Engineering Applications of Artificial Intelligence 81 (2019) 169–179

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Improving the effectiveness of keyword search in databases using query logs☆

Ziqiang Yu [a], Ajith Abraham [b], Xiaohui Yu [c], Yang Liu [d], Jing Zhou [e], Kun Ma [a,*]

[a] Shandong Provincial Key Laboratory of Network Based Intelligent Computing, University of Jinan, Jinan 250022, China
[b] Machine Intelligence Research Labs, MIR, Auburn, USA
[c] York University, Toronto, Ontario M3J1P3, Canada
[d] Wilfrid Laurier University, Waterloo, Ontario N2L3C5, Canada
[e] Beijing Guoshuang Technology Co., Ltd., China

ARTICLE INFO

Keywords:
Keyword search
Top-*k*
Query log
Relational database

ABSTRACT

Using query logs to enhance user experience has been extensively studied in the Web IR literature. However, in the area of keyword search on structured data (relational databases in particular), most existing works have focused on improving search result quality via designing better scoring functions, without giving explicit consideration to query logs. However, query logs can reflect the user preferences, so our work taps into the wealth of information contained in query logs and aims to enhance the search effectiveness by explicitly taking into account the log information when ranking the query results. Different from existing approaches only relying on a schema graph or a data graph, our work designs a comprehensive solution based on both the schema graph and the data graph for discovering top-*k* results with two stages. First, we identify top-*k* candidate networks with a query-log-aware ranking strategy by employing the largest frequent subtrees mined from query logs. Since a candidate network usually corresponds to multiple joined tuple trees, we further rank these joined tuple trees with the PageRank principle based on the data graph in the second stage. Finally, user studies on a real dataset validate the effectiveness of the proposed ranking strategy.

1. Introduction

The success of keyword queries as a common way of web search and exploration has spurred much interest in the research community in supporting effective and efficient keyword search in relational databases. It allows information retrieval (IR) from databases by simply giving a set of keywords, without requiring users to know either query languages (such as SQL) or the database schema. A large body of literature has appeared in this area, which can be broadly classified into two categories: the schema graph based approach (e.g., DISCOVER (Hristidis and Papakonstantinou, 2002), DISCOVER-II (Hristidis et al., 2003), SPARK (Luo et al., 2007), SPARK2 (Luo et al., 2011), DBXplor (Agrawal et al., 2002), and QUEST (Bergamaschi et al., 2016)) and the data graph based approach (e.g., BANKS (Hulgeri and Nakhe, 2002), BANKS-II (Kacholia et al., 2005), PACOKS (Lin et al., 2016), CI-Rank (Yu and Shi, 2012), Blinks (He et al., 2007), and *DSize-l* and *PSize-l* OS (Fakas et al., 2015)).

Despite the recent advances in keyword search over databases, there still exist two major issues to be addressed. **The first one is few work explicitly incorporates the query feedback into the ranking of query results.** Here, we take schema graph based approaches as an example. This type of approaches (Hristidis and Papakonstantinou, 2002; Hristidis et al., 2003; Agrawal et al., 2002; Bergamaschi et al., 2016) execute the querying process by two steps: *candidate network* (*CN*) generation and *candidate network* (*CN*) evaluation. For instance, DISCOVER first traverses the tuple set graph expanded from the schema graph to generate all *CN*s, and then creates a execution plan to evaluate all generated *CN*s. To enhance the search efficiency, DISCOVER-II (Hristidis et al., 2003) introduces some pruning conditions to avoid generating unnecessary tuple trees and designs a greedy algorithm to produce a near-optimal execution plan. More studies have been done to further improve the search efficiency and the effectiveness of results such as Luo et al. (2007, 2011), Liu et al. (2006) and Yang et al. (2014). However, these works on the effectiveness issue primarily focus on returning results with basic semantics, while user preferences are not explicitly considered during the whole process. This is also true for most data graph based approaches. In contrast, a user is more likely to find the

☆ No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.engappai.2019.01.017.

* Corresponding author.
E-mail address: ise_mak@ujn.edu.cn (K. Ma).

https://doi.org/10.1016/j.engappai.2019.01.017
Received 24 June 2018; Received in revised form 12 September 2018; Accepted 25 January 2019
Available online 5 March 2019
0952-1976/

answer he/she is interested in if his/her preference (captured through search history, etc.) is taken into account when ranking the results.

As an example, consider the following scenario. Company *XYZ* is a wholesale supplier with geographically distributed warehouses, each of which serves several sales districts. A database D is used to manage the information of the company's products and customers. D consists of seven tables with the following schema: *warehouse* (*warehouseID, ...*), *district* (*districtID, warehouseID, ...*), *customer* (*customerID, districtID, warehouseID, ...*), *item* (*itemID, ...*), *stock* (*itemID, warehouseID, ...*), *order* (*orderID, districtID, warehouseID, customerID, ...*), *orderline* (*orderID, number, itemID, ...*).

We assume that the warehouses are named $W_1, W_2, \ldots, W_m$ and items $I_1, I_2, \ldots, I_n$. Intuitively, users of D from different departments of the company would want different information from the database even when they issue the same query. For example, when an employee from the sales department issues a query "W_i, I_j", she is likely to prefer retrieving information regarding the sales of I_j in warehouse W_i. Therefore, search results corresponding to the join $item \bowtie orderline \bowtie order \bowtie warehouse$ should be promoted towards the top of the ranked list of results. This preference can be naturally reflected in the query log through past queries issued by her or her colleagues in the sales department. In contrast, an employee from the distribution department, who often checks stock and distributes goods from warehouses to stores, may prefer the stock information of item I_j in the warehouse W_i ($item \bowtie stock \bowtie warehouse$) for the same search. Again, this preference can be reflected in the log of past queries.

Since users preferences are significant to the ranking of query results, some works draw into user feedbacks when integrating keyword search based data (Bergamaschi et al., 2016; Gao et al., 2011; Yagci et al., 2017; Peng et al., 2009; Qiao et al., 2018). Bergamaschi et al. (2016) study the expression of keyword search queries with terms of metadata structures of databases to make the queries better reflect user preferences. Gao et al. Gao et al. (2011) also employ query logs for keyword search, but the query logs in their work are only used to help improve the effectiveness of keyword query cleaning. Peng et al. Peng et al. (2009) explore how to better reformulate initial queries to retrieve more relevant query results in relational databases by applying user feedbacks. However, these works just focus on enhancing the quality of keyword search queries but not exploring a novel ranking strategy based on the user preferences mined from query logs. Our work (Zhou et al., 2015) adequately incorporates the user feedback into the ranking of *CN*s, but it lacks of a strategy about identifying the top-k *JTT*s for users.

