2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

Fast Indexing Algorithm for Efficient kNN Queries on Complex Networks

Suomi Kobayashi
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
kobayashi@kde.cs.tsukuba.ac.jp

Shohei Matsugu
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
matsugu@kde.cs.tsukuba.ac.jp

Hiroaki Shiokawa
Center for Computational Sciences
University of Tsukuba
Tsukuba, Japan
shiokawa@cs.tsukuba.ac.jp

***Abstract*—k nearest neighbor (kNN) query is an essential graph data management tool to find relevant data entities suited to a user-specified query node. Graph indexing methods have the potential to achieve a quick kNN search response, the graph indexing methods are one of the promising approaches. However, they struggle to handle large-scale complex networks since constructing indexes and to querying kNN nodes in the large-scale networks are computationally expensive. In this paper, we propose a novel graph indexing algorithm for a fast kNN query on large networks. To overcome the aforementioned limitations, our algorithm generates two types of indexes based on the topological properties of complex networks. Our extensive experiments on real-world graphs clarify that our algorithm achieves up to 18,074 times faster indexing and 146 times faster kNN query than the state-of-the-art methods.**

***Index Terms*—Graph query, kNN search, Indexing**

I. Introduction

Given a graph G, can k nearest neighbor (kNN) nodes to a user-specified query node be found efficiently? This work presents a fast graph indexing algorithm to efficiently query kNN nodes on large complex networks.

As social applications advance, complex networks (or graphs) are becoming increasingly important to represent complicated data [1], [2]. To handle such networks, kNN queries [3] are essential building blocks in various applications. Given a query node in a graph, kNN query explores a set of nodes with the top-k shortest path distances from the query. Unlike traditional distance-based queries, kNN queries can return a result within a short time since they do not compute the whole of the graph. Due to this feature, kNN queries have been employed on various social applications such as a POI search and user recommendations.

Although kNN queries are useful in many applications, they have a serious drawback when handling real-world complex networks. Specifically, traditional kNN search algorithms are computationally expensive if the given graph is large. Historically, kNN queries are applied to small graphs such as ego-networks and road networks, which have a few thousand nodes at most. By contrast, recent social networking applications must handle large-scale complex networks with a few million nodes [4]. That is, applications suffer from a long computation time to query kNN nodes when the traditional algorithms run queries. Thus, the algorithms fail to find kNN nodes in large-scale real-world complex networks.

A. Existing Approaches and Challenges

Various approaches have been proposed to overcome the expensive costs in kNN queries. *Graph indexing methods* are the most successful to date [3], [5], [6]. Examples include G-Tree [3] and ILBR [6]. To achieve a fast kNN query on a graph, they construct an index by partitioning a graph and pre-computing the shortest-path distances among nodes in the graph before running a kNN query. For instance, G-Tree [3] partitions a graph into disjoint subgraphs using Metis [7], and it pre-computes the distances among all nodes included in each subgraph. Similarly, ILBR selects several landmark nodes from a graph, and constructs a Voronoi subgraph using the landmarks. Then, it then pre-computes the distance from the landmarks to nodes in each Voronoi subgraph. By searching kNN nodes on the index, these methods avoid computing unnecessary nodes and edges, achieving a fast kNN query.

Although indexing methods improve kNN query efficiency, they cannot handle large complex networks due to two reasons. First, indexing complex networks is expensive because they are designed under the assumption that most networks are the planar graph [8]. Their partitioning methods are efficient for planar graphs such as road networks but their assumptions are not suitable for complex networks with diverse structures [9]. Second, a long running time is necessary for a kNN query on complex networks due to the above assumption. Regardless of indexing, they must compute many distances for complex networks at the querying time because the indexed subgraphs do not include many edges in the complex networks. Thus, a fast graph indexing algorithm for efficient kNN queries remains elusive.

ASONAM '21, November 8–11, 2021, Virtual Event, Netherlands

ACM ISBN 978-1-4503-9128-3/21/11?/$15.00
https://doi.org/10.1145/3487351.3489442

B. Our Approaches and Contributions

Our goal is to achieve a fast indexing to efficiently find exact kNN nodes on large complex networks. We present a fast graph indexing method based on the well-known topological property of complex networks, called the *core-tree* property [10]. Real-world complex networks are typically composed of two parts: *a core* and *trees*. A core is a small dense subgraph whose nodes are highly connected within the subgraph. By contrast, a tree is a long stretched sparse subgraph. As reported in [11], the vast majority of nodes are included in trees, and only 0.6 – 9.3% nodes are contained in the core for a typical complex network.

