




# An Efficient Indexing Method for Dynamic Graph $k$ NN

Shohei Matsugu<sup>1</sup>, Suomi Kobayashi<sup>2</sup>, and Hiroaki Shiokawa<sup>1</sup> 

<sup>1</sup> Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan  
matsugu@kde.cs.tsukuba.ac.jp, shiokawa@cs.tsukuba.ac.jp

<sup>2</sup> Graduate School of Science and Technology, University of Tsukuba,  
Tsukuba, Japan  
kobayashi@kde.cs.tsukuba.ac.jp

**Abstract.**  $k$ -nearest neighbor ( $k$ NN) search is a fundamental problem in graph mining. This search finds the  $k$  most relevant nodes to a given query node. The increased use of social network services and map applications due to the proliferation of mobile devices has necessitated faster searches. Although pre-constructing an index using graphs can accelerate a  $k$ NN search, existing methods struggle handling dynamic graph updates. Herein we propose an efficient index update method for dynamic graphs that utilizes a core-tree structure to efficiently update the index in response to dynamic changes in the graph. Our experimental analysis using real-world data demonstrated that the proposed method can construct indexes more efficiently than the state-of-the-art method.

**Keywords:** graph algorithm ·  $k$ NN search · dynamic graph

## 1 Introduction

The  $k$  nearest neighbor ( $k$ NN) search [12, 13, 17, 21] is a fundamental graph analysis tool to understand complex networks. It finds the  $k$  nearest neighbor nodes to a user-specified query node in a given graph.  $k$ NN searches are employed in diverse applications [2, 6]. Although they only need to perform a local search near the query node,  $k$ NN searches struggle handling real-world networks.

Although  $k$ NN queries are useful in many applications, they have critical drawbacks in handling real-world networks. Specifically, they require a large computational time to answer  $k$ NN queries due to the size and density of the real-world networks [15]. Historically, traditional  $k$ NN search methods are applied to small graphs such as ego-networks and road networks [4]. These methods struggle to quickly compute  $k$ NN with  $10^4$  nodes [21]. Recent applications based on social networks require handling large and complex networks with  $10^6$  nodes [14, 19], compounding the high computational costs to find  $k$ NN nodes.

### 1.1 Existing Approaches and Challenges

Many studies have strived to overcome the aforementioned drawbacks. One approach category is *graph indexing methods* [1, 13, 21]. These methods construct an

index using graph partitioning, which enables the shortest path distances among several nodes to be pre-computed before answering a query. Examples of graph indexing methods are G-Tree [21] and ILBR [1]. G-Tree partitions a given graph into disjointed subgraphs using Metis [8]. Then the index is constructed from the shortest path distances among all node pairs in each subgraph. Similarly, ILBR selects several landmark nodes and constructs the index from the ALT [7] values according to the shortest path distances between a node and each landmark node. Although G-Tree and ILBR improve the computation time to answer a  $k$ NN query, they still suffer from a large indexing time as they mainly focus on handling planar graphs [3] with a low density. Their indices are efficient if a given graph is sparse. However, indices in a dense graph are not computed effectively since an exhaustive pre-computation is necessary. These methods also require high computation costs for querying  $k$ NN since the coverages of the indices are too small for non-planar graphs.

CT [9,10] is a state-of-the-art  $k$ NN search method using a *core-tree-aware (CT) index* based on the *core-tree property* [5]. The core-tree property is expressed as a graph comprised of a *core* and *trees*. The core is a small and dense subgraph, while trees are long stretched and sparse subgraphs. On the basis of the core-tree property, CT constructs a core-index and a tree-index by compiling each part of the graph. Specifically, for each tree in the graph, the tree-index stores the distances from its root node to all leaf nodes. Furthermore, core-index has the distances among the remaining non-tree nodes. Using the two indices, CT can realize efficient indexing and querying for a  $k$ NN search.

Although CT can efficiently perform a  $k$ NN search, it has serious drawbacks in real-world  $k$ NN applications. CT cannot respond to node or edge updates. However, real-world graphs are frequently updated. For example, in social networks, a new edge is linked if two users become friends. If a new edge is added, CT must reconstruct the indices from scratch since it cannot update its differences. Thus, CT fails to efficiently construct the CT-index for practical use.