Another problem faced with existing works is the gap between schema graph based and data graph based approaches. Schema graph based approaches usually aim to effectively search and rank the *CN*s, but rarely pay attention to further improve the performance of ranking *joined tuple trees* (*JTT*s) based on the data graph. For instance, DISCOVER-II (Hristidis et al., 2003) tries to rank the keyword search results by incorporating IR techniques, but it is only concerned about accelerating the generation of top-k *JTT*s based on selected *CN*s rather than designing a smarter score function to improve the ranking effectiveness of *JTT*s. As to the data graph based approaches (Hulgeri and Nakhe, 2002; Kacholia et al., 2005; Yu and Shi, 2012; He et al., 2007; Fakas et al., 2015), they usually discover and rank the *JTT*s based on the data graph straightly, so they usually consume much more time than the schema graph based approaches. An empirical performance evaluation (Coffman and Weaver, 2014) has shown that DISCOVER and DISCOVER-II are much faster than BANKS (Hulgeri and Nakhe, 2002) when they identify the same set of results on IMDB, MONDIAL, and Wikipedia datasets. Therefore, we believe that it will be useful to filter the irrelevant results for the data graph based approached by first identifying the *CN*s better meeting the user preferences, but few work attempt this way.

For the sake of above challenges, we introduce a two-cascading search approach employing user query logs to settle the keyword search query with two stages. The first stage is to generate and rank *CN*s incorporating the user preferences mined from query logs, because the query logs record the queries along with the results chosen by users for each query, which adequately reflect user feedbacks. In particular, we first mine the frequent patterns from the query log of every user. For a given query, we then score all *CN*s obtained by a standard *CN*-generating algorithm based on the schema graph, such as that from DISCOVER, by a new scoring function that combines the score based on the user query log and the score on the *CN* size through normalization and weighting. Finally, we choose top-k *CN*s better conforming to user preferences.

In the second stage, we investigate how to rank *JTT*s generated by the top-k *CN*s with the PageRank principle. We first construct a data graph corresponding to the database as Blinks (He et al., 2007) and view it as a web page graph, where each node is regarded as a web page. Just like calculating PageRank values for web pages, every node in the data graph is also assigned a PageRank value. We next design a score function that can evaluate the generated *JTT*s by combining the PageRank values of their nodes and their sizes, and then introduce a filter-and-pruning strategy to identify top-k *JTT*s.

So far, our work not only incorporates user preferences into *CN* generation based on the schema graph to narrow down the search space, but also further enhances the ranking effectiveness of *JTT*s with the PageRank principle based on the data graph. Our main contributions can be summarized as follows.

- We propose a two-cascading ranking approach that can absorb the advantages of schema graph based and data graph based approaches to improve the ranking effectiveness from two different aspects.
- We incorporate the user feedback into the sorting of *CN*s with the help of frequent patterns mined from user query logs, which not only enhances the ranking efficiency of *CN*s but also narrows down the search scope for the next stage.
- We prove the scoring process of *CN*s is NP-hard, and provide a dynamic programming algorithm to calculate the maximum score for a given *CN*.
- We devise a score function with the PageRank principle over a data graph to further sort the *JTT*s based on the selected top-k *CN*s.
- Extensive experiments and user studies are conducted to evaluate the proposed ranking strategy, and confirm its effectiveness.

The rest of the paper is organized as follows. Section 2 defines the vital conceptions and data structures used in this work. Section 3 presents the strategy of ranking *CN*s based on query logs. Section 4 describes the method with the PageRank principle to further rank *JTT*s based on the top-k *CN*s. Experimental results are presented in Section 5. Section 6 concludes this paper and discusses possible directions for future work.

2. Preliminaries

We consider a relational database with n relations $R_1, \ldots, R_n$. Each relation R_i has m_i attributes $a^i_1, \ldots, a^i_{m_i}$.

Definition 1 (*Labeled Directed Graph*)**.** Given a relational database D, we define the schema graph of D as a Labeled Directed Graph (LDG) $G = (V, E)$. Each node $v \in V$ represents the corresponding relation in D, and each edge $e \equiv v_i \rightarrow v_j$ ($v_i, v_j \in V$, $e \in E$) corresponds to a primary-key-foreign-key relationship between the relations represented by v_i and v_j. We assign unique ids (i.e., label) to all nodes and edges respectively.

Fig. 1 depicts a sample of five tables from the DBLP biography database (Zeng et al., 2012). The tables *Paper* and *Author* contain information on papers and researchers respectively; table *Conference* contains conference information. Table *PaperCitation* stores the citation

Paper

Pid	Title	Cid
P1	A hidden **Markov** model information retrieval system	C1
P2	Topics over time: a non **Markov** continuous-time model of topical trends	C2
P3	An HMM acoustic model incorporating various additional knowledge sources	C3
P4	ILDA: interdependent LDA model for learning latent aspects and their ratings from online product reviews	C4

Author

Aid	Name
A1	Miller
A2	Xuerui Wang
A3	**Markov**
A4	Moghuddam

Conference

Cid	Name
C1	22^{nd} International ACM SIGIR Conference
C2	12^{th} ACM SIGKDD international conference
C3	8^{th} Conference of the International Speech Communication Association
C4	34^{th} International ACM SIGIR Conference

Write

Aid	Pid
A1	P1
A2	P2
A3	P3
A4	P4

PaperCitation

Pid	CitedPid
P2	P4

Fig. 1. DBLP database sample.

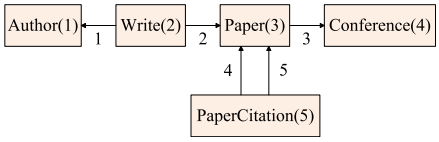


Fig. 2. The LDG of DBLP database.

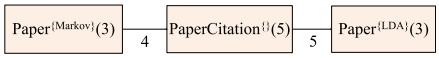


Fig. 3. Labeled free tree form of a *CN*.

relationships between papers; and table *Write* records the $m:n$ relationships between authors and papers. The LDG of the sample DBLP database from Fig. 1 is shown in Fig. 2.

