyword λ, the total size of all candidate lists for λ is $\sum_{v \in V_\lambda} |\mathsf{PV}(v, T)| = \sum_{v \in V_\lambda} \mathsf{depth}(v, T)$. We use the following example to illustrate the pivot based approach.

Example 5.11. Fig. 5.8 shows a tree T with $\mathsf{depth}(T) = 6$. For keyword λ, the nodes

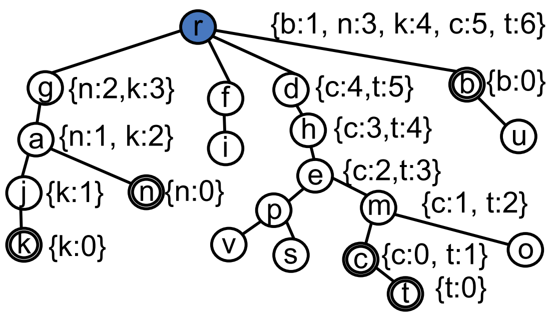


Figure 5.8: Basic Pivot Approach

that contain λ are marked with bold circles. For every node v, we create a candidate list $\mathsf{cand}_\lambda(v)$ that contains all keyword nodes in its subtree, sorted in nondecreasing distances to v. For example, $\mathsf{cand}_\lambda(g) = \{n : 2, k : 3\}$ means there are two keyword nodes n and k in the subtree rooted at g with distances 2 and 3 to g respectively. For node p, $\mathsf{PV}(p, T) = \{r, d, h, e\}$. For a k-NK query $Q = (d, \lambda, 3)$, the path from d to the root r contains two nodes d and r. We merge the lists $\mathsf{cand}_\lambda(d)$ and $\mathsf{cand}_\lambda(r)$ by adding a distance $\mathsf{dist}(r, d) = 1$ to all elements in $\mathsf{cand}_\lambda(r)$. The final answer for Q is $\{b : 2, c : 4, n : 4\}$.

5.5.2. Pivot Approach with Tree Balancing

The problem is not perfectly solved using the basic pivot approach above. The reasons are twofold. First, in the precomputing phase, the index size for each keyword λ is $\sum_{v\in V_\lambda} \mathsf{depth}(v, T)$, which can be large if $\mathsf{depth}(v, T)$ is large. Second, when processing a query $Q = (q, \lambda, k)$, we need to traverse all nodes from the query node q to the root of T. This is also costly if $\mathsf{depth}(q, T)$ is large. Thus the key to optimizing both index space and query time is to reduce the average depth of nodes on the tree. A simple solution is to rotate the tree T to find a proper root such that the average depth of nodes is minimized. However, such an approach cannot essentially solve the problem, as illustrated by the following example. Let $T(V, E)$ be a chain of $2n + 1$ nodes where every node contains keyword λ. The best way is to select the middle node on the chain as the root to minimize the average depth of nodes. The total index

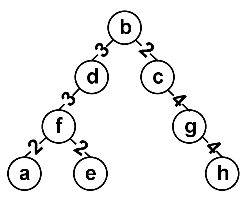
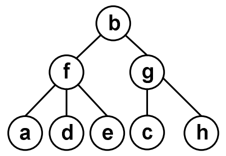
Original Tree T	$\mathsf{DT}(T)$	v	$\mathsf{PV}(v, \mathsf{DT}(T))$	v	$\mathsf{PV}(v, \mathsf{DT}(T))$
 b d c f g a e h 3 2 3 4 2 2 4		a	$\{b:8, f:2\}$	d	$\{b:3, f:3\}$
		e	$\{b:8, f:2\}$	c	$\{b:2, g:4\}$
		h	$\{b:10, g:4\}$	f	$\{b:6\}$
		g	$\{b:6\}$		

Figure 5.9: Distance Preserving Balanced Tree

size is $\sum_{v \in V_\lambda} \mathsf{depth}(v,T) = \sum_{v \in V(T)} \mathsf{depth}(v,T) = n(n-1)$, which is $O(n^2)$. Furthermore, we need to traverse n nodes to answer a query when the query node q is at one end of the chain, leading to $O(n)$ query time. This example shows that both the index space and query processing can still be very costly, even though we rotate the tree.

In order to reduce the average depth of nodes to optimize both index space and query processing time, we introduce a new structure called distance preserving balanced tree for $T(V,E)$, denoted as $\mathsf{DT}(T)$. Generally speaking, $\mathsf{DT}(T)$ preserves all distance information for any node pair on T and the height of $\mathsf{DT}(T)$ is at most $\log_2 |V|$. The formal definition of $\mathsf{DT}(T)$ is as follows.

Definition 5.5. (Distance Preserving Balanced Tree) Given a tree $T(V,E)$ with a positive weight on each edge, a Distance Preserving Balanced Tree of T, denoted as $\mathsf{DT}(T)$, is an unweighted tree with the following three properties.

P_1: $V(\mathsf{DT}(T)) = V(T)$.

P_2: $\mathsf{depth}(\mathsf{DT}(T)) \leq \log_2 |V|$.

P_3: For any two nodes u and v, let the lowest common ancestor of u and v on $\mathsf{DT}(T)$ be $o = \mathsf{LCA}_{\mathsf{DT}(T)}(u,v)$. The following equation always holds: $\mathsf{dist}_T(u,v) = \mathsf{dist}_T(u,o) + \mathsf{dist}_T(v,o)$.

Note that $\mathsf{DT}(T)$ is unweighted and the distances $\mathsf{dist}_T(u,v)$, $\mathsf{dist}_T(u,o)$ and $\mathsf{dist}_T(v,o)$ in P_3 are calculated on the original tree T, but not $\mathsf{DT}(T)$. The lowest common ancestor $\mathsf{LCA}_{\mathsf{DT}(T)}(u,v)$ is not necessarily the ancestor of u or v on the original tree T. Based on P_3, we can also divide our algorithm into two phases using

$\mathsf{DT}(T)$. In the preprocessing phase, for each keyword λ, and each node v that contains keyword λ, we propagate v into the candidate lists of its pivots on $\mathsf{DT}(T)$. In the query processing phase, we traverse from the query node q to the root node on $\mathsf{DT}(T)$. Using the balanced tree $\mathsf{DT}(T)$, the total size of the candidate lists for a keyword λ is bounded by $\sum_{v \in V_\lambda} \mathsf{depth}(v, \mathsf{DT}(T)) \leq \sum_{v \in V_\lambda} \log_2 |V|$, and the total size for all keywords is bounded by $O(|\mathsf{doc}(V)| \cdot \log |V|)$. For processing a query, we need to traverse at most $\log_2 |V| + 1$ nodes on the path from the query node to the root of $\mathsf{DT}(T)$.