Based on the above property, our algorithm reduces the indexing costs by separately indexing a core and trees. First, it extracts trees from a graph, and generates a *tree-index* by computing the distances between the root node to each leaf node in a tree. It then constructs a *core-index* from non-tree nodes by updating edge-weights of the graph. In the kNN query process, our algorithm explores nodes on the core-index by pruning unnecessary computations for trees based on the tree-index. This approach leads to the following properties:

- **Fast Indexing:** Our algorithm achieves faster indexing than the state-of-the-art methods proposed in the last few years (Section IV-B). We experimentally confirmed that our method outperforms state-of-the-art methods by up to four orders of magnitude.
- **Fast Querying:** Our proposed method is also faster than the state-of-the-art methods in terms of the kNN query time (Section IV-C). We experimentally demonstrate that it is up to 146 times faster than state-of-the-art methods.
- **Correctness:** Our method theoretically guarantees to find exact kNN nodes, although it prunes unpromising nodes using the tree-index (Theorem 1).

Our algorithm is the first solution that achieves fast indexing and kNN queries on large complex networks. For instance, it generates an index for a Pokec network with 1.6 million nodes within 5 seconds, and it finds exact kNN nodes within 1 second. Although kNN queries are useful in real-life applications, applying them to large graphs is difficult. Our fast and exact method will enhance many applications.

II. Problem Statement

Here, we briefly define the problem addressed in this paper. Let $G = (V, E, W)$ be a weighted undirected graph, where V, E, and W are sets of nodes, edges, and edge-weights, respectively. If a graph G has an edge between nodes $u, v \in V$, it is denoted as $e(u, v) \in E$. For each edge $e(u, v) \in E$, an edge-weight $w(u, v) \in W$ must be defined such that $w(u, v) \in \mathbb{N}$.

The kNN query problem is defined as follows:

Definition 1 (kNN query)**.** *Given a graph $G = (V, E, W)$, a query node $q \in V$, and an integer $k \in \mathbb{N}$, kNN query is a problem to find a set of nodes V_k such that $|V_k| = k$ and $dist(q, u) \leq dist(q, v)$, where $dist(u, v)$ is the shortest path distance between u and v, $u = arg\ max_{u' \in V_k} dist(q, u')$, and $v = arg\ min_{v' \in V \setminus V_k} dist(q, v')$.*

Algorithm 1 INDEXING

Input: a graph $G = (V, E, W)$;
Output: a graph index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;

▷ **(Step 1) Tree-indexing:**
1: $\mathbb{T} \leftarrow$ EXTRACTTREES(G) and $\mathbb{D} \leftarrow \emptyset$;
2: **for each** $T_i \in \mathbb{T}$ **do**
3: $r \leftarrow root(T_i)$ and $D_i \leftarrow \emptyset$;
4: **for each** $v \in T_i$ **do**
5: $D_i \leftarrow D_i \cup \{dist(r, v)\}$;
6: $\mathbb{D} \leftarrow \mathbb{D} \cup D_i$;
7: $\mathcal{T} = (\mathbb{T}, \mathbb{D})$;

▷ **(Step 2) Core-indexing:**
8: **for each** $T_i \in \mathbb{T}$ **do**
9: $r \leftarrow root(T_i)$ and $V_c \leftarrow (V \setminus T_i) \cup \{r\}$;
10: $E_c = \{e(u, v) \in E | u, v \in V_c\}$ and $W_c \leftarrow \emptyset$;
11: **for each** $e(u, v) \in E_c$ **do**
12: $W_c \leftarrow W_c \cup \{dist(u, v)\}$;
13: $\mathcal{C} = (V_c, E_c, W_c)$;
14: **return** $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;

III. Proposed Algorithm

We present our indexing method for a fast kNN query on large complex networks. Our indexing and kNN query algorithms are introduced in Section III-A and III-B, respectively.