## 1.2 Our Approaches and Contributions

Our goal is to extend CT to dynamic graphs. Although the CT-index is efficient for static graphs, the index must be reconstructed to update only a few nodes and edges. Here, we present a novel indexing method called *Dynamic CT (DCT)*. The underlying idea is to update only the indices that include changed nodes. This way the tree-index is always maintained to include only the tree structures. To update the index, the process is classified according to whether the changed node is included in the core-index or the tree-index. Consequently, DCT has the following attractive characteristics:

- **Efficiency:** DCT achieves faster updates than CT (Sect. 4). DCT can perform the difference computation up to 4,462 times faster than the reconstruction of CT.
- **Exactness:** We theoretically verify that DCT always outputs the same indices as CT while achieving indexing-time improvements. It can calculate the correct  $k$ NN search results using this index.

- **Easy to deploy:** DCT does not require new indices or pre-computation processing. It achieves index updating by quickly editing the constructed CT-index.

DCT is the first solution to realize  $k$ NN searches assuming graph updates. Our extension not only increases the utility of a graph query using a  $k$ NN search but also enhances the application scope of  $k$ NN searches.

## 2 Preliminary

We formulate the  $k$ NN querying problem in Sect. 2.1. Then we briefly explain CT, the state-of-the-art  $k$ NN search method, in Sect. 2.2. Due to space limitations proofs of lemmas and theorems are omitted.

### 2.1 Problem Definition

Here, let  $G = (V, E, W)$  be a weighted, undirected, and connected graph, where  $V, E$ , and  $W$  are the sets of nodes, edges, and edge-weight values, respectively.  $e(u, v) \in E$  denote that two nodes  $u$  and  $v$  are linked in  $G$ . For each edge  $e(u, v) \in E$ , an edge-weight value  $w(u, v) \in W$  is always assigned, where  $w(u, v) \in \mathbb{N}$  holds. The degree of node  $u$  is denoted as  $deg(u)$ .

$k$ NN is a task to find  $k$  nearest neighbor nodes to a query node. We first define the shortest path distance as:

**Definition 1 (Shortest path distance).** *Let a node path  $u = u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_i = v$  in  $G$  be the shortest path between the nodes  $u, v \in V$ . Here, the distance of this shortest path is defined as  $dist(u, v) = \sum_{j=0}^{i-1} (w(u_j, u_{j+1}))$ . Moreover,  $dist_k(q, V)$  represents the  $k$ -th smallest distance in  $\{dist(q, v) \mid v \in V\}$ .*

By using Definition 1, we formulate the  $k$ NN query problem as:

**Problem 1 ( $k$ NN query processing).** *Given a graph  $G = (V, E, W)$ , a query node  $q \in V$ , and an integer  $k \in \mathbb{N}$ , the  $k$ NN query is to find a node set  $V_k(q) = \{v \in V \mid dist(q, v) \leq dist_k(q, V)\}$ .*

### 2.2 Previous Method: CT Index

We briefly present a *core-tree-aware (CT) indexing method* [9, 10], which is the state-of-the-art method solving Problem 1.

As mentioned in Sect. 1.1, real-world graphs often follow the core-tree property; the graphs can be decomposed into a core and trees [5]. Using this, CT constructs a CT index  $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$ , where  $\mathcal{T}$  and  $\mathcal{C}$  are the tree-index and core-index, respectively. CT first extracts trees from a graph and indexes them in  $\mathcal{T}$ . Then it stores the remaining core nodes in  $\mathcal{C}$ . The tree-index  $\mathcal{T}$  and the core-index  $\mathcal{C}$  are defined as follows:

**Definition 2 (Tree-index  $\mathcal{T}$ ).** Let  $T_1, T_2, \dots$  be the trees in  $G$  and  $r_i$  be the root node of  $T_i$ . Then we denote  $D_i$  as a set of distances between  $r_i$  and each node  $v \in T_i$ , i.e.,  $D_i = \bigcup_{v \in T_i} \{dist(r_i, v)\}$ . We define the tree-index as  $\mathcal{T} = (\mathbb{T}, \mathbb{D})$ , where  $\mathbb{T}$  and  $\mathbb{D}$  represent the sets of  $T_i$  and  $D_i$ , respectively, i.e.,  $\mathbb{T} = \{T_1, T_2, \dots\}$  and  $\mathbb{D} = \{D_1, D_2, \dots\}$ .