Example 5.12. A tree T with $\mathsf{depth}(T) = 3$ and a distance preserving balanced tree of T, $\mathsf{DT}(T)$ with $\mathsf{depth}(\mathsf{DT}(T)) = 2$ are shown in Fig. 5.9. The weight of each edge is marked on T. Edge (b, d) is on T but not on $\mathsf{DT}(T)$, and edge (b, f) is on $\mathsf{DT}(T)$ but not on T. For two nodes a and d, $\mathsf{LCA}_{\mathsf{DT}(T)}(a, d) = f$, thus $\mathsf{dist}_T(a, d) = \mathsf{dist}_T(a, f) + \mathsf{dist}_T(d, f) = 2 + 3 = 5$. Note that f is not an ancestor of d on the original tree T. $\mathsf{PV}(v, \mathsf{DT}(T))$ for each node v in $\mathsf{DT}(T)$ is listed on the right part of Fig. 5.9.

Here we introduce our algorithm of processing a $\mathsf{k\text{-}NK}$ query on a tree T using $\mathsf{DT}(T)$, and in the next subsection, we will show that $\mathsf{DT}(T)$ always exists for any tree T. We will also describe how to construct $\mathsf{DT}(T)$ for a tree T and how to compute all candidate lists $\mathsf{cand}_\lambda(v)$ for all keywords λ and all nodes v on the tree $\mathsf{DT}(T)$.

Query Processing: Given a tree T and $\mathsf{DT}(T)$, Algorithm 5.6 shows how to process a query $Q = (q, \lambda, k)$. We traverse all nodes on the path from q to the root of $\mathsf{DT}(T)$, which is $\mathsf{PV}(q, \mathsf{DT}(T)) \bigcup \{q\}$ (line 2). For each traversed node v, we add $\mathsf{dist}_T(q, v)$ to all elements in $\mathsf{cand}_\lambda(v)$ and then merge the list into the current result R, since we need to first go from node q to node v (the first segment), and then go from v to the keyword nodes in $\mathsf{cand}_\lambda(v)$ (the second segment). Note that the time complexity of the $\oplus$ operator in line 3 is $O(|\mathsf{cand}_\lambda(v)|)$. However, by combining $\oplus$ with $\otimes_k$, it is easy to reduce the time complexity of line 3 to $O(k)$.

Example 5.13. Fig. 5.10 shows a distance preserving balanced tree $\mathsf{DT}(T)$ for the tree T shown in Fig. 5.8, with depth 4. For keyword λ, the nodes that contain λ are

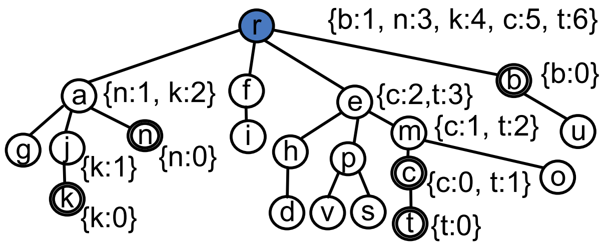


Figure 5.10: Pivot Approach with Tree Balancing

Algorithm 5.6: tree-pivot (Q,T)

```
Input: A k-NK query $Q = (q, \lambda, k)$, and a tree $T$.
Output: Answer for $Q$ on $T$.
1 $R \leftarrow \emptyset$;
2 foreach $v \in \mathsf{PV}(q, \mathsf{DT}(T)) \bigcup \{q\}$ do
3     $R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(v) \oplus \mathsf{dist}_T(q, v))$;
4 return $R$;
```

marked with bold circles in Fig. 5.10. For a query $Q = (e, \lambda, 3)$, we just need to merge 2 candidate lists $\mathsf{cand}_\lambda(e)$ and $\mathsf{cand}_\lambda(r)$ by adding a distance $\mathsf{dist}_T(e, r) = 3$ to all elements in $\mathsf{cand}_\lambda(r)$. However, if we use the basic pivot approach on the original tree T without tree balancing, we need to merge 4 candidate lists for nodes e, h, d and r respectively. The answer for Q is $\{c : 2, t : 3, b : 4\}$.

Theorem 5.4. *The time complexity for answering a* k-NK *query on a tree $T(V, E)$ using Algorithm 5.6 is $O(k \cdot \log |V|)$.*

5.5.3. Index Construction

Given a tree T, in order to answer a query $Q = (q, \lambda, k)$ using Algorithm 5.6, we need to build two indexes. The first index is the distance preserving balanced tree $\mathsf{DT}(T)$ for T and the second index is the candidate list $\mathsf{cand}_\lambda(v)$ for each keyword λ and each node v on $\mathsf{DT}(T)$. We introduce them separately in the following.

Constructing $\mathsf{DT}(T)$**:** Before introducing how to construct a tree $\mathsf{DT}(T)$ to satisfy the three properties in Definition 5.5, we first present an approach to constructing a tree T' from T, which satisfies properties P_1 and P_3. In other words, T' is distance preserving but not necessarily balanced. Let the initial T' be T. We change T' by performing the following steps.

(1) Randomly select a node r on T' as the new root and rotate T' accordingly.

(2) For each direct subtree T'_c of r on T', perform steps (1) and (2) on T'_c recursively.

Clearly, after steps (1) and (2), T' may not be isomorphic to T. We have the following two observations on T'. O_1: After performing step (1) on T', two nodes u and v are in different direct subtrees of r if and only if $\mathsf{LCA}_{T'}(u, v) = r$. Such a property also holds after performing step (2) on T' because step (2) only changes the structure within a subtree of r. O_2: Since the structure of T' is not changed after step (1), we have $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, r) + \mathsf{dist}_T(v, r)$ on the original tree T. From O_1 and O_2, we have $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, \mathsf{LCA}_{T'}(u, v)) + \mathsf{dist}_T(v, \mathsf{LCA}_{T'}(u, v))$ after step (2) on T'. Such a property also holds for any subtree of T' because it is processed using steps (1) and (2) recursively. As a result, T' satisfies property P_3.

Our $\mathsf{DT}(T)$ is constructed in a similar way as T'. In order to construct a balanced tree, in step (1), the root node r should be selected more carefully, instead of random selection. In our method, we select a *median node* to be the root node in step (1), which is defined as follows.

Definition 5.6. (Median Node) Given a tree T, the Median Node of T is a node r on T such that when using r as the root of T, for each direct subtree T_c of r on T, $|V(T_c)| \leq \frac{|V(T)|}{2}$ holds.

