Top-k Keyword Search Over Graphs Based On Backward Search

Jia-Hui Zeng, Jiu-Ming Huang, Shu-Qiang Yang

1College of Computer National University of Defense Technology, Changsha, China
2College of Computer National University of Defense Technology, Changsha, China
3College of Computer National University of Defense Technology, Changsha, China
jhzeng93@foxmail.com

Abstract: Keyword search is one of the most friendly and intuitive information retrieval methods. Using the keyword search to get the connected subgraph has a lot of application in the graph-based cognitive computation, and it is a basic technology. This paper focuses on the top-k keyword searching over graphs. We implemented a keyword search algorithm which applies the backward search idea. The algorithm locates the keyword vertices firstly, and then applies backward search to find rooted trees that contain query keywords. The experiment shows that query time is affected by the iteration number of the algorithm.

1. Introduction

Graphs are applied in many areas. For example, RDF (Resource Description Framework) data model is a graph-shaped data model. Keyword search is a useful tool when searching large graph data. It is a user-friendly way to retrieve graphs because it does not require users to know the structure of the graph and the syntax of the query language.

In this paper, we focus on top-k ranked keyword searching on vertex-labelled graphs. Each vertex in the graph can contain multiple keywords. Given a query list with multiple keywords, we want to find out the subgraph that contains all query keywords. There may be multiple subgraphs that meet the requirement in the graph. It is time-consuming to find all such subgraphs. Moreover, users usually want to obtain the most relevant results with the query. So an evaluation function is needed to assess and sort the results. The evaluation function will be discussed in section 2.

For its convenience, keyword search has been adopted in many situations. Keyword search can be used to query XML data, and there is a lot of works focusing on this problem (e.g., [3] [4] [5] [6] [7]).

Keyword search in relational databases also attracts the attention of many researchers. [8] [9] [10] [11] [12] are related works about keyword search in relational databases.

There are many studies about keyword search over graphs. [13] [14] [2] [15] [16] are well known methods about keyword search over graphs. With the development of Semantic Web, a large amount of RDF data is distributed to the Internet. There has been increasing interest in keyword queries over RDF data recently. Because RDF is a kind of graph-shaped data, keyword search can be used to search RDF data. Related work includes [17] [18] [19] [1] [21]. A recent survey about keyword search over XML data, relational databases and graphs can be found in [22].

The rest of the paper is organized as follows. Section 2 defines the problem to be solved in this paper. Section 3 describes the data structure used in the search algorithm. Section 4 describes the detailed implementation of the algorithm. Section 5 discusses the experimental results. Section 6 concludes the paper.

2. Problem Definition

We are concerned with querying the directed graph $G = (V, E)$. Because the undirected graph is a special directed graph, the algorithm in this paper can be applied to undirected graphs. We follow the problem definition in [2]. Each vertex can contain multiple keywords and each keyword can be contained by multiple vertices. We use $W(v)$ to represent all keywords that contained by v. Given a query q with m keywords $\{w_1, w_2, \dots, w_m\}$, $\{r, v_1, v_2, \dots, v_m\}$ is called a candidate answer $A(q)$ if it meets the requirements below:

For each w_i, $w_i \in W(v_i)$.

For each v_i, a path can be found in G from r to v_i.

Each element in $A(q)$ is a vertex and r is called the root of $A(q)$. There may be a lot of candidate answers for the query q in G. So the definition above can be extended to the top-k version. The quality of a candidate answer is measured by the evaluation function. Equation (1) is the evaluation function used in this paper.

$$Score\big(A(q)\big) = \sum_{i=1}^{m} d(r, v_i) \quad (1)$$

In Equation (1), $d(r, v_i)$ denotes the shortest path from r to v_i. In our definition, the smaller the score, the higher the quality of the answer. When all answers for q in G are sorted in ascending order based on the scores, we use $A(q, i)$ to represent the ith candidate answer. Given a query q, the goal of top-k keyword search is to find out $\{A(q,1), A(q,2), \ldots, A(q,k)\}$. Moreover, the root of each $A(q, i)$ is not the same.

