情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-DBS-151 No.28
2010/11/13

A report on Top-k keyword search algorithms with redundancy elimination on graph databases

Meirong Wang[†] Liru Zhang[†] and Tadashi Ohmori[†]

In recent database studies, a new trend for managing vast amounts of heterogeneous data is to represent all data as a graph database and to provide a ranked keyword search algorithm in that graph. Many previous studies [1][2][3][4][5][6] discussed new searching algorithms for finding better sub-graph in a graph database, in such a way to detect appropriate sub-graph in a ranked order under given keywords. In this paper, by modifying an existing graph-database search algorithm DPBF [1] in a straightforward strategy, we propose such an algorithm that can find exact top-k with eliminating redundant answers, in an exact ranked order in $O((2^{l+1}mk + 3^{l}nk \cdot \log k)\log(2^{l}nk))$ run time (l: the number of keywords; k: top-k; n: the number of nodes in the graph database; m: the number of edges in the graph database) with the space complexity of $O(2^{l+1}nk)$.

1. Introduction

Recently, keyword search has been a very popular way to explore information. And many studies supporting keyword search on structured databases have been proposed like DBXplore [5], DISCOVER [6], BANKS-I/II [2][3], DPBF [1]. For example, in case of relational databases, consider the tuples of relations as nodes and the foreign-key relationships between relations as edges; then the whole relational database is modeled as a graph. When a set P of keywords are given as a query, how to find an appropriate sub-graph of much smaller cost which satisfies P in that graph is the problem in this field [1][2][3][7][8].

Let us explain this problem by a simple example. Suppose that there is a graph database (Figure 1) to present a simple bibliography database. In Figure 1 the nodes t1, t2, t3, t4, t5, t6, t7 present seven papers. Author al wrote the papers {t2, t4} and author a2 wrote the papers {t3, t4, t5, t6, t7}. The nodes c1 and c2 describe that t2 is cited by t1 and t3. The node c3 means that t4 is cited by t5. And the node c4 means that t7 is cited by t6. We set the weight of each edge to be 1 to simplify the introduction. Let this weighted graph database be noted by G(V(G), E(G)), where V(G) is the set of tuples (nodes) and E(G) is the set of edges.

Assume that we are given a set of keywords P={p1, p2, p3 ,p4}, where p1, p2, p3 appear in the paper titles and p4 presents an author. Suppose the nodes t1, t2, t3, t5, a1 contain keywords as shown in Figure 1. These nodes are called leaf nodes. Then one definition of the search problem is to find top-k min-cost connected trees including at least one leaf node for each keyword as the top-k answers in this graph. Namely, this is to find the top-k answers which contain all the keywords in the ranked order of the increasing costs of connected trees. This definition is adopted in DPBF [1] and is known as finding the top-k min-cost connected tree based on the algorithm of GST-k (Group Steiner Tree).

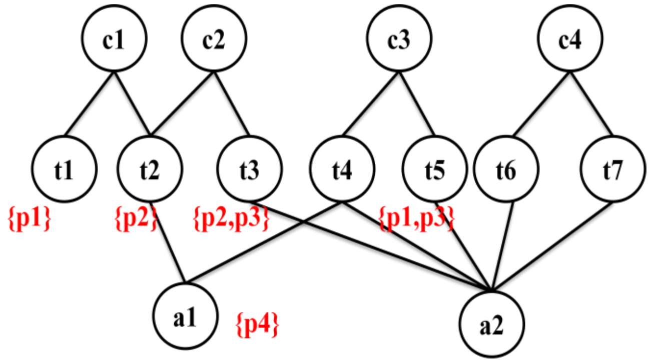

Figure 1 A simple graph database

Historically there are two approaches in the above search problem. One way is to find the shortest paths from the given leaf nodes to a special root node. Typical algorithms are BANKS-I/II [2][3], and BLINKS [4], but their answers are approximate when we regard a group stenier tree as an exact answer. The other way as a solution of Figure 1 is to grow and merge trees which contain leaf nodes until top-k min-cost stenier trees containing all the keywords are found. DPBF [1] is a typical algorithm in this category, and can enumerate qualifying group stenier trees in an ascending order of the weight of each tree.

DPBF grows and merges all the trees at different root nodes, so it can get the accurate top-1.

[†] Graduate School of Information Systems, The University of Electro-Communications

It just keeps minimum one of the trees with the same root node and the same keywords, so the answers from top-2 to top-k cannot be exact. Furthermore DPBF for finding top-k answers must produce some redundant answers which are not explained in [1]. Recently the studies of [7][8] proposed exact/approximate efficient enumeration algorithms with eliminating redundant answers under the definition of finding top-k minimum stenier trees, however the proposed algorithms for the exact enumeration are still complicated to be implemented. An efficient algorithm for finding exact top-k answers which can avoid redundancy and is easy to be implemented is still required.