The median node r is used to balance the size of each direct subtree of T when using r as the root of T, as a direct subtree of r in T contains at most half of the nodes in T. Clearly, if a median node always exists for any tree, we can select a median node of tree T as the root and recursively do this for each direct subtree of the root. In this way we can construct a tree T' with $\mathsf{depth}(T') \leq \log_2 |V(T)|$. The following lemma

```
Algorithm 5.7: DT-construct ($T$)
  Input: A tree $T$.
  Output: A distance preserving balanced tree $\mathsf{DT}(T)$.
1 $r \leftarrow$ the median node of $T$ ;
2 rotate $T$ with $r$ as the root;
3 $\mathsf{DT}(T) \leftarrow$ a tree with a single node $r$;
4 foreach direct subtree $T_i$ of $r$ in $T$ do
5     $\mathsf{DT}(T_i) \leftarrow$ DT-construct($T_i$);
6     add $\mathsf{DT}(T_i)$ as a subtree of $r$ in $\mathsf{DT}(T)$;
7 return $\mathsf{DT}(T)$;
```

shows that the median node always exists on any tree T, and also gives a method to find the median node of T.

Lemma 5.1. *Given a tree T, the median node of T is the node r, such that the subtree rooted at r contains more than $\frac{|V(T)|}{2}$ nodes and* $\mathsf{depth}(r, T)$ *is the maximum.*

According to Lemma 5.1, the median node r is unique on T. Otherwise if there are two such nodes with the same maximum depth, the size of the tree will be larger than $|V(T)|$. Given a tree T, we can easily find the median node of T using time $O(|V(T)|)$ by traversing each node in T only once.

Algorithm 5.7 shows how to construct $\mathsf{DT}(T)$ for a tree T. Specifically, given a tree T, we first find the median node r of T as the new root and then rotate T accordingly (line 1-2). The median node r is also the root of $\mathsf{DT}(T)$ (line 3). For each direct subtree T_i of r in T, we create $\mathsf{DT}(T_i)$ recursively and add $\mathsf{DT}(T_i)$ as a subtree of $\mathsf{DT}(T)$ (line 4-6).

Example 5.14. For the tree T shown in Fig. 5.8, $\mathsf{DT}(T)$ is shown in Fig. 5.10. $\mathsf{DT}(T)$ is constructed as follows. Since r is the median node of T, the root of $\mathsf{DT}(T)$ is r. For the first subtree under r in T, its median node is a, thus the first subtree under r in $\mathsf{DT}(T)$ is rooted at a. All other nodes in $\mathsf{DT}(T)$ are constructed similarly. We have $\mathsf{depth}(\mathsf{DT}(T)) = 4 \leq \log_2 |V(T)| = \log_2 20$.

Theorem 5.5. *Given a tree $T(V, E)$, Algorithm 5.7 constructs a distance preserving balanced tree $\mathsf{DT}(T)$ for T using $O(|V| \cdot \log |V|)$ time and $O(|V|)$ space.*

Constructing $\mathsf{cand}_\lambda(v)$: For a tree $T(V, E)$, given $\mathsf{DT}(T)$, the algorithm for constructing the candidate list $\mathsf{cand}_\lambda(v)$ for each node v and each keyword λ is quite simple. For each node v, we propagate its keyword information to all its pivots in $\mathsf{DT}(T)$. Our algorithm is shown in Algorithm 5.8. We first initialize every candidate list to be $\emptyset$ (line 1). Then we traverse each node v in $\mathsf{DT}(T)$ and each keyword λ that is contained in node v (line 2-3). For each pivot p of v as well as v itself, we calculate $\mathsf{dist}_T(p, v)$ on the original tree T, and add the element $v : \mathsf{dist}_T(p, v)$ to the candidate list $\mathsf{cand}_\lambda(p)$ (line 4-5). After all candidate lists are created, we sort the elements in every candidate list in nondecreasing order of the distances. The time complexity for line 2-5 is $O(|\mathsf{doc}(V)| \cdot \log |V|)$ since each keyword is propagated into at most $\log |V|$ candidate lists in $\mathsf{DT}(T)$. For line 6-7, we need $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ time to sort all candidate lists in $\mathsf{DT}(T)$.

Theorem 5.6. *For a tree T, Algorithm 5.8 computes the candidate lists $\mathsf{cand}_\lambda(v)$ for all nodes v and all keywords λ on $\mathsf{DT}(T)$ using $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ time and $O(|\mathsf{doc}(V)| \cdot \log |V|)$ space.*

5.6. Approximate K-NK on a Graph

In this subsection, we discuss how to answer a k-NK query $Q = (q, \lambda, k)$ on a graph G. We introduce two algorithms graph-boundk and graph-pivot for a bounded k and an arbitrary k respectively. We then propose a global storage technique to reduce the index size and query processing time. We also show how our approach can be extended to handle multiple keywords. Finally, we summarize the complexities of all algorithms introduced in this chapter.

Query Processing: Our general idea for query processing on a graph is introduced in Chapter 5.3. Suppose we have computed $r = O(\log |V|)$ distance oracles

```
Algorithm 5.8: cand-construct ($T$,$\mathsf{DT}(T)$)
  Input: A tree $T$, a distance preserving balanced tree $\mathsf{DT}(T)$.
  Output: $\mathsf{cand}_\lambda(v)$ for each $v$ on $\mathsf{DT}(T)$ and each keyword $\lambda$.
1 $\mathsf{cand}_\lambda(v) \leftarrow \emptyset$ for each node $v$ on $\mathsf{DT}(T)$ and each keyword $\lambda$;
2 foreach $v \in V(\mathsf{DT}(T))$ do
3     foreach $\lambda \in \mathsf{doc}(v)$ do
4         foreach $p \in \mathsf{PV}(v, \mathsf{DT}(T)) \bigcup \{v\}$ do
5             $\mathsf{cand}_\lambda(p) \leftarrow \mathsf{cand}_\lambda(p) \bigcup \{v : \mathsf{dist}_T(p, v)\}$;
6 foreach $v \in V(\mathsf{DT}(T))$ and keyword $\lambda$ do
7     sort elements in $\mathsf{cand}_\lambda(v)$ in nondecreasing order of distances;
```

```
Algorithm 5.9: graph-knk ($G$,$Q$)
  Input: A graph $G(V, E)$ and a k-NK query $Q = (q, \lambda, k)$.
  Output: The answer for $Q$ on $G$.
1 $R \leftarrow \emptyset$;
2 foreach Distance Oracle $\mathcal{O}_i$ do
3     $T_i \leftarrow$ shortest path tree for $\mathcal{O}_i$;
4     $R \leftarrow R \otimes_k \mathsf{tree\text{-}knk}(T_i, Q)$;
5 return $R$;
```

$\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r$ using the algorithm in [78]. Let the shortest path trees for the oracles be $T_1, T_2, \cdots, T_r$ respectively. Algorithm 5.9 shows our framework for answering Q on G. The algorithm simply enumerates all shortest path trees and answers the k-NK query using a tree based approach, denoted as tree-knk, on each shortest path tree T_i, and merges all the results using the $\otimes_k$ operator (line 4). Since we have two tree based solutions, namely, tree-boundk and tree-pivot, we have two corresponding algorithms on graphs, denoted as graph-boundk and graph-pivot, by instantiating tree-knk (line 4) to tree-boundk and tree-pivot respectively.