**Definition 3 (Core-index  $\mathcal{C}$ ).** Let  $V_c$  be a set of core nodes that are not included in any tree-index. We define the core-index as  $\mathcal{C} = (V_c, E_c, W_c)$ , where  $E_c = \{e(u, v) \in E \mid u, v \in V_c\}$ , and  $W_c = \{dist(u, v) \mid e(u, v) \in E_c\}$ .

Note that the shortest path between two nodes may have multiple routes. To efficiently compute  $W_c$ , CT employs the Dijkstra algorithm to update the weight values.

For convenience, we introduce the following notation:

**Definition 4 (Label function).** Given a node  $u$ , the label function  $f_l(u) = \text{tree}$  if  $u \in \mathbb{T}$  holds, and  $f_l(u) = \text{core}$  otherwise.

On the basis of the CT index, CT searches  $k$ NN nodes by applying the following lemma:

**Lemma 1.** Given a root node  $r_i$  in the tree  $T_i$ , and let  $d_{min} = \min\{dist(q, v)\}$ , where  $v \in Q \cup \{v \mid e(r_i, v) \in E_c, v \notin V_k(q)\}$ . If  $|V_k(q)| + |T_i| \leq k$  and  $d_{max}(r_i) \leq d_{min}$  hold, then  $T_i \subseteq V_k(q)$  holds, where  $d_{max}(v)$  is defined as the maximum shortest path distance from  $v$  to any node in  $T_i$ .

**The Overview of CT:** We explain the basic procedure of CT to efficiently compute  $k$ NN using the CT index. CT uses a priority queue  $Q$  to calculate the shortest path, which is similar to the Dijkstra algorithm. If  $q$  is included in the tree-index  $T_i$  in  $\mathcal{T}$ , it initially pushes  $r_i$  into  $Q$  and core-index  $\mathcal{C}$ . CT initially pushes the query node  $q$  into  $Q$ . Then it repeatedly searches  $k$ NN nodes from  $q$  using  $Q$  until it explores  $k$  nodes or reaches any root node  $r_i$  in the tree  $T_i$ . Once CT finds  $r_i$ , CT checks whether  $V_k(q)$  contains all  $T_i$ . If so, CT includes  $T_i$  in  $V_k(q)$  without searching  $T_i$ . Otherwise, CT explores  $T_i$  in the same way as the core node.

The main feature of CT is its application of an index construction algorithm specialized for planar graphs (e.g., traditional road networks) to more dense and diversely structured complex networks [16, 20]. However, CT is not adaptable to dynamic graphs that exist in the real world [18] as each graph update requires index reconstruction. Consequently, CT can incur a significant computation time even for minor graph changes.

### 3 Proposed Method: Dynamic CT

Here, we propose a novel method *Dynamic CT (DCT)* which extends CT to dynamic graphs. Here, we describe the concept of DCT and then provide details.

### 3.1 Ideas

We propose a method to dynamically update the CT index. In general, graph updates are represented as a collection of node or edge additions and removals. Although we focus on designing a method that accelerates the addition and removal of single edges, the proposed method does not lose generality because equivalent transformations are possible to add or remove nodes. The required processing for index updates depends on the type of transformation and the edge location (core or tree). Specifically, edge updates can change the core/tree state.

The area of this change must be limited for efficient dynamic updates. Therefore, we divide additions into four cases (Sect. 3.2) and removals into two cases (Sect. 3.3). We also theoretically calculate the update range of the graph for each case. For convenience, in the following sections, the parent node refers to the node closest to the root node on the path to any other node in the tree.