Consequently, in this paper, we modify DPBF by a straightforward strategy so that we can find exact top-k answers, with eliminating redundant answers, in an exact ranked order efficiently. To do so, we firstly define the redundant answers of DPBF-k and describe how to eliminate them. Thereafter, in order to get the top-k answers exactly, we propose an improved method named MDPBF, which can find exact top-k answers, with eliminating redundant answers, in exact ranked order.

The remaining of this paper is organized as follows. Section 2 introduces the main idea and redundancy of DPBF [1]. Section 3 represents the improved algorithm MDPBF. In section 4, we will show our experimental results. Section 5 summarizes this paper.

2. DPBF

DPBF [1] is the method which aimed at finding the top-k min-cost connected trees using the keyword queries based on graph database. The min-cost connected tree contains all the keywords, and the connected tree with smaller cost presents more important information. Subsection 2.1 briefly describes the DPBF according to [1]. Next in subsection 2.2 we will describe the redundancy of DPBF-k. The elimination method will be introduced in subsection 3.3.

2.1 The main idea of DPBF

DPBF [1] is based on the following recursive formulas to find the optimal connected tree. Suppose that P is the set of keywords and $p \subseteq P$ is a subset of the keywords. $T(v,p)$ means a min-cost tree rooted at node v which contains a keyword set p.
If node v directly contains a keyword subset p:

$$T(v,p) = 0 \tag{1}$$

If a tree has more than one node:

$$T(v,p) = min\{\, T_g(v,p) \cup T_m(v,p)\,\} \tag{2}$$

where

$$T_g(v,p) = \{\, (v,u) \oplus T(u,p) | u \in N(v)\,\} \tag{3}$$

$$T_m(v,p) = \{\, T(v,p_1) \oplus T(v,p_2) | \,(p_1 \cap p_2 = \emptyset) \wedge (p_1 \cup p_2 = p)\} \tag{4}$$

As notations:

- $\oplus$ is an operation to merge two trees into a new tree.
- $N(v)$ is a set of neighbor nodes of node v such as $N(v) = \{\, u | (v,u) \in E(G)\,\}$.

In Eq.(1), $T(v,p)$ is a single leaf node tree without any edge, so the cost is 0.

In Eq.(2), do grow/merge operations to get new trees and only keep the min-cost one of the trees having the same root node and the same keywords.

Eq.(3) and Eq(4) express the tree grow and tree merge operation respectively.

Figure 2 shows the tree grow and tree merge clearly.

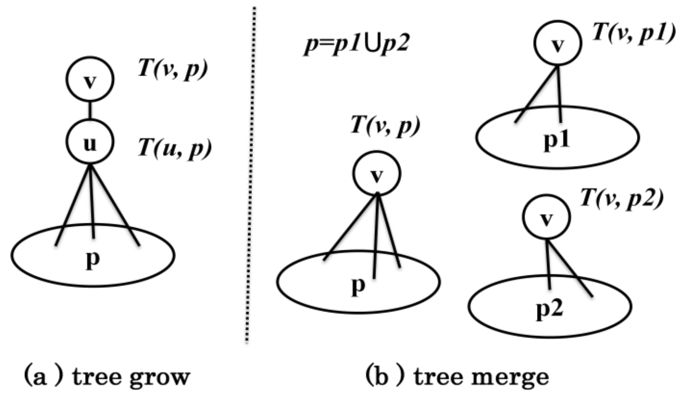

Figure 2 Tree Grow & Merge

However DPBF [1] just keeps the one with min-cost of the trees having the same root node and the same keywords, so it is possible to drop many trees which can form GST-n $(1 < n \leq k)$ potentially. So the top-k connected trees got by DPBF-k are approximate.

2.2 Redundancy of DPBF-k

An answer should be a non-redundant subtree that contains all the keywords [7] so it is important to eliminate redundant answers. In [1] there is not clear explaining about the redundancy elimination. In this subsection, we will introduce three kinds of redundant answers of DPBF-k.

In this paper, we propose three cases of redundant answers as shown in Figure 3.