Global Storage: As discussed above, we have r shortest path trees $T_1, T_2, \cdots, T_r$. For a keyword λ and a node v, let $\mathsf{cand}^i_{v,\lambda}$ be the candidate list of v on tree T_i, $1 \leq i \leq r$. To answer a k-NK query $Q = (q, \lambda, k)$ on a graph, consider a case when the candidate lists of node v on two different trees T_i and T_j are both merged into the result, in the form of

$$R \leftarrow R \otimes_k (\mathsf{cand}^i_{v,\lambda} \oplus \mathsf{dist}_{T_i}(q, v)) \otimes_k (\mathsf{cand}^j_{v,\lambda} \oplus \mathsf{dist}_{T_j}(q, v)).$$

This expression can be generalized to the case of merging the candidate lists of node v on more than two trees. Instead of keeping a candidate list $\mathsf{cand}^i_{v,\lambda}$ for each tree T_i ($1 \leq i \leq r$) separately, we propose a technique called *global storage* which keeps a global candidate list of node v and keyword λ for all trees $T_1, T_2, \cdots, T_r$. Denote the global candidate list of node v and keyword λ as $\mathsf{cand}_{v,\lambda}$. It is computed by

$$\mathsf{cand}_{v,\lambda} = \mathsf{cand}^1_{v,\lambda} \otimes \mathsf{cand}^2_{v,\lambda} \otimes \cdots \otimes \mathsf{cand}^r_{v,\lambda}.$$

For a node v, a node $v' \in \mathsf{cand}_{v,\lambda}$ may appear in the candidate list $\mathsf{cand}^i_{v,\lambda}$ of multiple trees T_i, but will be stored at most once in the global candidate list $\mathsf{cand}_{v,\lambda}$. Therefore, the global storage technique can effectively reduce the index size, but it adds difficulty to query processing due to two reasons: (1) we need to add $\mathsf{dist}_{T_i}(q, v)$ to $\mathsf{cand}^i_{v,\lambda}$ using the $\oplus$ operator, i.e., $\mathsf{cand}^i_{v,\lambda} \oplus \mathsf{dist}_{T_i}(q, v)$, but $\mathsf{dist}_{T_i}(q, v)$ is query dependent, thus cannot be precomputed; (2) the global candidate list may provide a different result list from the one computed by Algorithm 5.9 without using global storage. In the following, we will show that the global candidate list can be used to answer k-NK queries without sacrificing the result quality. We first define the domination relationship between two candidate lists.

Definition 5.7. For two candidate lists $R_1 = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$ and $R_2 = \{v_1 : d_{v_1}, v_2 : d_{v_2}, \cdots\}$ sorted in nondecreasing order of distances, R_1 is dominated by R_2, denoted as $R_1 \geq R_2$, if and only if $|R_1| \leq |R_2|$ and $d_{u_i} \geq d_{v_i}$ for all $1 \leq i \leq |R_1|$. Clearly, the domination relationship is transitive, i.e., if $R_1 \geq R_2$ and $R_2 \geq R_3$, then $R_1 \geq R_3$.

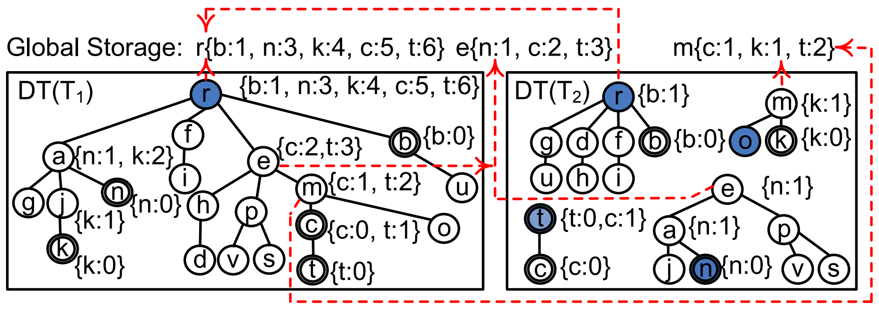


Figure 5.11: Global Storage Example for graph-pivot

To solve the first problem, we need to find a merge method that is independent of $\text{dist}_{T_i}(q,v)$ and at the same time, can generate an answer that is no worse than the answer computed without global storage. The solution is expressed in Equ. 5.1. For any two candidate lists $\text{cand}^i_{v,\lambda} \oplus \text{dist}_{T_i}(q,v)$ and $\text{cand}^j_{v,\lambda} \oplus \text{dist}_{T_j}(q,v)$, using Equ. 5.1, we can generate a better result by merging $\text{cand}^i_{v,\lambda}$ and $\text{cand}^j_{v,\lambda}$ using $\otimes_k$ first, then taking distances $\text{dist}_{T_i}(q,v)$ and $\text{dist}_{T_j}(q,v)$ out and applying the minimum value of them. Clearly, $(\text{cand}^i_{v,\lambda} \otimes_k \text{cand}^j_{v,\lambda}) \oplus \min\{\text{dist}_{T_i}(q,v), \text{dist}_{T_j}(q,v)\}$ is a valid candidate list for query Q, because $\text{cand}^i_{v,\lambda} \otimes_k \text{cand}^j_{v,\lambda}$ is a candidate list for node v and $\min\{\text{dist}_{T_i}(q,v), \text{dist}_{T_j}(q,v)\}$ suggests a path from q to v in G.

$$\begin{aligned}&(\text{cand}^i_{v,\lambda} \oplus \text{dist}_{T_i}(q,v)) \otimes_k (\text{cand}^j_{v,\lambda} \oplus \text{dist}_{T_j}(q,v)) \geq \\ &(\text{cand}^i_{v,\lambda} \otimes_k \text{cand}^j_{v,\lambda}) \oplus \min\{\text{dist}_{T_i}(q,v), \text{dist}_{T_j}(q,v)\}\end{aligned} \tag{5.1}$$

The second problem can be solved if we prove that by merging more candidate lists using the $\otimes$ operator, the answer will not get worse. Consider a node $v' \in \text{cand}_{v,\lambda}$, the merging operation finds the minimum distance between v' and v over multiple trees, which is a refined estimation of their distance on graph. We formulate such a situation using Equ. 5.2.

$$\text{cand}^i_{v,\lambda} \geq \text{cand}^i_{v,\lambda} \otimes \text{cand}^j_{v,\lambda} \tag{5.2}$$

Equ. 5.1 and Equ. 5.2 also hold for multiple candidate lists. Therefore, we show that using global storage will not sacrifice the result quality. More importantly, global storage can effectively reduce the index size and query processing time. It applies to both graph algorithms graph-boundk and graph-pivot. We use the following example

