SgIndex: An Index Structure Supporting Multiple Graph Queries

Shibiao Zhu, Yuzhou Huang, Zirui Zhang, and Xiaolin Qin(✉)

Nanjing University of Aeronautics and Astronautics, Nanjing, China
qinxcs@nuaa.edu.cn

Abstract. With the rise of social networks, traffic navigation and other fields, graph applications become increasing extensive. To improve query efficiency, indexes are built to manage large-scale graph data. However, these indexes cost large memory space and can only support one single graph operation. We propose a two-layered index structure SgIndex, in which the first layer stores subgraph information, and the second layer stores adjacency information to support multiple path operations and subgraph matching queries. We propose a subgraph matching algorithm based on path join, which completes subgraph matching by searching SgIndex twice. The experimental results show that SgIndex achieves better performance on path queries and subgraph matching than existing index structures, and reduces memory overhead.

Keywords: Index · Graph data · Path query · Subgraph matching

1 Introduction

Large-scale graph data with complex internal structure and diverse query requirements have emerged in different fields and various data operations have appeared in the application of large-scale graph data [2,11,15]. On the one hand, some algorithms have difficulties in adapting to large-scale graphs. On the other hand, most of the existing index structures lack generality. In practical applications, multiple indexes need to be established for the same graph to response to different query requirements. Therefore, this paper proposes a subgraph-based index structure SgIndex: for large-scale graph data, establish an index structure that meets various query requirements.

First, graph operations are expensive on large-scale graph data. Compared with traditional relational data or XML tree, graph data lacks structural constraints and is complicated to operate. Second, flexible and diverse query requirements for large-scale graph data mean that multiple indexes need to be built to meet these requirements. Furthermore, graph data operations usually require loop iterations, which is also the part that our index needs to support.

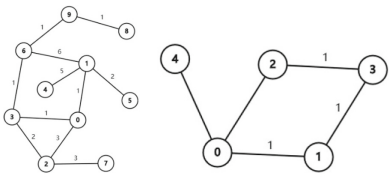
We have made the following contributions in this paper: 1. We propose an index that supports large-scale datasets, which reduces storage space by optimizing the design of the index; 2. The index structure can implement various

B. Li et al. (Eds.): APWeb-WAIM 2022, LNCS 13421, pp. 553–561, 2023.
https://doi.org/10.1007/978-3-031-25158-0_45

queries on graph data such as path query and subgraph query, including two-point path query, path query with limited path length, path query with limited path hops, and subgraph query.

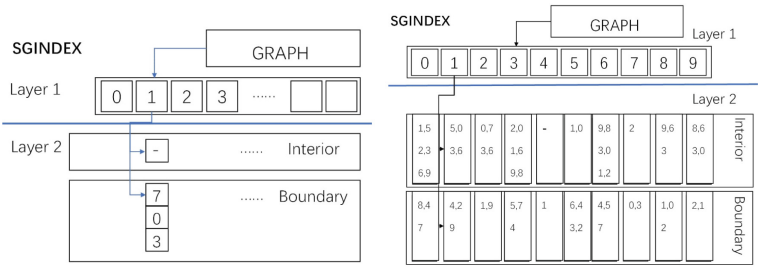
2 Related Work

Graph is already a widely used data structure, but querying large-scale graph data has always been a difficult problem [5]. A popular solution is to build an index that does not waste too much storage space, but also produces effective positive feedback on the query process under various circumstances. However, in practical applications, there are often multiple queries that need to be satisfied for the same graph data [2,14,15] (Fig. 1). Given a directed graph G and two vertices s and t in it, a reachability query asks G if there is a path from s to t. Reference [13] proposes a set of tools to quantify and query network reachability, and uses decision graphs as data structures to represent reachability matrices. Based on two classical shortest path algorithms, the shortest path problem can be decomposed into a linear complex problem [1,8], or the target solution can be optimized in terms of distance or time [7,9,10], or one or more preprocessing steps to speed up the shortest path query time [4,12]. Subgraph is one of the basic concepts of graph theory, which refers to a graph in which the vertex set and the edge set are subsets of the vertex set and edge set of a certain graph, respectively. Reference [3] adopts a left-side deep join ranking strategy, which



(a) Graph (b) Query Graph

Fig. 1. Graph example and query graph example



(a) SgIndex Structure (b) SgIndex Example

Fig. 2. SgIndex structure diagrame

models the enumeration process as a join problem. Subtree features are also used for indexing, and they take less time to index than more general subgraph features. BINDEX [6] is a secondary index with excellent query efficiency. It consists of Filter Layer and Refine Layer.