Given a query $Q = \{k_1,\ldots, k_m\}$, where k_i is a keyword, we can obtain a set of basic tuple sets $\overline{R}_i^{k_j}$ ($i = 1,\ldots,n$, and $j = 1,\ldots,m$). The basic tuple set $\overline{R}_i^{k_j}$ consists of all tuples of relation R_i that contain the keyword k_j. Then the basic tuple sets are processed to produce tuple sets R_j^K for the non-empty subset K of Q. R_i^K, a *non-empty tuple set* that contains the tuples of R_i that contain all keywords of K and no other keywords, is defined as $R_i^K = \{t | t \in R_i \wedge \forall k \in K, t \text{ contains } k \wedge \forall k' \in Q - K, t \text{ does not contain } k'\}$. For example, $Paper^{Markov}$ is the set {P1,P2}, and $Paper^{LDA}$ is {P4}. The database relations that appear in the schema graph is *free tuple set* denoted as $R^{\{\}}$ which means that the relation R does not have tuples that contain a keyword. The non-empty tuple sets combine with the schema graph of the database by adding corresponding edges to form the tuple set graph G^{TS}.

Definition 2 (*Candidate Network*)**.** A candidate network (*CN*) is a join network of tuple sets formed by traversing G^{TS} in a breadth-first mode.

A *CN* is a portion of G^{TS} and can be considered as labeled free tree which belongs to the family of *free trees*-the connected, acyclic and undirected graphs. Fig. 3 illustrates the labeled free tree of one *CN* for the query "*Markov,LDA*", which represents the join network $Paper^{Markov} \bowtie PaperCitation \bowtie Paper^{LDA}$.

Definition 3 (*Joined Tuple Tree*)**.** Given the tuple set graph G^{TS} for a database, a joined tuple tree (JTT) is a tree of tuples, where each edge (t_i, t_j) satisfies the following properties: (1) $t_i \in R_i$, $t_j \in R_j$, (2) $t_i \bowtie t_j \in R_i \bowtie R_j$, (3) R_i and R_j belong to the same CN.

Corresponding to the *CN* $Paper^{Markov} \bowtie PaperCitation \bowtie Paper^{LDA}$, ${P_2}^{Markov} \bowtie (P_2, P_4) \bowtie {P_4}^{LDA}$ is a joined tuple tree.

Definition 4 (*Query Log*)**.** A query log L is a set of entries. Each user has his own query log. For a specific user, each entry in his/her user log records the candidate network (in the form of labeled free tree) that the user chose to visit when all the candidate networks were presented to him/her in answering a given query. In short, L contains the chosen results of a user for one or more queries.

Definition 5 (*F-subtree*)**.** For a specific user (or a group of similar users) u, his/her entries in the query log form a set L_u where each recorded *CN* $\mathbf{c} \in L_u$ is a labeled free tree. For a given pattern T_i (a free tree), we say that T_i occurs in a logged *CN* $\mathbf{c}$ or $\mathbf{c}$ supports T_i if T_i is isomorphic to a subtree of $\mathbf{c}$. The support of the pattern T_i is the number of *CN*s in L_u that support T_i. The pattern T_i is said to be a frequent subtree (**F-subtree**) if its support, $sup(T_i)$, is no less than a predefined minimum support (**minsup**).

Definition 6 (*LF-subtree*)**.** For a F-subtree T_i, if there does not exist a pattern T_i' such that T_i' is a F-subtree and T_i is covered by T_i', then T_i is a **LF-subtree** (largest frequent subtree). The problem of mining LF-subtrees from query logs is to compute, for a given user u and his/her log L_u, the set $\mathcal{P}(L_u) = \{T_i | sup(T_i) \geq minsup\}$, where T_i cannot be covered by any other pattern in $\mathcal{P}(L_u)$.

The essence of finding LF-subtrees from a query log is the problem of mining interesting patterns from a dataset, which is still a hot research topic in recent years (Alavi and Hashemi, 2015; Chi et al., 2003; Gan et al., 2017; Duong et al., 2018; Lin et al., 2018). Here, we employ the algorithm *FreeTreeMiner* described in Chi et al. (2003) to calculate $\mathcal{P}(L_u)$.

3. Ranking *CN*s with query logs

When producing the set of *CN*s, many works simply rank the *CN*s by their sizes. For instance, DISCOVER adopts the following formula for scoring a *CN* $\mathbf{c}$:

$$Score_{SIZE}(\mathbf{c}) = 1/Size(\mathbf{c}) \tag{1}$$

By Eq. (1), smaller *CN*s are ranked before larger ones, and ties are broken arbitrarily. Users get query results ranked by the sizes of their corresponding *CN*s, which could be very different from the users' real needs. As an example, consider the following case. Suppose that a user

issues a query "*Markov, LDA*". The *CN*s (i) $Paper^{Markov} \bowtie Conference \bowtie Paper^{LDA}$ and (ii) $Paper^{Markov} \bowtie Write \bowtie Author \bowtie Write \bowtie Paper^{LDA}$ are included in the results of the *CN* generation step. According to Eq. (1), *CN* (i) is ranked higher than (ii) as it has less joins and hence a smaller size. But if the query log contains entries related to this user's past queries, we can take them into consideration when ranking the *CN*s. For example, we consider an extreme case where there is no pattern *Paper* ⋈ *Conference* and the support of the pattern *Author* ⋈ *Write* ⋈ *Paper* is very large. Intuitively, *CN* (ii) should be ranked higher than (i) as the preference of the user can be clearly inferred from his/her search history.

The problem we study in this section, is how to adapt to user preferences through query logs. This can be further boiled down to the problem of ranking the generated *CN*s using information mined from query logs.

3.1. A naive method based on query logs

A straightforward method to incorporate the query log information is to assign, for each user, a degree of preference (e.g., $p \in [0, 1]$) for each table in the database based on frequency of that table appearing in the log. The score of a generated *CN* for a given query can be computed by a linear combination of the preference degree of each table involved and the size of the *CN*. However, this does not work in some cases as the score may be dominated by a minority (sometimes even one) of the tables in the *CN*. For example, for the query "*Markov, LDA*", we suppose there exist two corresponding candidate networks (i) $Paper^{Markov} \bowtie Write \bowtie Author \bowtie Write \bowtie Paper^{LDA}$ and (ii) $Paper^{Markov} \bowtie PaperCitation \bowtie Paper^{LDA}$. If the degree of preference for *Author* is much higher than other tables for that user, then *CN* (i) may be ranked higher than *CN* (ii). However, although the user has a strong preference for *Author*, it is very likely that for this particular query the user would prefer a pattern in which one paper cites another. In this example, the high preference degree of a single table *Author* has dominated the scoring of the *CN*. Someone perhaps doubts that since the size of *CN* (i) is greater than that of *CN* (ii), so the ranking score of *CN* (i) may be smaller than that of *CN* (ii). But in reality, we cannot guarantee the table with the high frequency always locates in a candidate network with larger size.