Table 5.1: Algorithm Complexities on Trees (T) and Graphs (G)

	boundk	pivot
Query Time (T)	$O(\log\lvert V_\lambda\rvert + k)$	$O(k\cdot\log\lvert V\rvert)$
Index Time (T)	$O(\overline{k}\cdot\lvert\mathsf{doc}(V)\rvert)$	$O(\lvert\mathsf{doc}(V)\rvert\cdot\log^2\lvert V\rvert)$
Index Size (T)	$O(\overline{k}\cdot\lvert\mathsf{doc}(V)\rvert)$	$O(\lvert\mathsf{doc}(V)\rvert\cdot\log\lvert V\rvert)$
Query Time (G)	$O((\log\lvert V_\lambda\rvert + k)\cdot\log\lvert V\rvert)$	$O(k\cdot\log^2\lvert V\rvert)$
Index Time (G)	$O(\overline{k}\cdot\lvert\mathsf{doc}(V)\rvert\cdot\log\lvert V\rvert)$	$O(\lvert\mathsf{doc}(V)\rvert\cdot\log^3\lvert V\rvert)$
Index Size (G)	$O(\overline{k}\cdot\lvert\mathsf{doc}(V)\rvert\cdot\log\lvert V\rvert)$	$O(\lvert\mathsf{doc}(V)\rvert\cdot\log^2\lvert V\rvert)$

to illustrate global storage.

Example 5.15. We take the graph-pivot algorithm as an example. Fig. 5.11 shows two trees $\mathsf{DT}(T_1)$ and $\mathsf{DT}(T_2)$ for the shortest path tree T_1 and T_2 shown in Fig. 5.5, with candidate list marked beside each node for keyword λ. Using global storage, for the same node on different trees, we merge all its candidate lists using $\otimes$ and only keep one global candidate list. The global candidate lists for nodes r, e and m are marked on the top of Fig. 5.11. For query $Q_1 = (p, \lambda, 2)$, without global storage, we need to merge three candidate lists, $\mathsf{cand}^1_{e,\lambda} \oplus \mathsf{dist}_{T_1}(p, e)$, $\mathsf{cand}^1_{r,\lambda} \oplus \mathsf{dist}_{T_1}(p, r)$ and $\mathsf{cand}^2_{r,\lambda} \oplus \mathsf{dist}_{T_2}(p, e)$. Using global storage, only two candidate lists $\mathsf{cand}_{e,\lambda} \oplus \min\{\mathsf{dist}_{T_1}(p, e), \mathsf{dist}_{T_2}(p, e)\}$, $\mathsf{cand}_{r,\lambda} \oplus \mathsf{dist}_{T_1}(p, r)$ need to be merged. For query $Q_2 = (h, \lambda, 2)$, without global storage, we get the result $R = \{c : 3, t : 4\}$. Using global storage, we can get a result $R' = \{n : 2, c : 3\}$ with $R \geq R'$.

Handling Multiple Keywords: We discuss how to extend our approach to handle a k-NK query of multiple keywords with AND (denoted as $\wedge$) and OR (denoted as $\vee$) semantics. Without loss of generality, we assume the format of a keyword expression is $(\lambda_{1,1} \wedge \lambda_{1,2} \cdots) \vee (\lambda_{2,1} \wedge \lambda_{2,2} \cdots) \vee \cdots$. It is easy to handle $\vee$, by answering each $\lambda_{i,1} \wedge \lambda_{i,2} \cdots$ separately and merging the results using the $\otimes$ operator. For handling $\lambda_{i,1} \wedge \lambda_{i,2} \cdots$, we select a keyword $\lambda_{i,j}$ from $\{\lambda_{i,1}, \lambda_{i,2}, \cdots\}$ with the least frequency $\lvert V_{\lambda_{i,j}}\rvert$ as the primary keyword and consider other keywords as filter keywords. We answer the query for the single keyword $\lambda_{i,j}$. Before merging each candidate list

Table 5.2: Dataset Statistics

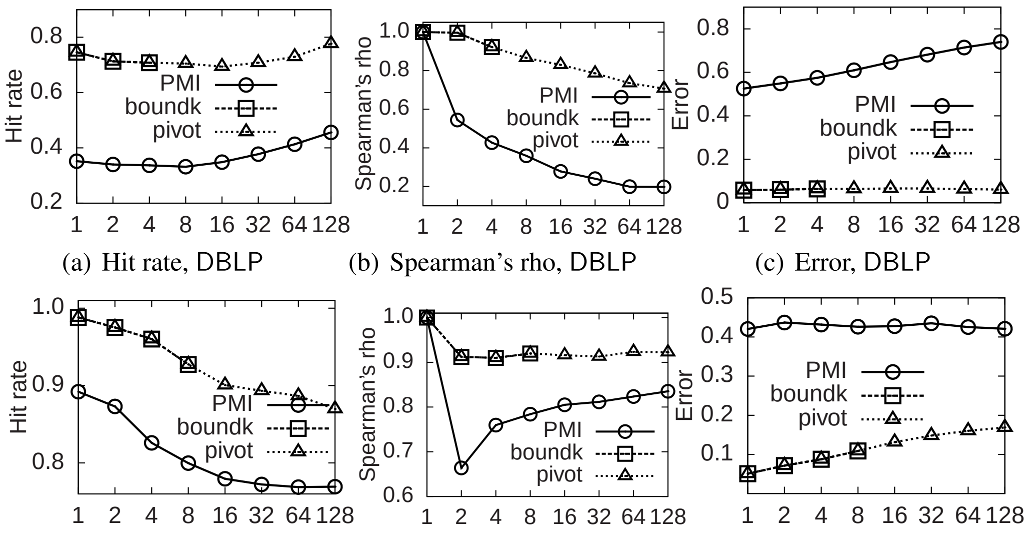
	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathsf{doc}(V)\rvert$	keywords
DBLP	$1,695,469$	$4,726,801$	$12,842,501$	$331,301$
FLARN	$1,070,376$	$1,356,399$	$6,966,665$	$2,730$

using the $\otimes$ operator, we remove the candidate nodes that do not contain one or more of the filter keywords from the candidate list. In this way, each element in the final answer satisfies the predicate specified in the keyword expression.

Comparison: Table 5.1 summarizes and compares the query time, index time and index size for boundk and pivot on trees and graphs. Here, the listed complexities of index time and index size are for all keywords in the tree/graph. boundk is faster than pivot in query processing on both trees and graphs. When k is small, the index time and index space for boundk are smaller than pivot on both trees and graphs. However, when k is large, the index time and index space for boundk are large, while the index time and index space of pivot are independent of k on both trees and graphs.