3. Data Structure Used In Algorithm

The algorithm mentioned in this paper applies the backward search idea. It starts to search simultaneously at all vertices that contain any keyword of q and expand to their neighboring vertices recursively. If a vertex connecting all query keywords of q was found, this vertex would be the root of a candidate answer. The termination condition determines when to stop the iteration process. We first introduce the data structure used in the algorithm.

3.1 Structure Table

The structure table is used to store the adjacent information of the graph. Assume that $u, v \in G$ and there is an edge from u to v in G. Then we call u is the upper vertex of v. We use $Upper(v)$ to indicate the collection of all upper vertices of v. The structure table is a two-column table in the database. The first column is used to store vertices in G. And the second column stores $Upper(v)$ of corresponding v in the first column. The data type of the second column is array. Taking Figure 1 as an example, the structure table of Figure 1 is shown in Table 1. Because v_5 has no upper vertex, there is no corresponding row in Table 1.

3.2 Inverted Index Table

A keyword can be contained by multiple vertices. The inverted index table is used to record the distribution of keywords. We use W_i to denote the collection of vertices containing the keyword w_i. The inverted index table is a two-column table in the database. The first column is used to store keywords. And the second column is used to store vertices that contain the corresponding keyword. For example, the inverted index table of Figure 1 is shown in Table 2.

3.3 Iteration List

The algorithm is based on the BFS (Breadth-First Search). A lot of vertices will be visited in the process of search. The iteration list is used to record vertices visited in each step. It is dynamically changing with the iteration process. In each element of iteration list, we store a list of m sets. We use $IL[i][j]$ to indicate the jth set in ith element of iteration list. The content of $IL[i+1][j]$ is shown in Equation (2).

$$IL[i+1][j] = \bigcup_{v \in IL[i][j]} Upper(v) - \bigcup_{p=0}^{i} IL[p][j] \quad (2)$$

In Equation (2), the range of values for i is $[0, +\infty)$ and for j is $[1, m]$. $IL[i][j]$ indicates the collection of vertices that can reach any vertex containing w_j in i hops. So $IL(0, j)$ is the same as W_j. For any $v \in w_j$, i is the length of shortest path from v to w_j, or more precisely, the length of shortest path from v to any vertex in W_j. Therefore, if $v \in IL[i][j]$, v would not be contained in $IL[i+\alpha][j]$ for $\alpha \in N^*$.

3.4 Shortest Path Map

In the process of search, the length of shortest path between two vertices will be recorded. We maintain a set of elements in a map and one for distinct vertex which has been visited in the process of search. We use $SP[v]$ to indicate the entry for v in the map. An array is stored in $SP[v]$. The length of the array is $m + 2$. We use $SP[v][j]$ to denote the j th element in $SP[v]$. $SP[v][j+2]$ represents the length of shortest path from v to w_j for $j \in [1, m]$. $SP[v][0]$ indicates the sum of known shortest path length of v and $SP[v][1]$ denotes the number of keywords in q that have not been reached by v so far.

4. Algorithm Details

Given a query q, the purpose of the algorithm is to find out k top ranked candidate answers. The algorithm consists of two parts. One is used to find roots of candidate answers and the other is used to build paths from the root to query keywords. The algorithm detail of finding roots appears in Algorithm 1.

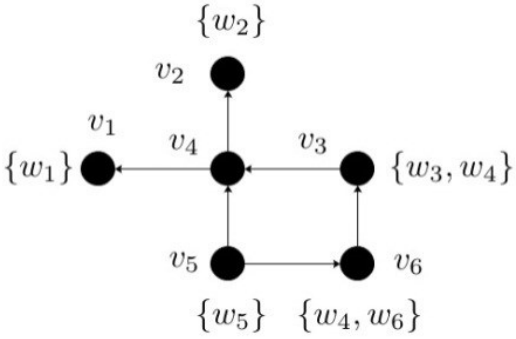

Figure 1. A sample graph

Table1. The structure table of the sample graph

Vertex	Upper Vertices
v_1	$\{v_4\}$
v_2	$\{v_4\}$
v_3	$\{v_6\}$
v_4	$\{v_3, v_5\}$
v_6	$\{v_5\}$