Our ideas have two advantages. First, DCT directly calculates only the index difference. By contrast, CT completely reconstructs the index for each graph update. Thus, DCT efficiently constructs the index for a graph  $k$ NN. Second, DCT always outputs the same index to CT while omitting redundant calculations. This is because DCT performs index restructuring by theoretically analyzing the area of the index affected by graph updates. Consequently, the obtained index is always the same as that constructed by CT from scratch.

### 3.2 Adding Nodes and Edges

We propose an algorithm that dynamically updates the CT index when nodes and edges are added to a graph. Here, only the addition of edges is considered. This is reasonable since adding a node is meaningless until an adjacent edge is added and it does not need to be distinguished from an isolated node that was already present.

The addition of an edge between nodes  $u$  and  $v$  can be classified into four cases:

**Case 1.**  $u, v \in T_{uv}$ :

**Case 2.**  $u \in T_u$ , and  $v \in T_v$ :

**Case 3.**  $u \in V_c$ , and  $v \in T_v$ :

In these three cases, we have the following property.

**Lemma 2.** *In Cases 1–3,  $tree(s)$  are no longer a tree after adding an edge.*

Lemma 2 indicates that the tree is no longer a tree structure when an edge is added by constructing a cycle. Then we add a new cycle to the core-index  $\mathcal{C}$ . In Case 1, the cycle occurs in a tree. In Case 2, the cycle occurs across two trees. In Case 3, the cycle occurs through the tree and the core. In every case, this new cycle is added to the core-index  $\mathcal{C}$ , and the tree-index  $\mathcal{T}$  is reconstructed.

**Case 4.**  $u, v \in V_c$ :

In this case, we simply add an edge between nodes  $u$  and  $v$  since the new edge does not affect the tree-index  $\mathcal{T}$ .

Using Lemma 2, we design a novel algorithm that adds an edge to the CT-index.

**Algorithm Overview:**  $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$  is updated using a two-fold process: linking nodes and restructuring trees. The linking step divides the cases by the location of the edge to be added. The edge is in the core or the tree. In each case, it calculates the cycle appearing through the tree-index and restructures to move it to the core-index. In the restructuring step, nodes are moved from a tree to the core. Then the node is recursively merged into its parent node for each leaf node in the updated trees.

### 3.3 Removing Nodes and Edges

We also introduce an algorithm to dynamically update the CT index when nodes and edges are removed from the given graph. Here, we consider only the removal of edges. Similar to adding edges, removal of a node is equivalent to the removal of all edges linked to it.

The removal of an edge between nodes  $u$  and  $v$  can be classified into two cases:

**Case 1.**  $u, v \in V_c$ : For  $u$ , if  $\text{deg}(u) = 1$  holds after the removal,  $u$  and only its adjacent node  $w$  become part of the tree. In this case,  $u$  must be moved into  $\mathcal{T}$ . Thus, DCT performs restructuring from  $u$  as a leaf node. By performing the same process for  $v$ ,  $\mathcal{T}$  can have the new tree that has become a tree due to the removal of the edge. By performing the same process for  $v$ ,  $\mathcal{T}$  can become a new tree due to edge removal.

**Case 2.**  $u, v \in T_{uv}$ : As mentioned in Sect. 2, we assumed the graph is connected. Thus, we remove nodes that are no longer connected to the core from the graph.

**Algorithm Overview:** To update  $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$  in DCT, the edge is initially removed. Then the cases are divided by the location of the edge to be removed. The edge is either in the core or the tree. For an edge in the core, DCT restructures if  $\text{deg}(u) = 1$  or  $\text{deg}(v) = 1$  holds. For an edge in a tree, DCT removes a disconnected path in the leaf side.

### 3.4 Complexity Analysis

Finally, we discuss the time complexity of DCT.