A. Indexing Construction

Based on the core-tree property, our algorithm generates an index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$, where $\mathcal{T}$ and $\mathcal{C}$ are defined as follows:

Definition 2 (Tree-index $\mathcal{T}$)**.** *Let $T_1, T_2, \ldots, T_i$ be trees included in G, r_i be a root node of T_i, and $D_i = \bigcup_{v \in T_i} \{dist(r, u)\}$. The tree-index is defined as $\mathcal{T} = (\mathbb{T}, \mathbb{D})$, where $\mathbb{T} = \{T_1, T_2, \ldots, T_i\}$ and $\mathbb{D} = \{D_1, D_2, \ldots, D_i\}$.*

Definition 3 (Core-index $\mathcal{C}$)**.** *Let V_c be a set of nodes included in a core of G. The core-index is defined as $\mathcal{C} = (V_c, E_c, W_c)$, where $E_c = \{e(u, v) \in E | u, v \in V_c\}$, $W_c = \{dist(u, v) | e(u, v) \in E_c\}$.*

For convenience, we also define a label function f_l as follows:

Definition 4 (Label function f_l)**.** *A label function f_l is defined as $f_l[u] =$ tree if $u \in \mathcal{T}$. Otherwise, $f_l[u] =$ core.*

Algorithm: Algorithm 1 is a pseudo-code of our algorithm to generate $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$. This algorithm has two components: the tree-indexing (lines 1-7) and the core-indexing (lines 8-14).

Given a graph G, the goal of the tree-indexing step is to construct the tree-index $\mathcal{T}$ in Definition 2. First, the algorithm extracts all trees $\mathbb{T} = \{T_1, T_2, \ldots, T_i\}$ in G. To obtain $\mathbb{T}$, it runs the EXTRACTTREES function (line 1), which employs the incremental aggregation method [12]. This aggregation method is summarized as follows:

1) Select a node $u \in V$ whose degree is one.
2) Aggregate u into its adjacent nodes.
3) Continue the above steps until no nodes are aggregated.
4) Output the aggregated nodes as a set of trees $\mathbb{T}$.

Now, we have a set of trees $\mathbb{T} = \{T_1, T_2, \ldots, T_i\}$, each of which is a set of nodes included in a tree. Algorithm 1

Algorithm 2 kNN QUERY PROCESSING

Input: an index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$, a query node q, and the number of results k;
Output: a set of kNN nodes V_k;

▷ **(Step 1) Initialization:**
1: $V_k \leftarrow \emptyset$, and priority queue $Q \leftarrow \emptyset$;
2: **if** $f_l[q] = \textit{tree}$ **then**
3: Obtain T_i *s.t.* $q \in T_i$ and $r \leftarrow \textit{root}(T_i)$;
4: Obtain $\textit{dist}(q, r)$ from D_i;
5: $\langle V_k, Q \rangle \leftarrow$ TREEPRUNING$(\langle r, \textit{dist}(q, r) \rangle)$;
6: **else**
7: $Q \leftarrow \{\langle q, 0 \rangle\}$;

▷ **(Step 2) kNN search:**
8: **while** $|V_k| \leq k$ **do**
9: $\langle u, \textit{dist}(q, u) \rangle \leftarrow Q.\textit{dequeue}()$;
10: **if** $f_l[u] = \textit{tree}$ **then**
11: $\langle V_k, Q \rangle \leftarrow$ TREEPRUNING$(\langle u, \textit{dist}(q, u) \rangle)$;
12: **else**
13: $V_k \leftarrow V_k \cup \{u\}$;
14: **for each** $\{e(u, v) \in E_c | v \notin V_k\}$ **do**
15: $\textit{dist}(q, v) \leftarrow \textit{dist}(q, u) + w(u, v)$;
16: Q.enqueue$(\langle v, \textit{dist}(q, v) \rangle)$;
17: **return** V_k;

▷ **Subroutine for tree pruning:**
18: **procedure** TREEPRUNING$(\langle r, \textit{dist}(q, r) \rangle)$
19: Obtain T_i *s.t.* $r \in T_i$;
20: **for each** $\{e(r, v) \in E_c | v \notin V_k\}$ **do**
21: $\textit{dist}(q, v) \leftarrow \textit{dist}(q, r) + w(r, v)$;
22: $Q \leftarrow Q \cup \{\langle v, \textit{dist}(q, v) \rangle\}$;
23: $d_{\min} \leftarrow \min\{\textit{dist}(q, v) | \langle u, \textit{dist}(q, v) \rangle \in Q\}$;
24: **if** $|V_k| + |T_i| \leq k$ and $\overline{d}(r) \leq d_{\min}$ **then**
25: $V_k \leftarrow V_k \cup T_i$;
26: **else**
27: **for each** $v \in T_i$ **do**
28: $f_l[v] \leftarrow \textit{core}$, $\textit{dist}(q, v) \leftarrow \textit{dist}(q, r) + \textit{dist}(v, r)$;
29: $Q \leftarrow Q \cup \{\langle v, \textit{dist}(q, v) \rangle\}$;

computes D_i in Definition 2 for each tree $T_i \in \mathbb{T}$ (lines 2-6). Finally, it outputs the tree-index $\mathcal{T} = (\mathbb{T}, \mathbb{D})$ (line 7).