The second negative case caused by the dominated tables is that, the scores of different candidate networks containing dominated tables will be convergent at the same score, making their ranking order be difficult to be identified. This is because the dominated tables are also vital relations in a database, and they are much more frequently adopted by users, just like the table *Paper* in DBLP. To stick with the above example, if *Paper* owns a much higher preference degree than others, it will dominate the scores of *CN* (i) and *CN* (ii) to make these two scores be almost identical.

The last but not least reason is that in some cases, the join of frequent tables may not be frequent. For example, it is possible that two tables, say *Paper* and *Author*, are both of high frequency, but the join $Paper \bowtie Write \bowtie Author$ may be rare in this log. In this case, any *CN* with this join as a component should not be ranked high despite the high frequency of *Paper* and *Author*. Intuitively, instead of considering the frequency of single tables, we should focus more on the frequency of those "join structures". This leads us to develop the methods described in the sequel.

3.2. Ranking CNs with LF-subtrees

In this section, we discuss how to rank *CN*s with LF-subtrees mined from query logs. We first design a sophisticated score function to rank *CN*s incorporating the user feedback, and further explain why the calculation of the maximum score of a *CN* is a NP-hard problem. To address it, we then propose a dynamic programming algorithm to identify the best proper partition for a *CN*, which is the essential issue for computing the maximum score of the *CN*.

3.2.1. Score function for ranking CNs

Since the LF-subtree is frequently adopted by users, we naturally view that the *CN*s including one or more LF-subtrees will better satisfy user preferences than the ones without LF-subtrees. Inspired by this, we deem that the calculation of scores of *CN*s must incorporate their covered LF-subtrees. Hence, we seek to augment the scoring function in Eq. (1) with the LF-subtrees mined from query logs. Let $\mathcal{P}(L_u)$ be the set of LF-subtrees mined from query log L_u for a given user (or group of users) u. For a *CN* $\mathbf{c}$, we use $FS(\mathbf{c})$ to denote the set of LF-subtrees from $\mathcal{P}(L_u)$ such that any LF-subtree T_i is covered by $\mathbf{c}$, i.e., $FS(\mathbf{c}) = \{T_i | T_i \Subset \mathbf{c} \wedge T_i \in \mathcal{P}(L_u)\}$. The set of edges in $\mathbf{c}$ not covered by $FS(\mathbf{c})$, together with their corresponding vertices, constitute another set denoted by $NFS(\mathbf{c})$. Naturally, $NFS(\mathbf{c})$ and $FS(\mathbf{c})$ have no overlapping edge.

As to the *CN* $\mathbf{c}$, we define its a *proper partition* $P_\mathbf{c}$ as a complete non-overlapping cover of $\mathbf{c}$ by a combination of elements from $FS(\mathbf{c})$ and $NFS(\mathbf{c})$ such that

- There is no overlapping edge between any pair of elements; and
- The union of the edges in all of the elements in the combination is equal to the set of edges in $\mathbf{c}$.

Obviously, each edge of $\mathbf{c}$ is contained in exactly one element of the combination. Particularly, if the partition P_c has more than one LF-subtree, then any two different LF-subtrees in P_c cannot possess the same edge. The score of the proper partition $P_\mathbf{c}$ is assigned as follows.

$$score_{PAR}(P_\mathbf{c}) = \sum_{T_i \in LFS} score(T_i), \tag{2}$$

$$score(T_i) = \left(Size(T_i)/Size(\mathbf{c})\right)^t \cdot N(sup(T_i)) \tag{3}$$

where *LFS* denotes the set of frequent LF-subtrees used in the partition and $sup(T_i)$ is the support of T_i ($T_i \in LFS$). The configurable parameter t ($t \geq 1$) is used to control the degree of preference for larger frequent structures, and $N(\cdot)$ is a normalization function to be described later. Notice that elements in $NFS(\mathbf{c})$ do not contribute the score of the partition.

In Eq. (3), normalization is applied to the support. The support of a frequent pattern mined from the query log ranges from *minsup* to an unknown large number. If the value of $sup(T_i)$ is too big, the score of the *CN* that contains the subtree T_i will become unreasonably large. Therefore, normalization must be done to limit the influence of the support value. In particular, when the support is extremely large, its effect should be dampened even more. Based on the above consideration, we use a sigmoid function as a starting point for normalizing the support values, which takes the form of $sigmoid(x) = 1/(1 + e^{-\alpha x})$, where the weight parameter α controls the linearity of the curve. In our case, the range of support is [*minsup*, $+\infty$). As *minsup* is greater than zero, the range of the sigmoid function is (0.5,1). However, the range of Eq. (1) is [0,1]. So, we have to scale the range of the sigmoid function to the range of (0,1) by the transformation $N(x) = 2 \cdot (sigmoid(x) - 0.5)$.

Note that there may exist many proper partitions for a candidate network. Let $\mathcal{P}_\mathbf{c}$ denote the set of all proper partitions of $\mathbf{c}$, each of which has a corresponding score computed by Eq. (2). The largest such score is used as the *query log score*, as indicated in Eq. (4). The rule comes from the intuition that we always want to get a partition in which the LF-subtrees have both larger support and larger size. But in fact, smaller LF-subtrees in $\mathcal{P}_\mathbf{c}$ have larger supports. Thus, a trade-off is needed. So, a candidate network is assigned the largest combination score as the log score.

$$score_{LOG}(\mathbf{c}) = max\{score_{PAR}(P_\mathbf{c}) | P_\mathbf{c} \in \mathcal{P}_\mathbf{c}\} \tag{4}$$

Finally, we combine the original size-based score and the query log score by weighting as in Eq. (5), where λ is the weight of the original score and controls the relative importance between the two parts. If $\lambda = 1$, then the new scoring function is the same as the one used in DISCOVER.

$$Score(\mathbf{c}) = \lambda \cdot Score_{SIZE}(\mathbf{c}) + (1 - \lambda) \cdot Score_{LOG}(\mathbf{c}) \tag{5}$$

To illustrate the advantage of this new scoring function, we also consider the example given at the beginning of Section 3. For the query {"*Markov*, *LDA*"}, two example *CN*s (i) $Paper^{Markov} \bowtie Conference \bowtie Paper^{LDA}$ and (ii) $Paper^{Markov} \bowtie Write \bowtie Author \bowtie Write \bowtie Paper^{LDA}$ are assigned with an score respectively by our ranking functions. As $\mathcal{P}(L_u)$ has no the pattern *Paper* ⋈ *Conference* while the pattern *Author* ⋈ *Write* ⋈ *Paper* has a large support, $S(i) < S(ii)$. Apparently, by incorporating the query log, the generated candidate networks can be ordered emerging user preferences.