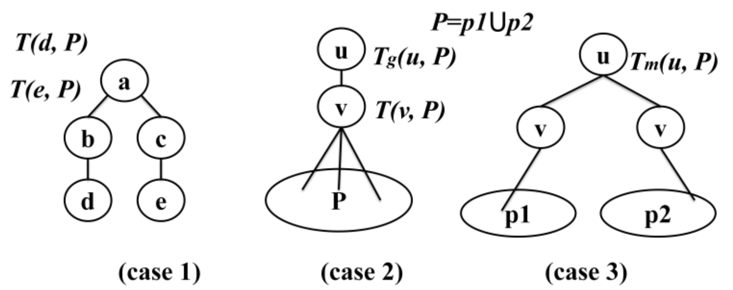

Figure 3 Three cases of redundant answers

Case 1: the trees which have the same nodes and edges but with different root nodes. We call these trees *redundant answers of isomorphs*.

An example (Figure 3 (1)): When the trees begin from different root nodes d and e, finally we get two trees $T(e,P)$ and $T(d,P)$. They contain the same information. But DPBF considers that they are different.

Case 2: the trees which are the super sets of the trees containing all keywords.

An example (Figure 3 (2)): Tree $T(v,P)$ has contained all the keywords. After doing a grow operation on it, we get the new tree $T_g(u,P)$ which contains the same information as $T(v,P)$ but has a bigger cost. In [1], we cannot judge whether DPBF-k continues grow/merge from a tree which has contained all the keywords clearly. So we assume this kind of redundant answers happens, and call them *redundant answers of growing*.

Case 3: the trees which count the cost of an edge more than once.

See an example in Figure 3 (3). When doing grow operations to the trees $T_1(v,p_1)$ and $T_2(v,p_2)$, we get two new trees $T^{'}(u,p_1)$ and $T^{''}(u,p_2)$. Do merge operation on $T^{'}$ and $T^{''}$ to get the tree $T_m(u,p_1\cup p_2)$. In fact T_m counts the cost of edge (v,u) twice. Consider a given keywords query $P=p_1\cup p_2$. Then $T(v,p_1\cup p_2)$ is already an answer, and so $T_m(u,p_1\cup p_2)$ is redundant. Because merge operation is done after grow operation, so this case of redundant answers is easy to happen. We call them *redundant answers of merging*.

All the elimination methods will be introduced in the Section 3 about the redundancy elimination of MDPBF.

3. MDPBF

In order to find the exact top-k answers and eliminate redundant answers, we propose an improved method named MDPBF.

3.1 The main idea of MDPBF

From the Eq(1)~(4) of DPBF, we can see that it just keeps minimum one with the same root node and the same keywords, and that is why it loses answers. In order to find the top-k exactly, a simple strategy is to keep the top-k connected trees with the same root node and the same keywords. Firstly, we improve the Eq(1)~(4) of DPBF [1].

As notations:

- $T_n(v,p)$ means the n-th smallest connected tree with root node v and keyword subset p where $(1\le n\le k)$ got by function min_n.
- The function min_n returns the connected tree with the n-th smallest cost from a set of connected trees having the same root node and the same keywords.

The modified equations are shown as follows.

If node v directly contains a keyword subset p:

$$T_1(v,p)=v \tag{5}$$

If a tree has more than one node:

$$T_n(v,p)=min\_n\{\,TG(v,p)\cup TM(v,p)\,\} \tag{6}$$

where

$$TG(v,p)=\{\,(v,u)\oplus T_i(u,p)|u\in N(v),1\le i\le k\,\} \tag{7}$$

$$TM(v,p)=\{\,T_i(v,p_1)\oplus T_j(v,p_2)|\,(p_1\cap p_2=\emptyset)\wedge(p_1\cup p_2=p,i\times j\le k)\,\} \tag{8}$$

Eq(5) is the same as the Eq(1). Eq(6), (7) and (8) are the expansions of Eq(2), (3) and (4) respectively. Instead of keeping the only minimum one, we store more trees got through grow/merge operations.

Because we aim at finding top-k, we just store top-k of the trees having the same root node and the same keywords. When we do the merge operation, there are at least $i\times j$ combinations, for reducing the size of queue, we just do merge where $i\times j\le k$.