Next, the core-indexing step generates the core-index $\mathcal{C}$ (lines 8-14). As shown in Definition 3, $\mathcal{C}$ is composed of V_c, E_c, and W_c. In lines 8-9, the algorithm constructs a set of nodes V_c, which includes all non-tree nodes and all root nodes in $\mathbb{T}$. Algorithm 1 then links nodes in V_c if they have edges in E (line 10). Finally, the algorithm updates the weights of edges (line 11-12). As shown in line 12, each weight is set as the shortest path distance between two adjacent core nodes.

B. kNN Query Processing

This section explains how a kNN query is computed on $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$. To reduce the query time, our algorithm attempts to avoid computing tree-nodes. The algorithm starts a kNN search from a core node in $\mathcal{C}$. Once it reaches a root node r of a tree T_i, it examines whether computing T_i can be skipped by the tree-index $\mathcal{T}$. If T_i can be kNN nodes, the algorithm adds them into V_k without computing T_i.

Algorithm: Algorithm 2 shows the pseudo-code of kNN query using $\mathcal{I}$. Algorithm 2 has two components: the main search algorithm (lines 1-17) and a subroutine TREEPRUNING (lines 18-29).

The main search algorithm explores kNN nodes for q using the index $\mathcal{I}$. At the beginning of the algorithm, it initializes a priority queue Q, in which nodes are prioritized by the distance from q. In the first step (lines 1-7), our algorithm initializes Q based on the label function $f_l[q]$. If $f_l[q] = \textit{core}$, it simply inserts q into Q (lines 6-7). Otherwise it invokes the subroutine TREEPRUNING (lines 2-5).

Given a root node r of T_i, TREEPRUNING examines whether T_i is included in V_k without computing nodes in T_i (lines 18-29). First, we introduce the following definition:

Definition 5 (Upper bound $\overline{d}$)**.** *Let r be a root node of T_i, the upper bound of distances $\overline{d}(r)$ is defined as follows:*

$$\overline{d}(r) = \begin{cases} \textit{dist}_{max}(T_i) - \textit{dist}(q, r) & (q \in T_i) \\ \textit{dist}(q, r) + \textit{dist}_{max}(T_i) & (\textit{Otherwise}) \end{cases}$$

where $\textit{dist}_{max}(T_i) = \max\{\textit{dist}(u, v) | \textit{dist}(u, v) \in D_i\}$.

Definition 5 indicates that $\overline{d}(r)$ is the maximum distance between the query node q and a node in T_i. That is, $\overline{d}(r)$ is the upper bound of distances between q and T_i. From Definition 5, we have the following property:

Lemma 1. *Let r be a root of T_i, and $d_{min} = \min \textit{dist}(q, v)$, where $v \in Q \cup \{v | e(r, v) \in E_c, v \notin V_k\}$. If $|V_k| + |T_i| \leq k$ and $\overline{d}(r) \leq d_{min}$, then $T_i \subseteq V_k$ holds.*

Proof: If $q \in T_i$, Lemma 1 trivially holds. Thus, we prove the lemma if $q \notin T_i$ by contradiction. Assume $u \in T_i$ but $u \notin V_k$. Since $|V_k| + |T_i| \leq k$ and $u \notin V_k$, for at least one node $u' \in V \backslash \{V_k \cup T_i\}$, $\textit{dist}(q, u') < \overline{d}(r)$. This contradicts $\overline{d}(r) \leq d_{\min}$. Hence, Lemma 1 holds. □

Lemma 1 implies that we can prune the computations of trees that satisfy the conditions shown in the lemma. Our method (lines 20-25) prunes T_i without computing T_i by Lemma 1. Otherwise, it regards T_i as core nodes (lines 26-29), which are computed in the subsequent procedure (lines 12-16).

Finally, Algorithm 2 runs kNN search step (lines 8-17). Once $\langle u, \textit{dist}(q, u) \rangle$ is obtained from Q (line 9), it continues the kNN search until $|V_k|$ reaches k. If $f_l[u] = \textit{tree}$, the algorithm examines TREEPRUNING (lines 10-11) as well as the initialization step. Otherwise, it traverses the core nodes by updating their distances (lines 12-16).