**Theorem 1.** *Updating  $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$  after adding an edge incurs  $O(|\bar{T}|)$  time on average, where  $|\bar{T}|$  represents the average size of trees in  $\mathcal{T}$ .*

**Theorem 2.** *Updating  $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$  after removing an edge incurs  $O(\max(|V_c|, |\bar{T}|))$  time on average.*

From Theorems 1 and 2, DCT can efficiently update the index for graph changes. According to [9, 10], the index construction of CT requires  $O(|E| \log |V|)$ . Since  $|V| > |V_c|$  and  $|V| > |\bar{T}|$  hold in general, our proposed method significantly improves the computational complexity required for reconstruction compared to the state-of-the-art method.

## 4 Experimental Evaluation

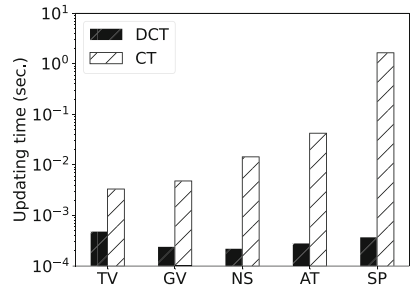
We experimentally evaluated the efficiency of DCT compared to CT [10].

**Datasets:** We tested five real-world social networks [11] used in previous works [10]<sup>1</sup>. Table 1 shows the size of the datasets.

**Experimental Setup:** We set  $k = 0.01|V|$  as default. All experiments were conducted on a Linux server with Intel Xeon CPU 2.60 GHz and 128 GiB RAM. All algorithms were implemented in C++ using “-O2” option. We compared the running time for 100 random edge additions and removals each.

**Table 1.** Statistics of real-world datasets.

	$ V $	$ E $	average degree
TV	3,892	17,262	4.4
GV	7,057	89,455	12.7
NS	27,917	206,259	7.3
AT	50,515	819,306	16.2
SP	1,632,803	22,301,964	13.7



**Fig. 1.** Efficiency for updating edges.

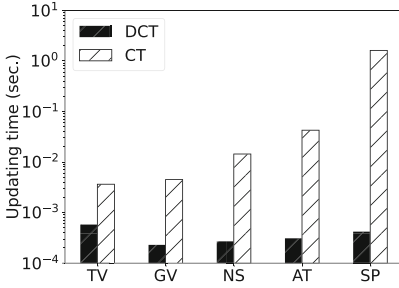
### 4.1 Efficiency for Updating

Figure 1 shows the indexing times to add and remove edges as  $m = 200$ . DCT achieves significantly faster indexing for graph updates by reducing the reconstruction cost using dynamic index updates. Furthermore, CT suffers from a significant reconstruction time for large graphs because computations are performed over the entire graph. By contrast, DCT only requires computations within the neighborhood of the updated subgraph. Thus, the average degree of the graph primarily affects the update time in DCT. DCT guarantees the same results as the CT algorithm but is up to 4,462 times faster than CT.

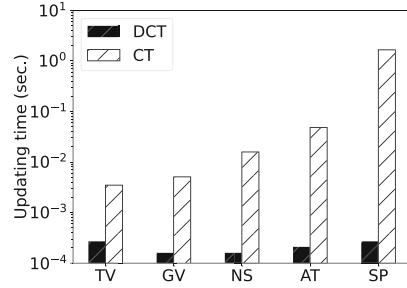
### 4.2 Efficiency for Adding/Removing Edges

Figures 2 and 3 plot the index update times to add/remove only edges. DCT effectively adds and removes edges. More time is required for edge addition than removal due to the difference in the affected area for dynamic indexing. Restructuring for edge removal requires merging two paths, at most, whereas that for edge addition may involve large-scale updates that include the core and its surrounding trees.

<sup>1</sup> All graphs are publicly available online from <http://snap.stanford.edu/data>.



**Fig. 2.** Efficiency for only adding edges.



**Fig. 3.** Efficiency for only removing edges.

## 5 Conclusion

Herein a novel dynamic index update algorithm, DCT, is proposed to efficiently compute  $k$ NN searches on large-scale complex graphs. The proposed method limits the affected area of the index when the graph is updated, significantly reducing the computation time to reconstruct indexes. In our experiments, the proposed method outperforms the state-of-the-art method by up to four orders of magnitude in terms of processing times for index construction and graph  $k$ NN searches. Hence, the proposed method, which considers the core-tree characteristics, effectively reduces the cost of a  $k$ NN search on real-world dynamic graphs.