3 Index Based on Subgraph

3.1 Index Structure and Establishment Method

Algorithm 1. Create SgIndex

Input: Graph data $G(V_G, E_G)$; Vertex set V_G; Edge set E_G; Threshold r;
Output: Index I;

```
1: i ← 0;
2: while i < |E_G| do
3:     if e_i.weight < r then
4:         Put e_i.end into the interior collection of e_i.start;
5:     else
6:         Put e_i.end into the boundary collection of e_i.start;
7:     i++;
8: i ← 0;
9: while i < |V_G| do
10:    Get interior collection of v_i as IN_i;
11:    j ← 0;
12:    while j < |IN_i| do
13:        Get interior node of v_j;
14:        Calculate the sum of the weights w;
15:        if w < r then
16:            Put the interior node of v_j into the interior collection of v_i;
17:        else
18:            Put the interior node of v_j into the boundary collection of v_i;
19:        j++;
20:    i++;
```

SgIndex is a subgraph-based index structure. This index has two layers. First, we need to locate each vertex of the graph to the head of the adjacency list by hash. Secondly, for each vertex, we store interior vertex information and boundary vertex information for it, as shown in Fig. 2(a). Both types of vertex information include vertex value, path weight, and path hop. For each vertex, its internal vertices are vertices in the local subgraph, and the vertices directly adjacent to subgraph are classified as boundary vertices. The scale of the subgraph is denoted as threshold.

The selection of the threshold is not an optimal problem. For the currently implemented query, the larger the threshold, the faster the query speed. The only cost is the space occupied by the storage index and the cost of index establishment. Although for local information query, such as general contact query of the new crown epidemic, fast query can be guaranteed as long as the limit threshold is a specified number of hops, but in practical applications, it is often impossible to easily predict the required query scale, so the threshold should be chosen based on acceptable storage overhead.

Algorithm 1 gives the pseudocode of the new index algorithm. First we fill the edge information of the graph into the index (lines 1–7). Second we iterate over the interior vertices of each vertex until the complete subgraph information is stored (lines 8–20). We can build an index as shown in Fig. 2(b).

3.2 Path Query Algorithm

Algorithm 2. Path Query

Input: SgIndex I; Start Point *start*; End Point *end*; Empty List *path*;
Output: Result Path List *path*;

```
1: Get IN_s, the interior vertex set of start, from I;
2: Get BN_s, the boundary vertex set of start, from I;
3: if end ∈ IN_s then
4:     return path;
5: else
6:     i ← 0;
7:     while i < |BN_s| do
8:         path ← FindPath(I, v_i, end, path);
9:         if path is found then
10:            return path;
11:        i++;
```

Algorithm 2 shows how to find a path of two points. First we query the index and get the interior set and boundary set of the starting vertex (lines 1–2). If the interior set contains the end vertex *end*, it will return *path* directly (lines 3–4). If not, it will iteratively search in the boundary set, and each round of iteration will use the vertex in the boundary set as the new starting vertex to perform a new path query, until the path is found (lines 5–11). The function $FindPath(I, v_1, v_2, path)$ indicates that by querying the index I, it is judged whether v_2 is in the interior set of v_1. If it exists, *path* is updated. If it does not exist, the next iteration will be performed in the boundary set of v_2.

Algorithm 3. Subgraph Matching

Input: SgIndex I; Query Graph G_q;
Output: Result Subgraph G_s;

1: Decompose subgraph G_q into set $Path$ and get join condition $Join$;
2: $i \leftarrow 0$;
3: **while** $i < |Path|$ **do**
4: $\quad CandidatePath_i \leftarrow Searchpath(I, path_i.hop, path_i.weight)$;
5: $\quad$ i++;
6: $i \leftarrow 0$;
7: **while** $i < |CandidatePath_0|$ **do**
8: $\quad j \leftarrow 0$;
9: $\quad$ **while** $j < |CandidatePath_1|$ **do**
10: $\quad\quad$ **if** $(candidatepath_{0_i}, candidatepath_{1_j}) \in Join$ **then**
11: $\quad\quad\quad G_s \leftarrow Join(candidatepath_{0_i}, candidatepath_{1_j})$;
12: $\quad\quad$ j++;
13: $\quad$ i++;
14: $iternum \leftarrow 2$;
15: **while** $iternum < |Path|$ **do**
16: $\quad G_s \leftarrow Union(G_s, CandidatePath_{internum})$;
17: $\quad iternum++$;
18: return G_s;