The algorithm of MDPBF is shown as follows:

```
Input:
    Database Graph G, a set $P$ of keywords, Top-k
Output:
    GST-k
$Q_G$:
    Priority queue sorted in the increasing order of costs of connected trees
$Q_L(v,p)=\{T_1(v,p),\dots,T_k(v,p)\}$:
    Priority queue sorted in the increasing order of costs of the trees with root node $v$ and
keyword set $p$.
```

```
1.  Begin
2.    $Q_G \leftarrow \emptyset$;
3.    For each $v \in V(G), p \subseteq P$ do $Q_L(v,p) \leftarrow \emptyset$;
4.    For each $v \in V(G), p \subseteq P$ do
5.       if v contains keyword then
6.         $T_1(v,p) \leftarrow v$;
7.         enqueue $T_1(v,p)$ into $Q_G$ and $Q_L(v,p)$;
8.    num ← 0;
9.    while $Q_G \neq \emptyset$ do
10.     dequeue $Q_G$ to $T_i(v,p)$ ;
11.      if $p = P$ then
12.        output $T_i(v,p)$;
13.        num ← num + 1;
14.        terminate if num = k ;
15.      for each $u \in N(v)$ do
16.        $T(u,p) \leftarrow T_i(v,p) \oplus (v,u)$;
17.        if the size of $Q_L(u,p)$ is smaller than k then
18.          enqueue $T(u,p)$ into $Q_G$ and $Q_L(u,p)$;
19.        else if $s(T(u,p)) < s(T_k(u,p))$ then
20.          $T_k(u,p) \leftarrow T(u,p)$;
21.          update $Q_G$ and $Q_L(u,p)$ with the new $T_k(u,p)$;
22.      $p1 \leftarrow p$;
23.      for each $p2$ s.t. $p1 \cap p2 = \emptyset$ do
24.        for all $j$ s.t. $1 \leq j \leq k/i$ do
25.          $T(v, p1 \cup p2) \leftarrow T_i(v,p1) \oplus T_j(v,p2)$;
26.          if the size of $Q_L(v, p1 \cup p2)$ is smaller than k then
27.            enqueue $T(v, p1 \cup p2)$ into $Q_G$
                 and $Q_L(v, p1 \cup p2)$;
28.          else if $s(T(v, p1 \cup p2)) < s(T_k(v, p1 \cup p2))$ then
29.            $T_k(v, p1 \cup p2) \leftarrow T(v, p1 \cup p2)$;
30.            update $Q_G$ and $Q_L(v, p1 \cup p2)$
                 with the new $T_k(v, p1 \cup p2)$;
31. End
```

Line 2-3: initialize the priority queues Q_G (global queue) and Q_L (local queue) to be empty

Line 4-7: enqueue the single leaf node trees into Q_G and Q_L.

Line 9-30: while Q_G is not empty, the algorithm repeats to enqueue/dequeue to make all trees grow/merge to reach GST-k.

Line 15-21: tree grow (Eq.(7))

Line 22-30: tree merge (Eq.(8))

As a notation: dequeue Q_G to tree T which is with the smallest cost in all trees in Q_T.

3.2 Redundancy of MDPBF

As we introduced in subsection 2.2, DPBF-k produces redundant answers probably. So MDPBF is also required to avoid the 3 cases of redundant answers of DPBF-k shown in Figure 3.

However MDBF produces a new kind of redundant answers (case 4). Note that MDPBF allows keeping k of the trees which are rooted at the same node and have the same keyword set. Thus it is possible to keep the top-k trees with the same root node and the same leaf nodes but different paths. This property can cause a cycle in $T_n(v,p)$. See an example shown in Figure 4. Suppose that $T_1(u,\{p1,p2,p3\})$ grows to $T_1'(x,\{p1,p2,p3\})$ $(= T_1 \oplus (u,x))$ and further T_1' grows (along the dotted path from x to v in Figure.4) to $T_3(v,\{p1, p2, p3\})$. If T_3 further grows to $T_3'(v,\{p1, p2, p3\})$ $(= T_3 \oplus (v,x))$, T_3' has a cycle.

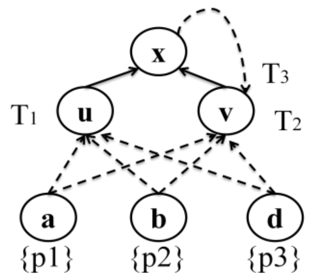

Figure 4 New case of redundant answer (case 4)

3.3 Redundancy elimination

There are four cases of redundant answers in MDPBF, next we introduce the elimination methods.

Firstly, we eliminate the case 1 of redundant answers.

- Method 1 (*isomorphs checking*): Because we aim at finding the top-k connected trees which contain all the keywords in ranked order of increasing costs of trees, there are not more than k answers. In order to avoid the case 1, we just need to check the answers found whether they are the isomorphs.

Second is the case 2 of redundant answers. We propose two alternative methods to avoid case 2:

- Method 2 (*redundancy labeling*): When a tree has contained all the keywords, the new grow/ merge trees from it will be redundant answers, so we can give this kind of redundant answers a label to record that they are redundant, and are not allowed to be result..
- Method 3 (*grow stopping*): Because the merge operation is done when two trees do not have the same keywords, the case 2 of redundant answers just happens on grow operation. In order to avoid it, when a tree has contained all the keywords, we dequeue it directly without doing the grow operation from it.