Table2. The inverted index table of the sample graph

Keyword	Vertices
w_1	$\{v_1\}$
w_2	$\{v_2\}$
w_3	$\{v_3\}$
w_4	$\{v_3, v_6\}$
w_5	$\{v_5\}$
w_6	$\{v_6\}$

Algorithm 1 Root Finding

Input: a query with m keywords $\{w_1, w_2, \dots, w_m\}$
Output: roots and scores of top-k answers

1: Initialize IL and SP to empty sets;
2: Initialize $WV[i]$ to an empty set for $i \in [1, m]$;
3: $step \leftarrow 0$;
4: **for** each $i \in [1, m]$ **do**
5: search for W_i from the inverted index;
6: add W_i to $IL[0][i]$;
7: **for** each $v \in W_i$ **do**
8: update $SP[v]$;
9: add v to $WV[i]$;
10: **end for**
11: **end for**
12: **if** $\bigcap_{i=1}^{m} WV[i] \neq \emptyset$ **then**
13: **for** each $v \in \bigcap_{i=1}^{m} WV[i]$ **do**
14: add v and $SP[v][0]$ to AS;
15: remove $SP[v]$ from SP;
16: **end for**
17: **end if**
18: **while** termination condition is not satisfied **do**
19: **for** each $i \in [1, m]$ **do**
20: $temp[i] \leftarrow \emptyset$;
21: **for** each $v \in IL[step][i]$ **do**
22: search for $Upper(v)$ from the database or NH;
23: add $Upper(v)$ to $temp[i]$;
24: **end for**
25: $temp[i] \leftarrow temp[i] - WV[i]$;
26: $WV[i] \leftarrow WV[i] \cup temp[i]$;
27: **for** each $v \in temp[i]$ **do**
28: update $SP[v]$;
29: **end for**
30: **end for**
31: $step \leftarrow step + 1$;
32: add $temp$ to $IL[step]$;
33: **if** $\bigcap_{i=1}^{m} WV[i] - AS \neq \emptyset$ **then**
34: **for** each $v \in \bigcap_{i=1}^{m} WV[i] - AS$ **do**
35: add v and $SP[v][0]$ to AS;
36: remove $SP[v]$ from SP;
37: **end for**
38: **end if**
39: $\tau_{stop} \leftarrow$ the kth smallest score of answers in AS;
40: **end while**
41: **return** AS;

In Algorithm 1, $WV[i]$ is used to store vertices that can reach w_i. AS represents the answer set. There are many entries in AS and each entry is a pair $(root, score)$. $root$ denotes the root of a candidate answer and $score$ is the score of this candidate answer according to Equation (1).

In line 22, NH denotes a data structure which is used to store $Upper(v)$ in memory. The structure of NH is similar to the structure table. All adjacency information of $v \in G$ is stored in the structure table, which is a table in the database. The presence of adjacency information in the database can facilitate the storage of large graphs. Moreover, the indexing mechanism of the database can speed up the retrieval of $Upper(v)$. But in the search process, the operation of finding adjacency information is very frequent. So we set up NH to store part of adjacency information in memory. When searching for $Upper(v)$, we first look from NH. If v existed in the NH, then the algorithm would read $Upper(v)$ from NH. If v did not exist in the NH, then the algorithm would read $Upper(v)$ from the structure table and put v and $Upper(v)$ in NH. In this way, it is not necessary to access the database repeatedly when searching for $Upper(v)$ of a same v.

$SP[v]$ records the length of shortest paths from v to query keywords. If the shortest path from v to w_j was found, then the length of this path would be stored in $SP[v][j+2]$. If not, $SP[v][j+2]$ would remain null.