So far, we can compute the largest score for *CN* **c** with Eq. (5) if its log score has been identified. However, since **c** probably corresponds to numerous proper partitions, so it turns to be a NP-hard problem to compute the maximum combination score (i.e., log score) corresponding to the *best proper partition* of **c**, and which will be shown in the next section.

3.2.2. Complexity of identifying the best proper partition

We first introduce the notations that will be used. Let Ψ be the set of all edges of *CN* **c**; $\mathbb{S} = FS(\mathbf{c}) \cup NFS(\mathbf{c})$. Then a LF-subtree can be represented with a subset of Ψ.

Definition 7 (*Best Proper Partition Problem*)**.** With Ψ and $\mathbb{S}$ as input, the best proper partition problem is to find a set $S^* \subseteq \mathbb{S}$ such that each element of Ψ appears in only one element of S^* and S^* maximizes $score(S^*) = \sum_{S \in S^*} score(S)$.

In the above definition, each element S in S^* is indeed a LF-subtree covered by **c** because the edges in $NFS(\mathbf{c})$ do not contribute to the query log score of **c**. Hence, our problem can be considered an optimization problem formulated as follows.

- *Instance:* Given a set of elements Ψ, and the set of LF-subtrees $FS(\mathbf{c}) = \{T_1, T_2, \ldots, T_n\}$, where T_i $(1 \le i \le n) \subseteq \Psi$ and has a corresponding score w_i.
- *Question:* Find a set $S^* \subseteq FS(\mathbf{c})$, which ensures that each element of Ψ appears in one and only one element of S^* and $\sum_{T_i \in S^*} w_i$ is maximized.

Correspondingly, the decision version of the *best proper partition problem*, can be formulated as follows.

- *Instance:* Given a set of elements Ψ, a set of subsets of Ψ, $FS(\mathbf{c}) = \{T_1, T_2, \ldots, T_n\}$, and a constant B, where T_i $(1 \le i \le n) \subseteq \Psi$ and has a corresponding score w_i.
- *Question:* Is there a set $S^* \subseteq \mathbb{S}$ such that each element of Ψ just in one and only one element of S^* and $\sum_{T_i \in S^*} w_i \geqslant B$?

Theorem 1. *The best proper partition problem is NP-hard.*

It is sufficient to prove best proper partition decision problem is NP-Complete. We can apply the *restriction* technique which shows the NP-Completeness of an NP problem by stating that a special case of the problem is NP-Complete. By limiting $B = min\{w_i | T_i \subseteq S^*\}$, the decision problem can be restricted to the *exact cover problem*, a problem known to be NP-Complete. Then, the decision problem is proved to be NP-Complete. Hence, the *best proper partition problem* is NP-hard.

3.3. Identification of the best proper partition

We now discuss the discovery of the *best proper partition* by dynamic programming. Assuming that the elements in $FS(\mathbf{c})$ are numbered, we define a set of indicator variables x_i for a given set $S \subseteq \mathbb{S}$ such that $x_i = 1$ if the ith element in $FS(\mathbf{c})$ appears in S , and $x_i = 0$ otherwise. Then an indicator vector $(x_1, x_2, \ldots, x_{|FS(\mathbf{c})|})$ can be formed.

Our problem can be considered as maximizing the following function with respect to S

$$F(S) = \sum_{i=1}^{|FS(\mathbf{c})|} x_i \cdot score(T_i)$$

subject to the constraints: (i) $T_i \bigcap T_j = \emptyset$, if $x_i = 1, x_j = 1, 1 \le i, j \le |FS(\mathbf{c})|$, and $i \ne j$ (ii) $\bigcup_{i=1}^{|FS(\mathbf{c})|} x_i T_i \subseteq \Psi$ and (iii) $x_i \in \{0, 1\}$, where $score(T_i)$ is calculated by Eq. (3). The constraint (i) ensures that for a partition no pair of LF-subtrees share the same edges, and (ii) makes a partition consisting of multiple LF-subtrees is still a subset of Ψ.

3.3.1. SPP Algorithm

According to the above rule, we design SPP, an algorithm based on dynamic programming, to identify the *best proper partition* of **c**. This algorithm includes following two phases.

Phase 1 Constructing RLF index to manage all LF-subtrees. We first build an Relevant LF-subtree (RLF) index to manage $\mathcal{P}(L_u)$, the set of all LF-subtrees mined from the query log of the user u. For any LF-subtree T_i in RLF, it has a relevant LF-subtree list L_{T_i}, which records all LF-subtrees that at least have one common edge with T_i. For every element in RLF, we define a *Bound Score* (*BScore* for short) as Eq. (6). Based on this equation, we can deduce that for any two elements T_i and T_j, if $BScore(T_i)$ is greater than $BScore(T_j)$, then $score(T_i) > score(T_j)$, where $score(\cdot)$ is defined as Eq. (3). This rule is proved in Theorem 2. The reason why we use $BScore(\cdot)$ instead of $score(\cdot)$ is that the calculation of $score(\cdot)$ always involves the size of a specified *CN*, while the *BScore* of a LF-subtree is only associated with its size and frequency but has no relationship with any *CN*, so the *BScore* of every LF-subtree can be calculated beforehand in off-line. With the help of *BScore*, we unveil a property that can promote the discovery of the best proper partition by filtering the unavailable combinations. That is, for two LF-subtrees T_i and T_j, and a candidate partition P_c of **c** to be expanded, if T_i and T_j both can be added into P_c and $BScore(T_i) \ge BScore(T_j)$, putting T_i into P_c will make **c** have a greater combination score.

$$BScore(T_i) = Size(T_i)^t \cdot N(sup(T_i)) \tag{6}$$

Theorem 2. *For any two LF-subtrees T_i and T_j, if $BScore(T_i) \ge BScore(T_j)$, then $score(T_i) \ge score(T_j)$ for any candidate network.*