Next is the case 3 of redundant answers. DPBF just keeps the min-cost one of the trees having the same root node and the same keywords, so this case of redundancy can be avoided automatically in DPBF. MDPBF keeps top-k of the trees having the same root node and the same keywords, so this case of redundant answers can be produced by MDPBF.

- Method 4 (*edges checking*): In case 3 $T_m(u,P)$ counts the cost of edge (v,u) twice, hence its cost is bigger than $T_g(u,P)(=T(v,p_1 \cup p_2) \oplus (v,u))$ which counts the cost of edge (v,u) only once $(T(v,p_1 \cup p_2) = T_1(v,p_1) \oplus T_2(v,p_2))$. In order to avoid this kind of redundant answer, when a new edge is added into a tree, we can check whether this edge has been contained, if so, give up this merge operation.

Final is the case 4 which contains a cycle.

- Method 5 (*nodes checking*): In order to avoid the case 4, one way is to remember all the nodes of a tree directly, then when a new node is added into a tree, we firstly check whether this new node has been contained, if so, we can give up this grow operation.

Except the above the methods, we also proposed a method named leaf nodes checking to avoid case 3 and 4 at the same time.

- Method 6 (*leaf node checking*): in case 3 and case 4, MDPBF may produce some answers which contain useless edges or nodes, so in order to avoid them, we can check all the trees with the same root node and the same leaf nodes. As a side effect, for each root node *v* with any keyword subset *p*, those trees remembered by MDPBF can have the same root node *v* but they must have different sets of leaf nodes that satisfy *p*. As an example, in Figure 4, when $T_2(v,\{p1,p2,p3\})$ grows to $T_2'(x,\{p1,p2,p3\})(= T_2 \oplus (v,x))$, MDPBF remembers a smaller one of T_1' and T_2', but not both.

In summary, we prepare three algorithms with different redundancy elimination referred above.

- DPBF: with the redundancy elimination method 1, 2. Because the DPBF just keeps the min-cost one of the trees with the same root node and the same keywords, so DPBF can avoid the case 3 automatically. Of course, DPBF cannot find exact top-k.

As a practical MDPBF, we propose two versions:

- MDPBF1.1: with the redundancy elimination method 1, 2 and 6. The method 1 is used to avoid the case 1 and the method is used to avoid the case 2. The method 6 can avoid both of case 3 and 4.
- MDPBF1.2: with the redundancy elimination method 1, 3, 4 and 5. We used the same method to avoid the case 1 as MDPBF1.1. We used method 3 to reduce the productions of the redundant candidate trees. And the method 4 is to hold back happens of case 3 and method 5 aimed at stopping the production of case 4.

3.4 Time and space complexities of MDPBF

Times complexity: We use a binary heap to make it easy to evaluate the retrieval efficiency when the worst. Until we find the top-k, we need to repeat grow/merge. Set the size of keyword query be *l*, there are 2^l combinations of keywords. When the number of nodes is *n*, we have to keep $2^l nk$ candidate connected trees. So the maximum size of Q_G is $2^l nk$. And the maximum of loops also equals to $2^l nk$.

The sum of times of do grow to adjacency nodes is $\sum_{v \in V(G)} (|N(v)| k 2^l) = 2^{l+1} mk$.

($m =$ the number of edges of graph database) .

The sum of times to do merge for connected trees with pair wise disjoint keyword sets is

$$O(k \cdot logk) n \sum_{l_i=1}^{l} \binom{l}{l_i} 2^{l-l_i} = O(3^l nk \cdot \log k).$$

Thus the time complexity of MDPBF w.r.t Q_G is $O((2^{l+1} mk + 3^l nk \cdot \log k) \log(2^l nk))$.

Space complexity: MDPBF use Q_G and Q_L to keep all the trees. The maximum number of trees is $2^l nk$ at worst. So the space complexity of MDPBF is basically $O(2^{l+1} nk)$.

4. Experiment results

We did experiments to compare three algorithms DPBF, MDPBF1.1 and MDPBF1.2. All the experiments were performed on a 2.66GHz CPU machine with 3GB memory and implemented by JAVA l_5.

Firstly we used a simple graph database shown in Figure 1 to compare the GST-4 got by MDPBF (MDPBF1.1. and MDPBF1.2 produced the same answers) with the one got by DPBF. The GST-4 got by MDPBF is shown in Figure 5. For the same graph database, DPBF-k can find connected trees of Figure 5 (a), (b) and (d), but cannot get the tree of Figure 5 (c). From this figure, we can see that MDPBF can produce more connected trees which contain all the keywords and have smaller costs.