The meaning of Algorithm 1 is as follows. Firstly, initialize the variables used in the algorithm. Then search for W_i from the inverted index in the database. Next put W_i in $IL[0][i]$. For each vertex v in W_i, update the corresponding values of $SP[v]$. For example, if $SP[v]$ existed and $SP[v][i]$ was null, then we would assign $step$ to $SP[v][i]$. If $SP[v]$ did not exist, then we would create an entry $SP[v]$ and assign $step$ to $SP[v][i]$. If $SP[v]$ existed and $SP[v][i]$ was not null, then we would not make any changes. Afterwards, update $SP[v][0]$ and $SP[v][1]$. Next, put v in $WV[i]$. That means v has been visited. Then check if the intersection of $WV[i]$ is empty. If not, that means there are vertices containing all query keywords. Then put these vertices with corresponding scores in AS and remove them from SP. Next, determine whether the termination condition is satisfied. If not, start the iteration process. For each $v_i \in IL[step][i]$, search for $Upper(v_i)$ from the database or NH and put $Upper(v_i)$ in $temp[i]$. In that way, what $temp[i]$ stores is upper vertices of $v \in IL[step][i]$. Remove the vertices that have already been visited from $temp[i]$ and put the remaining vertices in $WV[i]$. Next, update the $SP[v]$ for $v \in temp[i]$. Afterwards, add 1 to $step$ and put $temp$ in $IL[step]$. Next determine whether new roots of candidate answers are found. When new roots are found, put these vertices with corresponding scores in AS and remove them from SP. After that, choose the k th smallest score of answers in AS as threshold τ_{stop}. Then continue determining the termination condition. Lastly, return AS in which the first k vertices are roots of $\{A(q,1), A(q,2), \dots, A(q,k)\}$.

We used two termination conditions. The first termination condition is $\tau_{stop} \geq step$. When $\tau_{stop} \geq step$, it is impossible to find a better answer. For a possible candidate answer rooted at v that has not been put in the answer set, there is at least one keyword that has not been reached by v yet. So the length of shortest

path from v to w must be greater than $step$. That means the score of this possible candidate answer must be greater than the kth smallest score of answers in the answer set. So it is impossible to find a better answer when this condition is satisfied. The second termination condition is $\tau_{stop} \geq minScore(SP)$. $minScore(SP)$ is used to calculate the lower bound of each possible answer's score which is rooted at v, and select the smallest lower bound. The calculation method is shown in Equation (3).

$$B_l(v) = \sum_{j=1}^{m} f(v,j)SP[v][j+2] + \big(1 - f(v,j)\big)(step+1) \quad (3)$$

If v can reach w_j, then $f(v,j)$ equals 1. Otherwise, $f(v,j)$ equals 0. If the kth smallest score of candidate answers in AS is no greater than $minScore(SP)$, then the iteration process can be terminated because the candidate answer with smaller score cannot be found any more. Assume that there is a candidate answer $A(q)$ rooted at v with smaller score when $minScore(SP)$ is less than the kth smallest score. In that case, the root of $A(q)$ does not exist in the AS. That means that there is at least one keyword w_j that has not been reached by v yet. Then the length from v to w_j is at least $step + 1$. So it is impossible for $A(q)$ to be a top-k answer. This termination condition can help find the top-k answers for the query.

After getting k roots, paths from each root to w_j can be constructed by searching IL, SP and the structure table. The detailed description of the algorithm is shown in Algorithm 2.

Algorithm 2 returns the path from $root$ to w_j. Assume that $u, v \in G$ and there is an edge from v to u in G. We call u is the lower vertex of v. We use $Lower(v)$ to denote all lower vertices of v. $queue_i$ is a queue in which vertices can be reached by $root$ in i hops. $queue_i.removeAll$ means to clear all elements in $queue_i$. There may be multiple shortest paths, and the Algorithm 2 only returns one of them.

The top-k answers for keyword search over the graph can be found by combining Algorithm 1 and Algorithm 2.