Proof. Due to $t \ge 1$, we can infer that $Size(\mathbf{c})^t \ge 1$. Since $Size(T_i)^t \cdot N(sup(T_i)) \ge Size(T_j)^t \cdot N(sup(T_j))$, we can deduce that $\left(Size(T_i)/\tau\right)^t \cdot N(sup(T_i)) \ge \left(Size(T_j)/\tau\right)^t \cdot N(sup(T_j))$, where τ is a positive constant. Set $\tau = Size(\mathbf{c})$, then $\left(Size(T_i)/Size(\mathbf{c})\right)^t \cdot N(sup(T_i)) \ge \left(Size(T_j)/Size(\mathbf{c})\right)^t \cdot N(sup(T_j))$ must be established. □

We calculate the *BScore* of every LF-subtree in RLF and rank them based on their *BScores* in a descending order, which will make the elements with greater *BScores* be processed first. To save the space, we represent the LF-subtree list of an element in RLF as a vector $(x_1, x_2, \ldots, x_{|\mathcal{P}(L_u)|})$, where $x_i = 1$ means T_i has at least one common edge with this element; $x_i = 0$ otherwise.

Phase 2 Identifying the best proper partition of c. For the *CN* **c**, we first identify $FS(\mathbf{c})$, the set of LF-subtrees covered by **c** with the LF-CN-M algorithm that will be presented in next section. After ranking LF-subtrees in $FS(\mathbf{c})$ according to their *BScores*, we then can identify the best proper partition of **c** with multiple rounds of scanning RLF. In the first round, suppose P_b is the proper partition to be determined and let $\mathcal{L} = FS(\mathbf{c})$. At the beginning, we add T_f, the first element of $\mathcal{L}$, into P_b and visit RLF index to obtain L_{T_f}, the relevant LF-subtree list of T_f. According to Definition 7, we infer that the LF-subtrees in L_{T_f} cannot be included by P_b. So from the set $\mathcal{L}$, we remove the common elements both in L_{T_f} and $\mathcal{L}$ (i.e., $\mathcal{L} = \mathcal{L} - L_{T_f}$), and delete T_f from $\mathcal{L}$. After this, we continue to choose the first LF-subtree $T_f{}'$ from $\mathcal{L}$ and add it into P_b to enlarge the $BScore$ of P_b denoted by $BS(P_b)$, where $BS(P_b) = \sum_{T_j \in P_b} BScore(T_j)$. Similarly, we update the set $\mathcal{L}$ as $\mathcal{L} = \mathcal{L} - L_{T_f{}'}$ and remove $T_f{}'$ from $\mathcal{L}$, and then dispose the surplus elements in $\mathcal{L}$ to expand P_b with the same way until $\mathcal{L}$ becomes empty.

In the ith round $(i \ge 2)$, we also set $\mathcal{L} = FS(\mathbf{c})$ and then remove the former $(i\text{-}1)$ elements from $\mathcal{L}$. Like the first round, we process the surplus elements of $\mathcal{L}$ to form a partition P_c with $BS(P_c)$ as large as

possible. If $BS(P_c) > BS(P_b)$, we set $P_b = P_c$ to label the current best proper partition; otherwise, the partition P_c can be safely discarded.

After $|FS(\mathbf{c})|$ rounds detection, we can verify that P_b must be the best proper partition because every possible combination of LF-subtrees in $FS(\mathbf{c})$ has been detected. The pseudo-code of SPP is shown in Algorithm 1. Once P_b is identified, we can deduce that $Score_{LOG}(\mathbf{c})$ equals to $Score_{PAR}(P_b)$ and then get $Score(\mathbf{c})$ finally.

To enhance search efficiency, we introduce a pruning rule that can timely terminate the search round that has no possibility to generate a new partition with a greater score than that of the current proper partition. Specifically, when processing the elements in $\mathcal{L}$ in the ith round, we assume that the current partition being identified is P_c. If $BS(P_c) + \sum_{T_i \in \mathcal{L}} BScore(T_i) \leq BS(P_b)$, then this round detection can be terminated because $BS(P_c)$ cannot be greater than $BS(P_b)$ even all elements in $\mathcal{L}$ can be added into P_c. To apply this pruning rule, we will keep $\sum_{T_i \in \mathcal{L}} BScore(T_i)$, the summary of *BScores* of all elements in $\mathcal{L}$ in every round, and update this value with the varying elements in $\mathcal{L}$.

Algorithm 1 SPP

Require:
$\mathcal{P}(L_u)$; *CN* **c**
Ensure:
Best proper partition P_b
1: $FS(\mathbf{c})$=LF-CN-M ($\mathcal{P}(L_u)$, **c**);
2: $\mathcal{L} = \phi$, $P_b = \phi$, i=0;
3: **while** $i \leq |FS(\mathbf{c})|$ **do**
4: $\mathcal{L}$=$FS(\mathbf{c})$, $P_c = \phi$;
5: **if** $i \neq 0$ **then**
6: Remove the former i element(s) from $\mathcal{L}$;
7: **end if**
8: **while** $\mathcal{L} \neq \phi$ **do**
9: Put the first element T_f of $\mathcal{L}$ into P_c and remove T_f from $\mathcal{L}$;
10: Get L_{T_f} from the RLF index, and $\mathcal{L} = \mathcal{L}$-$L_{T_f}$;
11: **if** $BScore(P_c)+\sum_{T_i \in \mathcal{L}} BScore(T_i) \leq BScore(P_b)$ **then**
12: **break**;
13: **end if**
14: **end while**
15: **if** $BScore(P_b) \leq BScore(P_c)$ **then**
16: $P_b = P_c$;
17: **end if**
18: **end while**
19: **return** P_b;

3.3.2. LF-CN-M algorithm

Now, we discuss LF-CN-M, the algorithm used in SPP to identify $FS(\mathbf{c})$, the set of LF-subtrees covered by **c**. In this algorithm, we have to compare every LF-subtree and **c** to detect whether the LF-subtree is isomorphic to a subgraph of **c**, which is a NP-complete problem (Liu and Qiao, 2014). To address the graph isomorphism issue, extensive algorithms such as Ullmann, VF, VF2 and GNCCP (Liu and Qiao, 2014) have been proposed to discover all isomorphism subgraphs from the target graph. But in our study, we are only concerned about whether a LF-subtree T_i is isomorphic to a subgraph of **c** but not searching all isomorphism subgraphs of **c** to T_i. Hence, we design a LF-subtree and Candidate Network Matching (LF-CN-M) algorithm to accommodate the characteristics of our research issue.