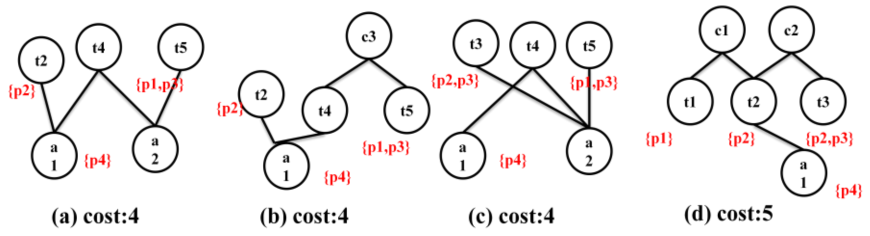

Figure 5 Top-4 answers (All edge-costs are 1)

Next we performed preliminary experiments to compare the answer quality of MDPBF1.1 with MDPBF1.2. Take Figure 6 as an example, give a set of keywords {p2, a1}, and the nodes 5, 8 and 9 are leaf nodes. MDPBF1.1 can produce the trees 1, 2 and 3. The MDPBF1.2 can produce the trees 1, 2, 3 and 4. In MDPBF1.1 the tree 4 is redundant because it contains the same leaf nodes and the same root node with tree 1. Tree 4 has different information with tree 1, but its cost is much higher than tree 1. So we can see that the answer definition of MDPBF1.1 is different from MDPBF1.2. MDPBF1.1 tries to find the combinations of different leaf nodes at different root nodes. And MDPBF1.2 considers that all the trees with different paths are different and all of them can be answers.

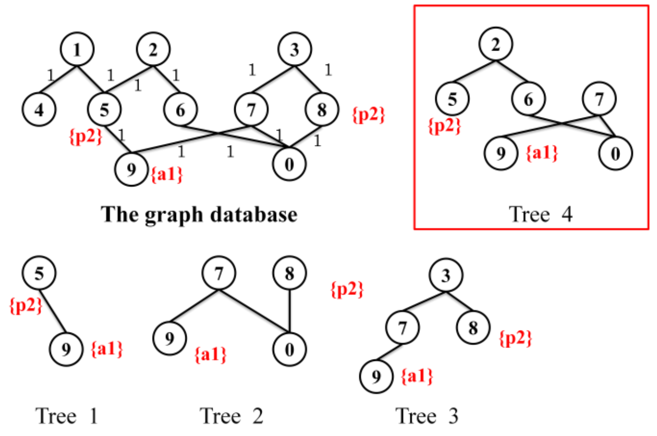

Figure 6 A simple example

Next we performed experiments to compare their performance. The tests are based on the random graph with 1000 nodes and 1500 edges. Every edge-cost is set to 1. The hit rate for each keyword is 1% (1% of all nodes of graph are leaf nodes for each keyword). *k* (of top-k) =10. We changed the number of keywords from 1 to 5. The experimental results are shown in Figure 7, Figure 8 and Figure 9. From the experimental results, we can see that MDPBF1.2 can reduce the size of priority queue and find the top-10 efficiently. Although the quality of answers produced by MDPBF1.1 is high, the performance of MDPBF1.1 is not very well. That is because it produces many candidate trees and the GST-k is found in a wide range. MDPBF1.2 maybe finds the top-k in a narrow range but its efficiency is increased.

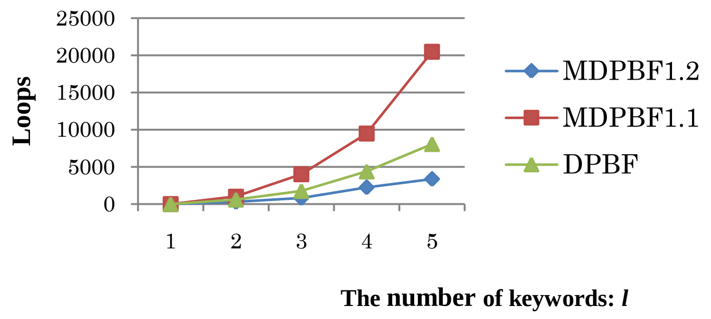

Figure 7 The number of main loop vs. *l* (*k*=10, *n*=1000)