5.7. Experiments

In this subsection, we report the performance of our methods boundk, pivot, and their global storage implementations boundk-gs and pivot-gs, with two baseline solutions BFS and PMI. BFS is a brute-force search that uses Dijkstra's algorithm to identify the nearest k keyword nodes, and PMI (Partitioned Multi-Indexing) [5] is the state-of-the-art approximate algorithm based on distance oracle [78]. For all the distance oracles involved we set the parameter $r = \log_2 |V|$. We implemented all methods in GNU C++, and conducted all experiments on a Windows machine with an Intel Xeon 2.7GHz CPU and 128GB memory. All methods run in main memory. A 32GB memory limit is set for index size.



(a) Hit rate, DBLP (b) Spearman's rho, DBLP (c) Error, DBLP

(d) Hit rate, FLARN (e) Spearman's rho,FLARN (f) Error, FLARN

Figure 5.12: Hit rate, Spearman's rho and Error by Varying k

Datasets and Queries. We use two real graphs, DBLP[2], and Florida road network FLARN[3], with statistics listed in Table 5.2.

DBLP includes $1,060,763$ articles, $631,589$ authors and $3,117$ conferences/journals, all of which are treated as nodes. There is an edge between nodes u and v, if u is an author of article v, or u is an article published in conference/journal v. The keywords of an author node include first name and last name, the keywords of an article node include title words, editor, year, publisher, isbn, etc., and the keywords of a conference/journal node include association and name. A weight $(\log_2 \mathsf{deg}(u) + \log_2 \mathsf{deg}(v))$ is assigned to edge (u, v), where $\mathsf{deg}(u)$ denotes the degree of node u. Compared with the unit edge weight setting, the numerical edge weights can effectively differentiate the weights of all edges in a graph. Thus for any k-NK query, this helps produce a ranking of top-k answer nodes with less ties in their distances as the ground truth, which is important for fair and unambiguous ranking quality evaluation.

In FLARN, a node represents an intersection or endpoint, an edge denotes a road segment, and the edge weight is the distance of the road segment. We obtained the

[2]http://www.informatik.uni-trier.de/~ley/db

[3]http://www.dis.uniroma1.it/challenge9/download.shtml

keywords of nodes from the OpenStreetMap project[4] with a bounding box. However, only $7,172$ nodes out of $1,070,376$ have keywords. To address the keyword sparseness issue and better discriminate different methods, we assign a random number (between 0 and 4) of keywords to the nodes with no keyword. After this step, there are still $213,081$ nodes without any keyword in FLARN.

We remove stop words in DBLP and FLARN. For each dataset, we generate 500 k-NK queries in the form of $Q = (q, \lambda, k)$, where $q \in V$ is a randomly selected query node, and λ is a keyword randomly selected by following the keyword frequency distribution in the document collection. We test $k = 1, 2, \ldots, 128$.

Evaluation Metrics. We use six metrics for evaluation: *hit rate*, *Spearman's rho* [82], *error*, *query time*, *index time*, and *index size*. *Spearman's rho* measures the rank correlation between an approximate rank result and the ground truth. *Hit rate* and *error*, defined as follows, measure the quality of an approximate result. For a query $Q = (q, \lambda, k)$, denote the exact result as $R = \{u_1 : d_1, \ldots, u_k : d_k\}$ in nondecreasing order of their distances, and $\overline{d} = d_k$ as the upper bound distance of the result R. Denote an approximate result set as $R' = \{u'_1 : d'_1, \ldots, u'_k : d'_k\}$ in nondecreasing order of their distances. The hit rate is defined as:

$$\mathsf{hit}(R') = |\{i \in [1, k]| \mathsf{dist}(u'_i, q) \le \overline{d}\}|/k$$

and the error is the average relative error of the estimated distances w.r.t. the ground truth:

$$\mathsf{err(R')} = \sum_{1 \le i \le k} |d'_i/d_i - 1|/k$$

Hit rate, Spearman's rho and Error. Figures 5.12(a)–(c) show the hit rate, Spearman's rho, and error on DBLP respectively when we vary k. Our method pivot improves the hit rate of PMI by 96%, and improves Spearman's rho by 111% on average. The error of pivot is within 0.066 for all k values, demonstrating that the distance estimated by pivot is very close to the exact distance. Notably, pivot reduces the error of PMI by an order of magnitude, i.e., from 0.630 to 0.063 on average. Furthermore, the

[4]http://wiki.openstreetmap.org/wiki/Main_Page

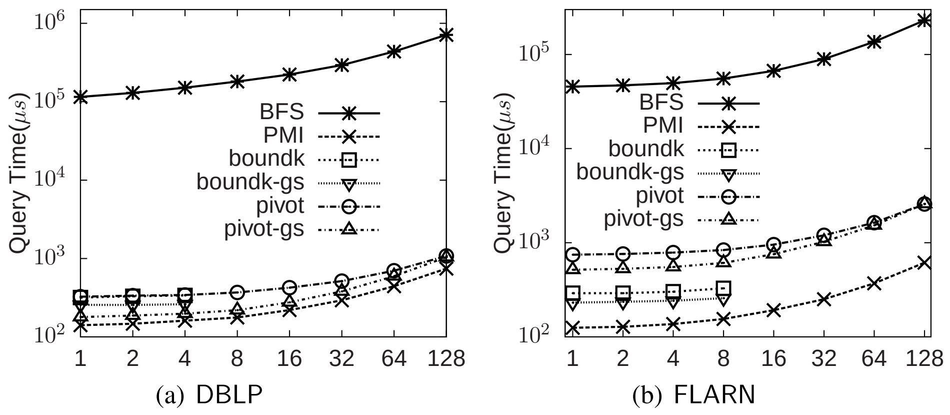


Figure 5.13: Query Time in Microseconds by Varying k

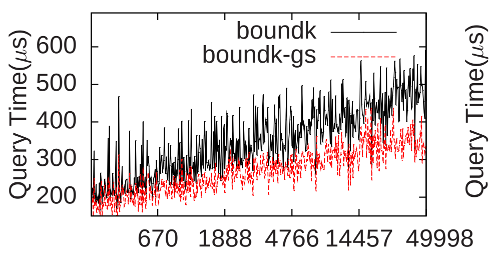
error of pivot does not increase with k, while that of PMI increases by 40% with the increase of k. Note when $k = 1$, Spearman's rho is constantly 1.

Figures 5.12(d)–(f) show the hit rate, Spearman's rho, and error on FLARN respectively. pivot improves both the average hit rate and Spearman's rho of PMI by 14%. The error of pivot is below 0.168 for all k values and is 4 times smaller than that of PMI on average.