Before presenting LF-CN-M algorithm, there still exist two preprocessing tasks. Firstly, we set an unique natural number as the identifier of each relation in the database, to conveniently label the different relations. Secondly, we will adjust the structures of every LF-subtree and the candidate network. In particular, for a LF-subtree T_i, we select the node (i.e., relation) with the smallest number as its new root and then reorganize it to make the sibling nodes be in an ascending order from left to right based on their identifiers. As to *CN* **c**, we make every node of **c** as a root and transform **c** into a corresponding tree. Thus we will need to keep multiple different trees corresponding to **c**. Fig. 4 gives an example about illustrating the reorganization process.

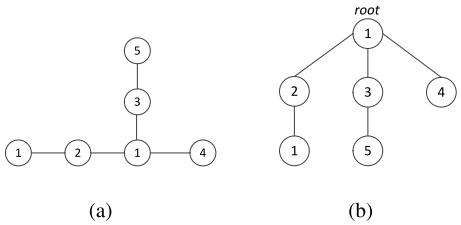


Fig. 4. LF-subtree reorganization. Fig. 4(a) represents a LF-subtree and Fig. 4(b) is this LF-subtree that has been reorganized.

After the preprocessing tasks, LF-CN-M can be applied to discover all LF-subtrees being isomorphic to subgraphs of **c** and its pseudocode is displayed in Algorithm 2. In LF-CN-M, we process every node of **c** in an ascending order based on their identifiers. Suppose the sequential order of nodes is $\{n_1, n_2, \ldots, n_s\}$, we first identify $\mathcal{F}_i$, a set of LF-subtrees whose roots have the same identifier with n_i ($1 \leq i \leq s$) and then compare every LF-subtree in $\mathcal{F}_i$ with the tree corresponding to **c** with n_i being root. This comparison can be achieved by MatchGraph algorithm, and we provide its pseudocode in Algorithm 3.

Suppose the tree with n_i being root matching **c** is ${T_c}^i$, then we present how to compare ${T_c}^i$ with a LF-subtree T_j in $\mathcal{F}_i$ using MatchGraph. We set the root of T_j as n_r and detect whether n_i and n_r satisfy two following conditions: (1) the degree of n_r must be not greater than that of n_i; (2) the children of n_r must be a subset of the children of n_i. If n_i and n_r do not meet these two conditions, then T_j is not covered by **c** and the detection can be terminated. Otherwise, we say n_i and n_r are matched and then continue to compare their children until all nodes of T_j have been scanned. If all nodes of T_j could be traversed, which indicates that T_j is covered by **c**; otherwise, T_j is not a subgraph of **c**. In this process, the dominated task is extensive comparisons of two children sets for two different nodes, but this can be accelerated by the preprocessing work. This is because we have reorganized LF-subtrees and **c**, making the LF-subtree to be compared has the same root with the tree of **c** and the children of any node in these trees are also ordered, which can greatly cut down the unnecessary comparisons.

Once all LF-subtrees in $\mathcal{F}_i$ have been compared with **c**, LF-CN-M will continue the next round detection, that is, comparing **c** to the LF-subtrees in $\mathcal{F}_{(i+1)}$. Like this way, $FS(\mathbf{c})$ can be identified after comparing **c** with all LF-subtrees in $\mathcal{P}(L_u)$.

Algorithm 2 LF-CN-M

Require:
CN **c**, $\mathcal{P}(L_u)$
Ensure:
$FS(\mathbf{c})$
1: Initialize two sets $\mathcal{F}$ and $\mathcal{R}$ respectively;
2: Rank the nodes in **c** based on their identifiers in an ascending order;
3: **for** each node $n_i \in \mathbf{c}$ **do**
4: Find the LF-subtrees with roots being n_i and put them into the set $\mathcal{F}$;
5: **for** each LF-subtree T_i in $\mathcal{F}$ **do**
6: **if** MatchGraph(T_i, **c**)==true **then**
7: Add T_i into $\mathcal{R}$;
8: **end if**
9: **end for**
10: **end for**
11: **return** $\mathcal{R}$;

Theorem 3. *For a given LF-subtree T_i with n nodes and the CN* **c***, the time complexity of MatchGraph is $O(n \cdot m)$, where n and m separately represent the number and maximum degree of nodes in T_i.*

Proof. Since T_i has n nodes, we can deduce that MatchGraph at most conducts n times comparisons even if **c** has more than n nodes. In every comparison, the major task is detecting whether the children of two

Algorithm 3 MatchGraph

Require:
CN **c**
A LF-subtree T_i
Ensure:
A boolean value
Initialize two queues *LF-queue* and *CN-queue* respectively;
Put the roots of T_i and **c** into $LF-queue$ and $CN-queue$ separately;
while $LF-queue \neq empty$ **and** $CN-queue \neq empty$ **do**
  $n_l = LF-queue$.getFirstElement();
  $n_c = CN-queue$.getFirstElement();
  if $n_l.degree == n_c.degree$ **and** $n_c.getChildren() == n_c.getChildren()$ **then**
    Put the children of n_l into $LF-queue$;
    Put the children of n_c into $CN-queue$;
  else
    Return false;
  end if
  if $LF-queue == empty$ **then**
    Return true;
  end if
end while

different nodes coming from T_j and **c** are identical, and the time cost of this operation is at most $((m + m') \times t_c)$, where m and m' are supposed as the maximum degrees of the nodes in **c** and T_j respectively, and t_c is a constant time to detect whether two nodes have the same identifier. Here, we can suppose $m' = m$, then the total time cost of conducting n comparisons is $2m \times n \times t_c$, so the time complexity of MatchGraph is $O(n \cdot m)$. □