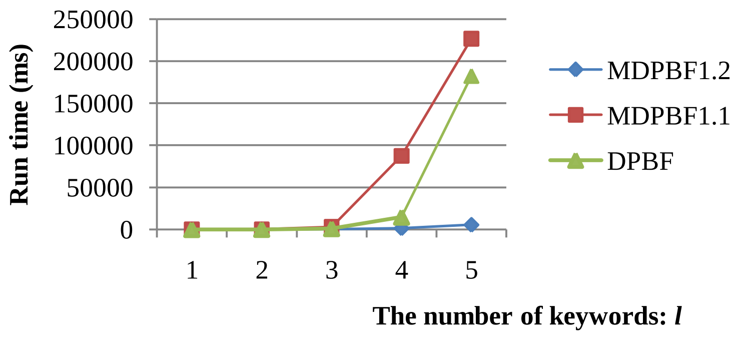

Figure 8 Run time (ms) vs. *l* (*k*=10, *n*=1000)

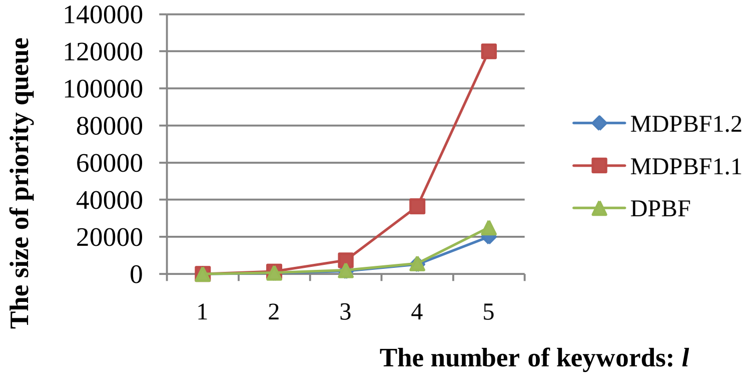

Figure 9　The size of priority queue vs. *l* (*k*=10, *n*=1000)

In order to do nodes checking, MDPBF1.2 need to keep the nodes sets of trees directly, so when the size of nodes set of a tree is very big, MDPBF1.2 takes memory. So we did many experiments to test how much it can work on the graph with 1000 nodes. We added the k (top-k) from 10 to 50. The test results are shown in Figure 10, Figure 11 and Figure 12.

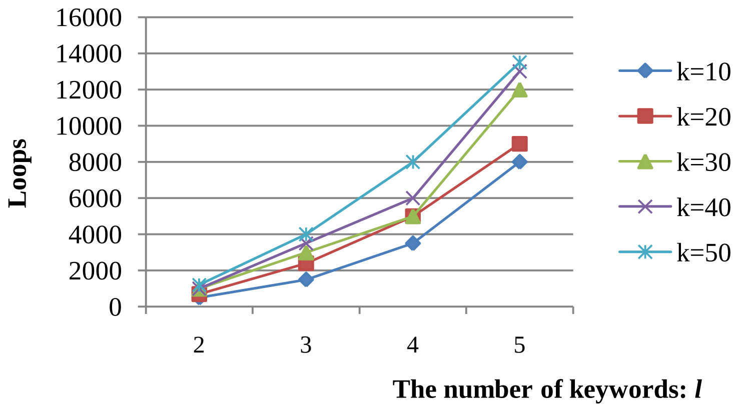

Figure 10　The number of main loop vs. *l* (MDPBF1.2, *n*=1000)

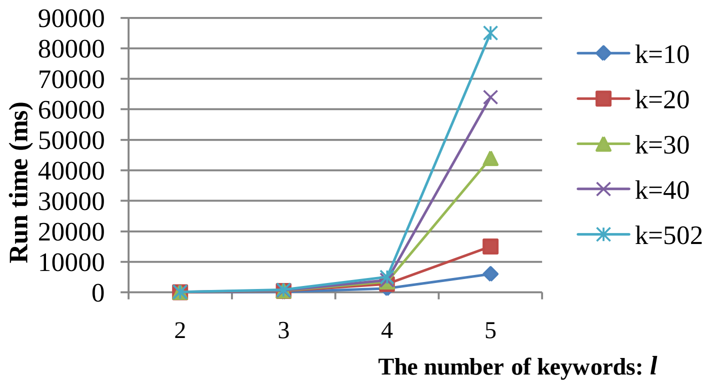

Figure 11　Run time (ms) vs. *l* (MDPBF1.2, *n*=1000)