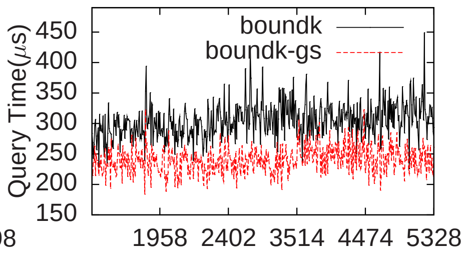
Note that the performance of BFS is omitted in Figure 5.12, as it returns the exact result. Furthermore, the result quality between using and not using global storage does not differ substantially, for the sake of clarity, the global storage methods boundk-gs and pivot-gs are also omitted in Figure 5.12. But we do observe that global storage technique improves the hit rate of boundk/pivot by 1.3% on DBLP and 0.7% on FLARN, and reduces the error by 6.7% on DBLP and 16.9% on FLARN on average. Given the memory limit of 32GB for index size, boundk can only support $k \leq 4$ on DBLP and $k \leq 8$ on FLARN in Figure 5.12 as its index size increases linearly with $\overline{k}$.

Query Time. Figure 5.13 shows the query time of different methods in log scale when we vary k. The query time of BFS is 10^5–10^6 microseconds, which is two to three orders of magnitude slower than the other methods.

Figure 5.13(a) shows the query time on DBLP. The query time of all methods increases with the increase of k. PMI is the most efficient. The query time of boundk, boundk-gs, pivot and pivot-gs is less than 2 times that of PMI, which is quite close.



(a) DBLP



(b) FLARN

Figure 5.14: Query Time of boundk Varying Keyword Frequency

Global storage reduces the query time of boundk by 22% and that of pivot by 25%. Remarkably, each of our proposed approaches can report a result within 1 millisecond for all k values.

Figure 5.13(b) shows the query time on FLARN. We can observe that PMI is the fastest, closely followed by boundk and boundk-gs, whose query time is less than two times that of PMI and one third that of pivot for all k values. pivot and pivot-gs take a little longer as their query time depends on the tree depth which is large on FLARN. But their query time is within 3 milliseconds for $k = 128$, which is still quite efficient. Global storage helps reduce the query time of boundk by 20% and that of pivot by 15%.

Figure 5.14 further plots the query time of boundk and boundk-gs on the 500 k-NK queries in ascending order of the query keyword frequency in the graph. We set $k = 4$ in this experiment. For illustration, we also label a few query keyword frequencies on the x axis. The query time shows a sharper increasing trend on DBLP than FLARN, as the frequency difference between DBLP keywords is larger. These empirical results are consistent with the theoretical result, i.e., the query time complexity of boundk depends on $\log |V_\lambda|$, where $|V_\lambda|$ is the frequency of keyword λ.

Index Time and Index Size. Figure 5.15 shows the total index time (IT) and index size (IS) for indexing all keywords by different methods. We observe that the index time of pivot is 2.6 times that of PMI on DBLP, and 8.2 times on FLARN. The index construction time of pivot is longer on FLARN than on DBLP. This is because the

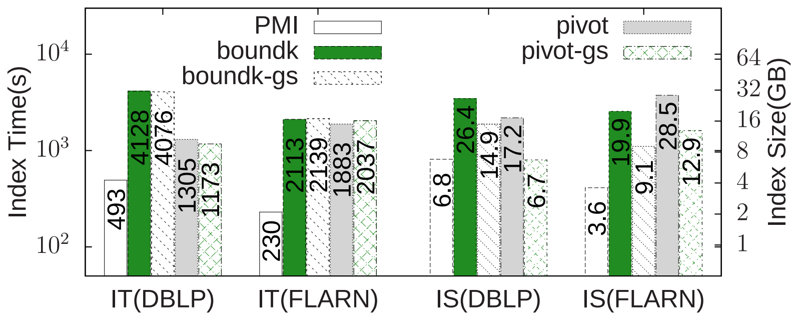


Figure 5.15: Index Time and Index Size

complexity of pivot grows linearly with the tree depth, and the larger diameter of FLARN leads to a larger tree depth. All methods can finish the index construction for all keywords in a graph within 1.15 hours.

Given the memory limit of 32GB for index size, boundk can only support $k \leq 4$ on DBLP and $k \leq 8$ on FLARN, as its index size increases linearly with $\overline{k}$. In contrast, pivot/pivot-gs have no such limitation. The index size of pivot is 2.5 times that of PMI on DBLP and 7.9 times on FLARN, due to the larger diameter of FLARN. By keeping a global candidate list and removing duplicate index items, global storage reduces the index size of pivot by 61% on DBLP and 55% on FLARN. It also reduces the index size of boundk by 44% on DBLP and 54% on FLARN. Remarkably, the index size of pivot-gs is 6.7GB on DBLP, which is even *smaller* than that of PMI (6.8GB). This result proves the superiority of global storage.

5.8. Summary

In this chapter, we study top-k nearest keyword (k-NK) search on large graphs. We propose two exact k-NK algorithms on trees to handle a bounded k and an arbitrary k respectively. We extend tree based algorithms to graphs and propose a global storage technique to further reduce the index size and query time. Extensive performance studies on real large graphs demonstrate the effectiveness and efficiency of our algorithms.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Due to the massive size of graphs from various domains nowadays, even simple graph queries become challenging tasks. Three queries with a wide range of applications are investigated in this thesis, *shortest distance* query, *weight constraint reachability* query, and *top-k nearest keywords* query.

For a shortest distance query, we devise two landmark embedding schemes, an *error bounded landmark scheme* and a *local landmark scheme*, where the former can guarantee an error bound for estimated distance, and the latter can significantly improve the distance estimation accuracy without increasing the offline embedding or the online query complexity. For a WCR query, we propose a memory-based approach which promises a constant query time. Besides, in order to increase its scalability, we devise an I/O-efficient approach for answering a WCR query on massive graphs. For a k-NK query, we start with a special case when the graph is a tree, based on which we present our algorithm for approximate k-NK query on a graph. A global storage technique is devised to further reduce the index size and the query time. We did extensive experiments on the three queries respectively to show the effectiveness and efficiency of our methods.

The investigation on graph query processing is still far from an end. New applications are posing new challenges. For example, in a location based social network such as Gowalla and Brightkite, users report locations from their mobile devices to share activities with their friends, therefore, a personalized query processing based on both social networks and road networks is in demand which is one of our future works. In addition, we intend to increase the scalability of the existing query processing by techniques including summarizing, sampling and sketching, or by developing I/O efficient solutions instead of memory-based approaches in the near future.

BIBLIOGRAPHY

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data (SIGMOD 1989)*, pages 253–262, 1989.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. *CoRR*, abs/1304.4661, 2013.

[4] Takuya Akiba, Christian Sommer, and Ken ichi Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*, pages 144–155, 2012.

[5] Bahman Bahmani and Ashish Goel. Partitioned multi-indexing: Bringing order to social search. In *WWW*, pages 399–408, 2012.

[6] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, 2007.

[7] Gurkan Bebek and Jiong Yang. PathFinder: Mining signal transduction pathway segments from protein-protein interaction networks. *BMC Bioinformatics Journal*, 8:335, 2007.