Algorithm 2 Path Building

Input: $root$, w_j
Output: a path from $root$ to w_j

```
1: i ← 0;
2: length ← SP[root][j + 2];
3: v ← root;
4: path[0] ← v;
5: while i < length do
6:    if v is not null then
7:       if Lower(v) ∩ IL[length − i − 1][j] ≠ ∅ then
8:          queue_{i+1}.removeAll;
9:          queue_{i+1}.add(Lower(v) ∩ IL[length − i − 1][j]);
10:         i ← i + 1;
11:      end if
12:   else
13:      i ← i − 1;
14:   end if
15:   v ← pop(queue_i);
16:   path[i] ← v;
17: end while
18: return path;
```

5. Experimental Results

The vertex-labelled graph used in experiments is randomly generated. It includes 1000000 vertices and 1000000 keywords. We use different integers to represent different keywords. The DBMS used in the experiment is PostgreSQL.

We carried out Algorithm 1 under different termination conditions on a randomly generated graph with 1000000 vertices. We randomly select some integers as query keywords. Table 3 lists eight queries. Figure 2 shows the time Algorithm 1 takes to find roots of top 2 answers under two termination conditions and Figure 3 shows the number of iterations in different conditions. The unit of time is milliseconds.

From Figure 2, we can see that the query time under Termination 2 is shorter than that of Termination 1. Moreover, when the number of query keywords increases, the difference between the query time of two conditions also increases. From the Figure 3, we can see that the number of iterations under Termination 2 is no greater than the number of iterations under Termination 1. That means Termination 2 can terminate the Algorithm 1 in fewer steps. By comparing two figures, we can see that the query time decreases as the number of iterations decreases. In query Q3, the iteration number of Termination 1 is the same as that of Termination 2 and the query time is not much different. And in other queries, we can see that as the difference between iteration numbers increases, the difference between query times also increases. So Termination 2 can terminate the Algorithm 1 more quickly than Termination 1 because the iteration number under Termination 2 is less than that of Termination 1. Moreover, the number of query keywords also has an impact on query time. The iteration number of Q4 under Termination 1 is the same as the iteration number of Q7

under Termination 2. But the query time of Q4 under Termination 1 is shorter than the query time of Q7 under Termination 2. So the number of query keywords can also have an impact on the query time. The more query keywords, the longer the query time.

In a word, Termination 2 can terminate the algorithm more quickly than Termination 1 because Termination 2 has less number of iterations than Termination 1. Moreover, the number of keywords is positively correlated with the query time.

Table3. Queries

Queries	Keyword vertices
Q_1	(1,2)
Q_2	(10,26598)
Q_3	(11,3299)
Q_4	(666,888)
Q_5	(3976,644)
Q_6	(987,25267)
Q_7	(550560,402060,200442)
Q_8	(12013,172248,281573)

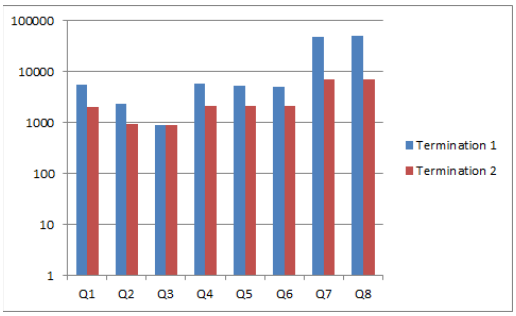

Figure 2. Query time under different termination conditions

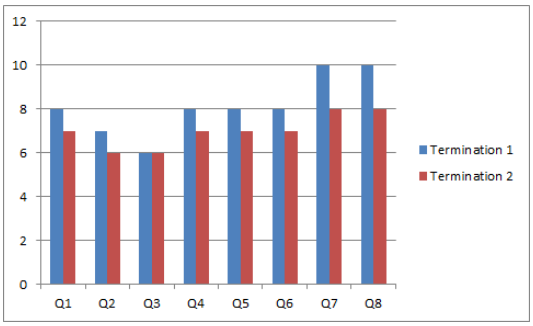

Figure 3. Comparing the number of iterations

6. Conclusion

In this paper, we implemented a method that addresses the issue of top-k keyword search over graphs. For each query keyword w_i, we first find vertices which contain w_i. Then using BFS to extend each keyword vertex. If a vertex that can reach all query keywords was found, this vertex would be the root of a candidate answer. The algorithm will terminate when the termination condition is satisfied. When the algorithm stops, it will return top-k answers and each answer is a rooted tree. We compared the query time and iteration numbers when using different termination conditions. And the experiment shows that the second termination is better than the first one, because it has fewer numbers of iterations than the first termination condition.