After the termination, the following property holds:

Theorem 1 (Correctness)**.** *V_k obtained by Algorithm 2 is equivalent to kNN nodes searched on G.*

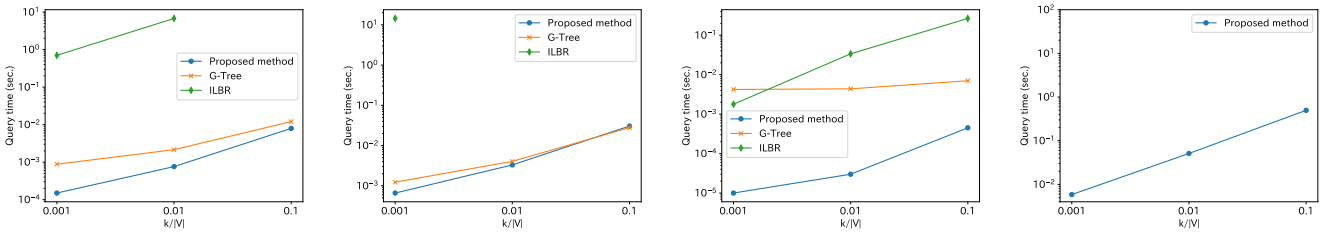
Proof: From Lemma 1, TREEPRUNING prunes T_i only if all nodes in T_i are included in V_k. Otherwise, the nodes are labeled as *core* by Algorithm 2 (line 28). Since it traverses all core nodes until $|V_k|$ reaches k, Theorem 1 holds. □

IV. EXPERIMENTAL ANALYSIS

Here, we experimentally discuss the efficiency of our algorithm in terms of the indexing time and kNN querying time.

A. Experimental Setting

Methods: We experimentally compared our method with two state-of-the-art indexing methods: G-Tree [3] and ILBR [6]. For G-Tree, we set $f = 4$ as well as the same setting employed in [3]. All algorithms were implemented by C++ and compiled by gcc 9.2.0 using -O2 option. All experiments were conducted on a server with an Intel Xeon CPU (2.60 GHz) and 128 GiB



(a) NY (b) FLA (c) TV (d) SP

Fig. 1: Effect of k for kNN query time

TABLE I: Statistics of real-world datasets.

Name	$\|V\|$	$\|E\|$	Type	Source
CAL	21,048	21,693	road network	[3]
NY	264,346	366,923	road network	[5]
FLA	1,070,376	2,712,798	road network	[13]
TV	3,892	17,262	social network	[14]
GV	7,057	89,455	social network	[14]
NS	27,917	206,259	social network	[14]
AT	50,515	819,306	social network	[14]
SP	1,632,803	22,301,964	social network	[14]

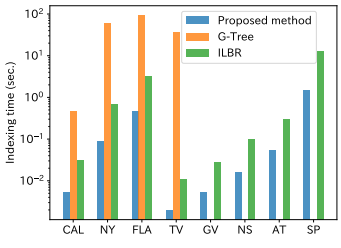


Fig. 2: Indexing time.

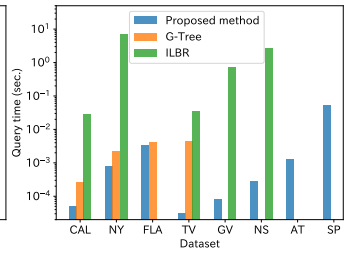


Fig. 3: kNN query time.

RAM. We randomly selected 30 query nodes for each dataset. Here, we report their average time. Unless otherwise stated, we set $k = 0.01 \times |V|$.

Datasets: We employed eight real-world graphs, which were published in previous studies [3], [5] and several public repositories [13], [14] (Table I). CAL, NY, and FLA denote a road network. All others indicate a social network.

B. Indexing Efficiency

Figure 2 shows the indexing time on the real-world datasets. The results of G-Tree are omitted from GV, NS, AT, and SP as indexing on the datasets was not finished within one hour. Our algorithm significantly outperforms G-Tree and ILBR under all examined conditions. It has an improved indexing efficiency up to four orders of magnitude higher than the state-of-the-art methods. For instance, our method is 18,074 times faster than G-Tree on TV. As a comparison of the indexing time by graph types (*i.e.*, road or social networks) demonstrates that our algorithm is better suited for social networks than road networks. This is because Algorithm 1 can reduce its running time if a graph contains many trees since each tree T_i requires only $O(|T_i|)$ time for distance pre-computations. Social networks inherit the core-tree property of complex networks, and most subgraphs of a social network are trees. Hence, our algorithm can significantly reduce the indexing time for social networks.