**Acknowledgement.** This work was partly supported by JSPS KAKENHI Grant Numbers 22K17894, 22J10972, 22KJ0398, JST PRESTO Grant Number JPMJPR2033, and JST AIP Acceleration Research JPMJCR23U2.

## References

1. Abeywickrama, T., Cheema, M.A.: Efficient landmark-based candidate generation for  $k$ NN queries on road networks. In: Candan, S., Chen, L., Pedersen, T.B., Chang, L., Hua, W. (eds.) DASFAA 2017. LNCS, vol. 10178, pp. 425–440. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-55699-4\\_26](https://doi.org/10.1007/978-3-319-55699-4_26)
2. Alom, Z., Carminati, B., Ferrari, E.: Detecting spam accounts on Twitter. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 1191–1198. IEEE (2018)
3. Barthélemy, M.: Morphogenesis of Spatial Networks. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-20565-6>
4. Bast, H., Funke, S., Matijević, D.: Ultrafast shortest-path queries via transit nodes. In: The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, 13–14 November 2006, vol. 74, pp. 175–192. DIMACS/AMS (2006)
5. Benson, A., Kleinberg, J.: Link prediction in networks with core-fringe data. In: The World Wide Web Conference, pp. 94–104 (2019)



6. Chen, Z., Li, P., Xiao, J., Nie, L., Liu, Y.: An order dispatch system based on reinforcement learning for ride sharing services. In: 2020 IEEE 22nd International Conference on High Performance Computing and Communications, pp. 758–763 (2020)
7. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: SODA, vol. 5, pp. 156–165 (2005)
8. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: Proceedings Supercomputing '95, San Diego, CA, USA, 4–8 December 1995, p. 29. ACM (1995)
9. Kobayashi, S., Matsugu, S., Shiokawa, H.: Indexing complex networks for fast attributed  $k$ NN queries. *Soc. Netw. Anal. Min.* **12**(1), 82 (2022)
10. Kobayashi, S., Matsugu, S., Shiokawa, H.: Fast indexing algorithm for efficient  $k$ NN queries on complex networks. In: Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, pp. 343–347. ASONAM '21, Association for Computing Machinery, New York, NY, USA (2022)
11. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection (2014). <http://snap.stanford.edu/data>
12. Li, R.H., Qin, L., Yu, J.X., Mao, R.: Influential community search in large networks. *Proc. VLDB Endow.* **8**(5), 509–520 (2015)
13. Li, Z., Chen, L., Wang, Y.: G\*-tree: an efficient spatial index on road networks. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 268–279. IEEE (2019)
14. Matsugu, S., Fujiwara, Y., Shiokawa, H.: Uncovering the largest community in social networks at scale. In: Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI2023), pp. 2251–2260 (2023)
15. Matsugu, S., Shiokawa, H., Kitagawa, H.: Fast and accurate community search algorithm for attributed graphs. In: Hartmann, S., Küng, J., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) Database and Expert Systems Applications. DEXA 2020. LNCS, vol. 12391, pp. 233–249. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-59003-1\\_16](https://doi.org/10.1007/978-3-030-59003-1_16)
16. Onizuka, M., Fujimori, T., Shiokawa, H.: Graph partitioning for distributed graph processing. *Data Sci. Eng.* **2**(1), 94–105 (2017)
17. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 43–54 (2008)
18. Shiokawa, H.: Scalable affinity propagation for massive datasets. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, vol. 35, no. 11, pp. 9639–9646 (2021)
19. Shiokawa, H., Amagasa, T., Kitagawa, H.: Scaling fine-grained modularity clustering for massive graphs. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence, pp. 4597–4604. IJCAI'19 (2019)
20. Shiokawa, H., Fujiwara, Y., Onizuka, M.: Scan++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proc. VLDB Endow.* **8**(11), 1178–1189 (2015)
21. Zhong, R., Li, G., Tan, K.L., Zhou, L., Gong, Z.: G-Tree: an efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.* **27**(8), 2175–2189 (2015)