Acknowledgment

The work described in this paper is supported by National Basic Research Program of China (973 Program) granted No.2013CB329601, The National Key Research and Development Program of China (2016QY03D0601, 2016QY03D0603) and National Natural Science Foundation of China (No.61502517, No.61672020, No.61662069).

References

[1] W. Le, F. Li, A. Kementsietsidis, and S. Duan, Scalable Keyword Search on Large RDF Data, IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 11, pp. 2774–2788, 2014.

[2] H. He, H. Wang, J. Yang, and P. S. Yu, BLINKS: Ranked Keyword Searches on Graphs, ACM SIGMOD International Conference on Management of Data, pp. 305-316, 2007.

[3] D. Florescu, D. Kossmann, and I. Manolescu, Integrating Keyword Search into XML Query Processing, international world wide web conferences, vol. 33, no. 1, pp. 119–135, 2000.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, XSEarch: A Semantic Search Engine for XML, Very Large Data Bases, pp. 45-56, 2003.

[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, XRANK: Ranked Keyword Search over XML Documents, ACM SIGMOD International Conference on Management of Data ACM, pp. 16-27, 2003.

[6] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, On the Integration of Structure Indexes and Inverted Lists, pp. 779–790, 2004.

[7] Y. Li, C. Yu, and H. V. Jagadish, Schema-Free XQuery, Very Large Data Bases, pp. 72–83, 2004.

[8] S. Agrawal, S. Chaudhuri, and G. Das, DBXplorer: A System for Keyword-Based Search over Relational Databases, International Conference on Data Engineering, 2002.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, Keyword Searching and Browsing in Databases using BANKS, pp. 431–440, 2002.

[10] V. Hristidis, and Y. Papakonstantinou, DISCOVER: Keyword Search in Relational Databases, Very Large Data Bases, pp. 670-681, 2002.

[11] V. Hristidis, L. Gravano, and Y. Papakonstantinou, Efficient IR-Style Keyword Search over Relational Databases, Very Large Data Bases, pp. 850–861, 2003.

[12] F. Liu, C. Yu, W. Meng, and A. Chowdhury, Effective Keyword Search in Relational Databases, pp. 563–574, 2006.

[13] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, Bidirectional Expansion For Keyword Search on Graph Databases, Very Large Data Bases, pp. 505–516, 2005.

[14] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, Finding Top-k Min-Cost Connected Trees in Databases, pp. 836–845, 2007.

[15] B. Dalvi, M. Kshirsagar, and S. Sudarshan, Keyword Search on External Memory Data Graphs, Proceedings of The Vldb Endowment, vol. 1, no. 1, pp. 1189–1204, 2008.

[16] J. Shi, D. Wu, and N. Mamoulis, Top-k Relevant Semantic Place Retrieval on Spatial RDF Data, pp. 1977–1990, 2016.

[17] S. Elbassuoni and R. Blanco, Keyword Search over RDF Graphs, pp. 237–242, 2011.

[18] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data, pp. 405–416, 2009.

[19] X. Lian, E. De Hoyos, A. Chebotko, B. Fu, and C. F. Reilly, k-nearest keyword search in RDF graphs, Journal of Web Semantics, vol. 22, no. 0, pp. 40–56, 2013.

[20] C. Halaschek, B. Alemanmeza, I. B. Arpinar, and A. P. Sheth, Discovering and Ranking Semantic Associations over a Large RDF metabase, Very Large Data Bases, pp. 1317–1320, 2004.

[21] H. Fu and K. Anyanwu, Effectively Interpreting Keyword Queries on RDF Databases with a Rear View, pp. 193–208, 2011.

[22] H. Wang, and C. C. Aggarwal, A Survey of Algorithms for Keyword Search on Graph Data, Managing and Mining Graph Data. Springer US, pp. 249-273, 2010.