3.3 Subgraph Matching Algorithm

The subgraph matching algorithm based on this index can be divided into three steps: 1. Decompose the query graph into multiple paths (line 1); 2. Generate an n-tuple for each path by index (lines 2–5); 3. Compare n-tuples by index to get the resulting subgraphs (lines 6–18).

The function $Searchpath(I, h, w)$ means to search for paths with $v_i.hop = h$ and $v_i.weight = w$ through index I. The function will output a collection of paths. The function $Join(path_1, path_2)$ means that the two paths and the join condition are combined into a subgraph, then the subgraph will be added to the subgraph set G_s. The function $Union(G_s, Path)$ compares subgraphs in the set G_s with paths in the set $Path$, and unions sunbgraphs and paths that meet the join conditions.

4 Experiments

4.1 Experimental Environment and Dataset

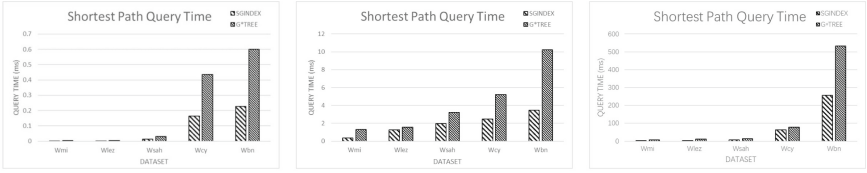
The machine used in this experiment has Intel(R) Core(TM) i5-10300H CPU @ 2.50 GHz processor, 16.0 GB memory, 64-bit operating system. The compiler used is Visual Studio 2019.

The datasets are all from KONECT, respectively: wikipedia_link_mi(Wmi), wikipedia_link_lez(Wlez), wikipedia_link_sah(Wsah), wikipedia_link_cy(Wcy), wikipedia_link_bn(Wbn). The scale of these datasets is shown in Table 1 (Fig. 4).

Table 1. Datasets

Name	Vertices	Edges
Wmi	7,996	116,464
Wlez	5,171	204,133
Wsah	15,531	352,209
Wcy	142,648	2,967,435
Wbn	226,501	2,832,143

4.2 Path Query



(a) Shortest Path Query Time (Short Path) (b) Shortest Path Query Time (Long Path) (c) Shortest Path Query Time (No Path Exists)

Fig. 3. Shortest path query

(a) Shortest Path Query Time (Short Path) (b) Shortest Path Query Time (Long Path) (c) Shortest Path Query Time (No Path Exists)

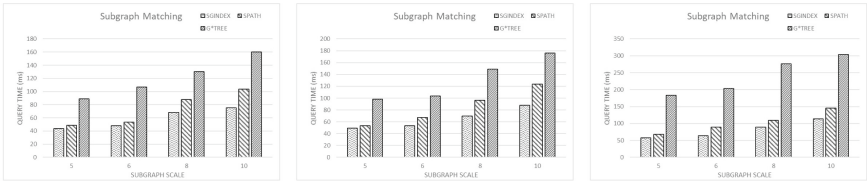
Fig. 4. Shortest path query (compare with system)

Figure 3(a) illustrates that the shortest path query time increases with the size of the dataset. For the smallest dataset Wmi, the query time on SgIndex is 50% of that on G*Tree; while for the largest dataset Wbn, the query time required on SgIndex is 35% of that on G*Tree. SgIndex performs better on larger datasets, which is determined by the structure of the bi-level index. SgIndex summarizes and stores subgraph information so that the shortest path can be obtained without traversing the entire graph. Figure 3(b) shows that SgIndex achieves better results in longer paths. Figure 3(c) is our test for extreme cases. When the required path does not exist, we need to traverse at least all the successor vertices of the starting vertex. Experiments show that SgIndex has withstood this

test and achieved an advantage of more than 50%. Figure 3(d) is a comparison between using our index structure and using a mature graph data management system.