C. kNN Query Efficiency

Figure 3 shows the kNN query time when $k = 0.01 \times |V|$. Results are omitted if the kNN query did not finish within 1 minute or its index is not available in Figure 2. Our algorithm significantly outperforms the other methods for social networks. By contrast, its improvements are relatively small for road networks. This is because our algorithm can skip search trees by Lemma 1. As such, it can improve the indexing efficiency up to two orders of magnitude compared to the state-of-the-art methods on social networks. Our algorithm achieves quick kNN query processing for NS, AT, and SP, whereas the other methods failed to compute the query.

In Figure 1 shows the impact of k for query time. We varied k as $0.001 \times |V|$, $0.01 \times |V|$, and $0.1 \times |V|$. Due to the space limitations, we report results only for NY, FLA, TV, and SP. Results are omitted if the kNN query did not finished within 1 minute. Our method is significantly faster than the others on social networks regardless of k. By contrast, its improvements are small for road networks. If a graph has many trees such as those in social networks, our algorithm can drastically prune trees without computing nodes by Lemma 1. Consequently, our method is better suited for large-scale complex networks than state-of-the-art methods.

V. Conclusion

We propose a novel indexing algorithm to efficiently compute kNN queries on large-scale complex networks. Our algorithm separatedly constructs the indexes for a core and trees, reducing both the index construction time and kNN query time. Consequently, the computations for trees are dynamically pruned, while ensuring that the same results as a kNN search on a graph are returned. Our algorithm outperforms state-of-the-art methods in experiments by up to four orders of magnitude in terms of indexing and query processing time.

Acknowledgement: This work was partly supported by JSPS KAKENHI Grant-in-Aid for Young Scientists Grant Number 18K18057, and JST PRESTO JPMJPR2033, Japan.

REFERENCES

[1] H. Shiokawa, T. Amagasa, and H. Kitagawa, "Scaling Fine-grained Modularity Clustering for Massive Graphs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 7 2019, pp. 4597–4604.

[2] H. Shiokawa, "Scalable Affinity Propagation for Massive Datasets," *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2021)*, vol. 35, no. 11, pp. 9639–9646, May 2021.

[3] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong, "G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 8, pp. 2175–2189, August 2015.

[4] H. Shiokawa, "Fast ObjectRank for Large Knowledge Databases," in *Proceedings of the 20th International Semantic Web Conference (ISWC 2021)*, 2021.

[5] Z. Li, L. Chen, and Y. Wang, "G*-Tree: An Efficient Spatial Index on Road Networks," in *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019)*, 2019, pp. 268–279.

[6] T. Abeywickrama and M. A. Cheema, "Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks," in *Proceedings of the 22nd International Conference on Database Systems for Advanced Applications (DASFAA 2017)*, 2017, pp. 425–440.

[7] G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning," in *Proceedings of the IEEE/ACM SC95 Conference (SC 1995)*, 1995, pp. 29–es.

[8] J. Wang, S. Anirban, T. Amagasa, H. Shiokawa, Z. Gong, and M. S. Islam, "A Hybrid Index for Distance Queries," in *Proceedings of the 21st International Conference on Web Information Systems Engineering (WISE 2020)*, 2020, pp. 227–241.

[9] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-Scale Graphs," *PVLDB*, vol. 8, no. 11, p. 1178–1189, July 2015.

[10] Y. Shavitt and T. Tankel, "Hyperbolic Embedding of Internet Graph for Distance Estimation and Overlay Construction," *IEEE/ACM Transactions on Networking*, vol. 16, no. 1, p. 25–36, February 2008.

[11] A. Benson and J. Kleinberg, "Link Prediction in Networks with Core-Fringe Data," in *Proceedings of The Web Conference 2019 (WWW 2019)*, 2019, p. 94–104.

[12] H. Shiokawa, Y. Fujiwra, and M. Onizuka, "Fast Algorithm for Modularity-based Graph Clustering," in *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013)*, 2013, pp. 1170–1176.

[13] C. Demetrescu, "The 9th DIMACS Implementation Challenge," http://users.diag.uniroma1.it/challenge9/download.shtml, June 2010.

[14] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, June 2014.