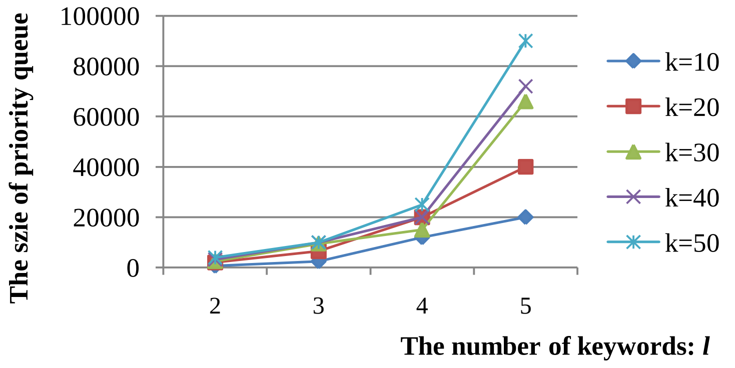

Figure 12　The size of priority queue vs. *l* (MDPBF1.2, *n*=1000)

From Figure 10, Figure 11 and Figure 12, we can see that MDPBF1.2 can work well when the number of keywords is less than 6 and the k is not bigger than 50. The maximum size of nodes set of tree is not more than 30.

We also performed experiments to test whether MDPBF1.2 can work well on large graph databases. The random graph contained 10000 nodes and 15000 edges. And the hit rate per

Vol.2010-DBS-151 No.28
2010/11/13

keyword is 0.1%. The result is shown in Table 1.

As notations:

- *k*: equals to 10
- L: the number of keywords
- Loops: the number of loops when GST-k are found
- Qsize: the maximum size of priority queue (keep all the candidate connected trees)
- Tmax: the maximum size of nodes sets of answer trees
- H 1: the rate of the trees with the same root node and leaf node in candidate trees
- N1 : the number of the trees with the same root node and leaf node in answers

From Table 1, we found that MDPBF1.2 just work well when the number of keywords is less than 4, that is because the graph is very sparse, and before we find the top-k, the program needs to repeat grow/merge many times and the size of priority queue becomes bigger and bigger. From this table, we also see that the rate of the trees with the same root node and the same leaf nodes in all the candidate trees is not so high, but in answer trees it is hard to see the trees with the same root node and the leaf nodes. If we use the expression (k/Qsize)*H1 to calculate the rate of the trees with the same root node and the leaf nodes in all answer trees, we can see the probability is very small.

For example, when the number of keywords is 3, (k/Qsize)*H1 (= (10/32000)*5%) equals to 0.001%.

Table 1　Test result about 10000 nodes (*k*=10)

L	Loops	Run time(ms)	Qsize	Tmax	H1	N1
2	2,000	800	6,500	15	2%	0
3	10,000	10,000	32,000	20	5%	0
4	23,000	170,000	100,000	25	7%	0

5.　Conclusions

This paper described redundant answers of DPBF-k and their elimination methods that are not referred in [1]. We also proposed an improved method named MDPBF to find the top-k min-cost connected trees exactly with a new redundancy elimination method. MDPBF is an improve algorithm of DPBF [1] based on repeating grow/merge operations to produce connected trees. The time complexity of MDPBF is $O((2^{l+1}mk + 3^{l}nk \cdot \log k)\log(2^{l}nk))$ and its space complexity is $O(2^{l+1}nk)$. We performed experiments to compare the answers and performance of DPBF with MDPBF. From the experimental results, we found that MDPBF can produce exact top-k. MDPBF1.1 can produce answers with higher quality, but MDPBF1.2 can reduce the size of priority queue and find top-k efficiently. Next we will try to balance the answer quality with search efficiency to make MDPBF work well on large graph databases.

References

1)　B.Ding, J.X. Yu,　L.Qin, X. Zhang, and X. Lin: Finding Top-k Min-Cost Connected Trees in Databases. In proc. of the 23rd International Conference on Data engineering (ICDE'07), pp.836-845 (2007).

2)　G. Bhalotia, A. Hugeri, C. Nakhe, S. Chakrabarti, S. Sudarshan. Keyword searching and browsing in database using banks. In Proc. of ICDE'02, pp.431-440 (2002).

3)　K. Varun, P. Shashank, C. Soumen, S. Sudarshan, D. Rushi, and K. Hrishikesh. Bidirectional expansion for keyword searches on graph database. In Proc.VLDB'05, pp.505-516 (2005).

4)　H. He, H. Wang, J. yang and P. S. Yu. BLINKS ranked Keyword Searches on Graphs. In SIGMOD'07, pp.305-316 (2007).

5)　S. Agrawal, S. Chaudhuri, and G. Das.　DBXplorer: A system for keyword-based search over relational databases. In Proc. of ICDE'02, pp.5-16 (2002).

6)　V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In Proc. of VLDB'03, pp.670-681 (2003).

7)　V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In Proc. of VLDB'03, pp.670-681 (2003).

8)　B. Kimelfeld and Y. Sagiv. Finding and Approximating Top-k Answer in Keyword Proximity Search. In ACM PODS'06, pp.173-182 (2006).