[8] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer Berlin / Heidelberg, 2000.

[9] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

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

[11] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. On incremental maintenance of 2-hop labeling of graphs. In *Proceedings of the 17th international conference on World Wide Web (WWW 2008)*, pages 845–854, 2008.

[12] Xin Cao, Lisi Chen, Gao Cong, and Xiaokui Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.

[13] Shiva Chaudhuri and Christos D. Zaroliagis. Shortest paths in digraphs of small treewidth. part i: Sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

[14] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*, pages 893–902, 2008.

[15] Yen-Yu Chen, Torsten Suel, and Alexander Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[16] Jiefeng Cheng and Jeffrey Xu Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[17] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008)*, pages 193–204, 2008.

[18] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: Efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[19] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 937–946, 2002.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[21] Ke Deng, Xiaofang Zhou, Heng Tao Shen, Shazia W. Sadiq, and Xue Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.

[22] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

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

[24] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *Proceedings of the 27th International Conference on Data Engineering (ICDE 2011)*, pages 39–50, 2011.

[25] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the 1998 Symposium on Principles of Database Systems (PODS 1998)*, pages 139–148, 1998.

[26] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global internet host distance estimation service. *IEEE/ACM Trans. Networking*, 9(5):525–540, 2001.

[27] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[28] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[29] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.

[30] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering and Experiments*, pages 129–143, 2006.

[31] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940, 2008.

[32] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *Proc. 2010 Int. Conf. Information and Knowledge Management (CIKM'10)*, 2010.

[33] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2):100–107, 1968.

[34] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Compact reachability labeling for graph-structured data. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2005)*, pages 594–601, 2005.

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

[36] Danny Hermelin, Avivit Levy, Oren Weimann, and Raphael Yuster. Distance oracles for vertex-labeled graphs. In *ICALP (2)*, pages 490–501, 2011.

[37] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[38] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 894–905, 2006.

[39] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[40] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[41] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *Proceedings of the VLDB Endowment (PVLDB 2011)*, 4(9):551–562, 2011.

[42] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD Conference*, pages 123–134, 2010.

[43] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD Conference*, pages 445–456, 2012.

[44] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD international conference on Management of data (SIGMOD 2009)*, pages 813–826, 2009.

[45] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD 2008)*, pages 595–608, 2008.

[46] R. Johnsonbaugh and M. Kalin. A graph generation software package. In *Proceedings of the 22nd SIGCSE technical symposium on Computer science education (SIGCSE 1991)*, pages 151–154, 1991.

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

[48] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[49] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[50] Samir Khuller, Anna Moss, and Joseph Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.

[51] J. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *Proc. 2004 Symp. Foundations of Computer Science (FOCS'04)*, pages 444–453, 2004.

[52] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. 2004 Int. Conf. Very Large Data Bases (VLDB'04)*, pages 840–851, 2004.

[53] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Tim Schmidt. Hierarchical graph embedding for efficient query processing in very large traffic networks. In *SSDBM*, pages 150–167, 2008.

[54] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[55] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Record*, 30(1):19–24, 2001.

[56] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowledge Discovery from Data*, 1(1), 2007.

[57] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[58] Q. Ma and P. Steenkiste. On path selection for traffic with bandwidth guarantees. In *Proceedings of the 1997 International Conference on Network Protocols (ICNP 1997)*, pages 191–202, 1997.

[59] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[60] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proc. 7th ACM SIGCOMM Conf. Internet measurement*, pages 29–42, 2007.

[61] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[62] E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Int. Conf. on Computer Communications (INFOCOM'01)*, pages 170–179, 2001.

[63] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network database. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 802–813, 2003.

[64] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proc. 2009 Int. Conf. Information and Knowledge Management (CIKM'09)*, pages 867–876, 2009.

[65] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *ICDE*, pages 462–473, 2012.

[66] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints):1, 2012.

[67] Miao Qiao, Hong Cheng, Lu Qin, Jeffrey Xu Yu, Philip S. Yu, and Lijun Chang. Computing weight constraint reachability in large networks. *VLDB J.*, 22(3):275–294, 2013.

[68] Miao Qiao, Hong Cheng, and Jeffrey Xu Yu. Querying shortest path distance with bounded errors in large graphs. In *SSDBM*, pages 255–273, 2011.

[69] Miao Qiao, Lu Qin, Hong Cheng, Jeffrey Xu Yu, and Wentao Tian. Top-k nearest keyword search on large graphs. In *Proc. 2013 Int. Conf. Very Large Data Bases (VLDB'13)*, 2013.

[70] Lu Qin, Jeffrey Xu Yu, Lijun Chang, and Yufei Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

[71] M. J. Rattigan, M. Maier, and D. Jensen. Using structure indices for efficient approximation of network properties. In *Proc. 2006 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'06)*, pages 357–366, 2006.

[72] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th annual ACM symposium on Theory of computing (STOC 2004)*, pages 184–191, 2004.

[73] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proc. 2008 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'08)*, pages 43–54, 2008.

[74] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[75] Jagan Sankaranarayanan and Hanan Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, 2009.

[76] Jagan Sankaranarayanan and Hanan Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1158–1175, 2010.

[77] Jagan Sankaranarayanan, Hanan Samet, and Houman Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[78] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proc. 2010 Int. Conf. Web Search and Data Mining*, pages 401–410, 2010.

[79] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An efficient connection index for complex XML document collections. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, pages 237–255, 2004.

[80] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proceedings of the 21th International Conference on Data Engineering (ICDE 2005)*, pages 360–371, 2005.

[81] C. Shahabi, M. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proc. 10th ACM Int. Symp. Advances in Geographic Information Systems (GIS'02)*, pages 94–100, 2002.

[82] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol.*, 15(1):72–101, 1904.

[83] Yufei Tao, Stavros Papadopoulos, Cheng Sheng, and Kostas Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, pages 589–600, 2011.

[84] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[85] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *STOC*, pages 183–192, 2001.

[86] Silke TrißI and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD 2007)*, pages 845–856, 2007.

[87] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD international conference on Management of data (SIGMOD 2011)*, pages 913–924, 2011.

[88] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN 2009)*, pages 37–42, 2009.

[89] Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[90] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of*

the 22th International Conference on Data Engineering (ICDE 2006), page 75, 2006.

[91] F. Wei. TEDI: Efficient shortest path query answering on graphs. In *Proc. 2010 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'10)*, pages 99–110, 2010.

[92] K. Xu, L. Zou, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Answering label-constraint reachability in large graphs. In *Proceedings of the 2011 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2011)*, pages 1595–1600, 2011.

[93] Bin Yao, Mingwang Tang, and Feifei Li. Multi-approximate-keyword routing in gis data. In *GIS*, pages 201–210, 2011.

[94] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment (PVLDB 2010)*, 3(1):276–284, 2010.

[95] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.