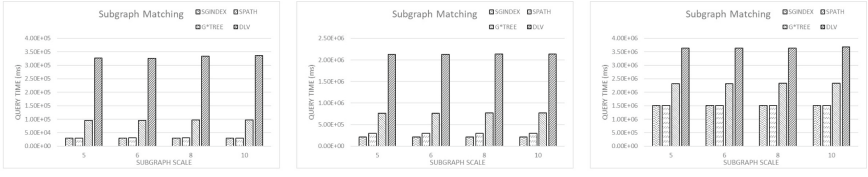
4.3 Subgraph Matching

We select the classic SPATH index in the field of subgraph matching and the above G*Tree as a comparison, and use the above real graph data to conduct experiments. Since SPATH takes up too much memory on large graphs, it exceeds the limit supported by our experimental environment. We only conduct experiments on three datasets, Wmi, Wlez and Wsah. We select subgraphs with 5, 6, 8, and 10 vertices for experiments. As can be seen from Fig. 5, SgIndex is more efficient than SPATH in subgraph matching query, especially with the increase of subgraph size. This is because the index established by SPATH still adopts the traditional method of finding candidate vertices and pruning them one by one. However, our method combines the advantages of indexing and invokes indexing in both the search candidate path stage and the pruning stage, thereby improving the overall query efficiency. Figure 6 shows how our index compares to different systems.



(a) Subgraph Matching on Wmi (b) Subgraph Matching on Wlez (c) Subgraph Matching on Wsah

Fig. 5. Subgraph matching



(a) Subgraph Matching on Wmi (b) Subgraph Matching on Wlez (c) Subgraph Matching on Wsah

Fig. 6. Subgraph matching (compare with system)

5 Conclusion

From the perspective of practical application, we propose an index that supports large-scale data sets, and reduces the storage space from the design of the index; at the same time, the index structure we propose is a general index structure that can realize queries on various graph data. Experiments show that our index has advantages in both storage space and query efficiency.

Acknowledgement. This work was supported by the National Natural Science Foundation of China (61972198).

References

1. Arz, J., Luxen, D., Sanders, P.: Transit node routing reconsidered. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 55–66. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38527-8_7
2. Cui, W., Xiao, Y., Wang, H., Wang, W.: Local search of communities in large graphs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 991–1002 (2014)
3. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 405–418 (2008)
4. Klein, P.N., Mozes, S., Weimann, O.: Shortest paths in directed planar graphs with negative lengths: a linear-space o (n log2 n)-time algorithm. ACM Trans. Algorith. (TALG) **6**(2), 1–18 (2010)
5. Li, L., Zhang, F., Zhang, Z., Li, P., Bu, C.: Multi-fuzzy-objective graph pattern matching in big graph environments with reliability, trust and social relationship. World Wide Web **23**(1), 649–669 (2020)
6. Li, L., et al.: Bindex: a two-layered index for fast and robust scans. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 909–923 (2020)
7. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speedup Dijkstra's algorithm. J. Exp. Algorith. (JEA) **11**, 2–8 (2007)
8. Nannicini, G., Baptiste, P., Barbier, G., Krob, D., Liberti, L.: Fast paths in large-scale dynamic road networks. Comput. Optim. Appl. **45**(1), 143–158 (2010)
9. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, pp. 867–876 (2009)
10. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. J. Exp. Algorith. (JEA) **5**, 12-es (2000)
11. Seo, J., Guo, S., Lam, M.S.: Socialite: an efficient graph query language based on datalog. IEEE Trans. Knowl. Data Eng. **27**(7), 1824–1837 (2015)
12. Sommer, C.: Shortest-path queries in static networks. ACM Comput. Surv. (CSUR) **46**(4), 1–31 (2014)
13. Tesfaye, B., Augsten, N., Pawlik, M., Böhlen, M.H., Jensen, C.S.: An efficient index for reachability queries in public transport networks. In: Darmont, J., Novikov, B., Wrembel, R. (eds.) ADBIS 2020. LNCS, vol. 12245, pp. 34–48. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54832-2_5

14. Yuan, L., Qin, L., Zhang, W., Chang, L., Yang, J.: Index-based densest clique percolation community search in networks. IEEE Trans. Knowl. Data Eng. **30**(5), 922–935 (2017)
15. Zhuge, H., Liu, J., Feng, L., Sun, X., He, C.: Query routing in a peer-to-peer semantic link network. Comput. Intell. **21**(2), 197–216 (2005)