4. Ranking *JTT*s with the PageRank-based principle

In above section, we give the method to rank *CN*s with incorporating user preferences. However, a *CN* usually corresponds to multiple *JTT*s and these *JTT*s still need to be ranked effectively. For this, we propose SPP-PR based on SPP, a PageRank-based method to further rank the *JTT*s generated by top-*k* *CN*s. In SPP-PR, we first construct a weighted directed graph G(V, E) corresponding to the database. Each tuple t_i is represented by a node v_i in V. For any two nodes v_i and v_j, there is a directed edge $\langle v_i, v_j \rangle$ and a backward edge $\langle v_j, v_i \rangle$ if and only if there exists a foreign key on tuple t_i that refers to the primary key tuple t_j. Further, we can view the data graph as a graph of web pages (in the original PageRank context), where each node corresponds to a page and the edge represents a reference from one page to another, we are then able to compute the PageRank value w_i for any node n_i in an off-line fashion. For a selected *CN* **c** matching a keyword query q, we suppose it corresponds to a set of *JTT*s, $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$, where J_i ($i \in [1, n]$) covers all keywords of q. For a *JTT* J_i, we defines its score as $S_{PR}(J_i) = (1 - e^{-2\log_{S(J_i)} W})/(1 + e^{-2\log_{S(J_i)} W})$, where $S(J_i)$ is the size of J_i, and W represents the summary of PageRank scores of all nodes in J_i, i.e., $W = \sum_{n_i \in J_i} w_i$. Since $\log_{S(J_i)} W$ belongs to (0, $+\infty$) in theory, so the value range of $S_{PR}(J_i)$ is (0,1). Unlike the score functions that only employ scores of the tuples covering keywords for evaluating joined tuple trees in many existing works (Hristidis et al., 2003), our score function combines the PageRank values of all tuples in a *JTT*. This is because the PageRank values of free tuples also probably have positive influence on the ranking of *JTT*s. We take $Paper^{Markov} \bowtie Write \bowtie Author \bowtie Write \bowtie Paper^{LDA}$ as an example. If a generated *JTT* contains a tuple in *Author* with a higher PageRank value, it is reasonable to enhance the ranking of this *JTT* because an author with higher PageRank value probably better meets the user preferences.

Since the score function of a *JTT* based on the PageRank principle has been identified, the next purpose is to search the top-k *JTT*s with maximum scores. Here, we cannot directly apply the evaluation algorithms proposed by DISCOVER-II (Hristidis et al., 2003) to rank *JTT*s because of the different scoring metrics, so we design the following algorithm. Suppose the top-*k* *CN*s selected in the above section is $C = \{c_1, c_2, \ldots, c_k\}$, for any candidate network c_i, we calculate the PageRank score for every *JTT* corresponding to c_i and then rank these *JTT*s in an descending order in terms of their PageRank scores. We first select k *JTT*s of c_1 and put them into a set $\mathcal{J}_r$. If c_1 has more than k *JTT*s, we set the PageRank score of the kth *JTT* as a pruning bound; otherwise, we continue to process the *JTT*s matching c_{i+1} ($1 < i + 1 \leq k$) until the number of elements in $\mathcal{J}_r$ reaches k, and set the PageRank score of the kth element as the pruning bound. After this, if we encounter a *JTT* J_x corresponding to c_i and its PageRank score is greater than the pruning bound, we will remove the kth element from $\mathcal{J}_r$ and add J_x into it, and then update the pruning bound; If the PageRank score of J_x is not greater than the pruning bound, we can safely discard J_x as well as the elements after J_x, and then continue to process the *JTT*s matching the next candidate network c_{i+1}. In this way, the *JTT*s in $\mathcal{J}_r$ will be the top-k results after all *CN*s in C have been processed.

5. Experiments

We conduct experiments to evaluate our proposed methods and compare them with an existing approach that does not consider user feedbacks.

5.1. Dataset and settings

Due to the lack of publicly available databases with query logs, we use the DBLP database[1] in our experiments and build our own query log through a controlled user study. The dataset is about 870MB and contains 12,594,911 tuples.

The DBMS used is MySQL with default configurations. We build indexes for all primary keys and foreign keys. Full-text indexes are built for all textual attributes. The experiments are conducted on a workstation with a 2.71 GHz Intel Core i5-7200u processor and 8GB of main memory.

The query set comes from a user study. Ten graduate students from different research areas participate in our experiment as query initiators. They formulated 60 "meaningful" queries consisting of varying number of keywords related to their research areas. For each participant, we applied 3-fold cross-validation on his queries. The queries were randomly divided into three sets. In each trial, two folds were used as the training set to generate the query log and the other was testing set. For each query in the training set, with the predefined parameters (such as T_{max}), all of the generated *CN*s, the number of which may range from tens to hundreds , were presented to the corresponding participant in sequence, in ascending order of size; the participant was asked to choose *"yes"* or *"no"* for each *CN* according to whether that *CN* meets his/her requirement. The *"yes"* *CN*s were recorded in the query log. Up to here, each participant had his own query log in each trail. We set *minsup*=10. By the settings above, each participant has more than 200 frequent LF-subtrees on average and the corresponding supports range from 10 to about 300. For each query in testing set, all the generated *CN*s were ranked by our methods and presented to the participant. The participant assessed the result quality using a six-point scale ranging from 0 to 5 (5="perfect" and 0="bad"). Similarly, the *JTT*s generated by SPP-PR also can be evaluated by the participants with the same strategy.

We use SPP and SPP-PR to denote our proposed approaches. SPP is the approach that ranks *CN*s by incorporating the user feedback, and SPP-PR further ranks *JTT*s with the PageRank principle. The parameters involved in the experiments are illustrated in Table 1 with explanation. And Table 2 shows some sample queries from a user. A user is required to issue queries that are reasonable. Generally, a query contains at least two keywords and no more than four.

[1] http://dblp.uni-trier.de/xml/

Table 1
Parameter under investigation.

λ	weight of the original score
t	preference for larger structure in a *CN*
T_{max}	the maximum allowed *CN* size
α	the Sigmoid function parameter
K	top-K results

Table 2
Query example.

Q_1: *bender, p2p*	Q_6: *Hardware, luk, wayne*
Q_2: *sigmod, xiaofang*	Q_7: *Ishikawa, P2P, Yoshiharu*
Q_3: *fagin, middleware*	Q_8: *hongjiang, Multimedia, zhang*
Q_4: *Owens, VLSI*	Q_9: *vldb, xiaofang*
Q_5: *p2p, Steinmetz*	Q_{10}: *intersection, nikos*

5.2. Effectiveness

To measure the effectiveness, we adopt four metrics, namely, *Normalized Discounted Cumulative Gain* (NDCG), *Precision at K* (P@K), *11-point Precision/Recall* and *F-measure*. Each is described in detail as follows.

- **NDCG at K:** For a given query q in the testing set, the ranked candidate networks and *JTT*s are assessed manually to compute NDCG@K-SPP and NDCG@K-SPP-PR. NDCG is computed as:

$$NDCG_q = \frac{DCG_q}{IDCG_q}, DCG_q = \sum_{i=1}^{k} \frac{2^{r(i)} - 1}{log_2(i+1)}$$

  where $r(i)$ is an integer given by the user to indicate the degree of the preference for the result returned at position i and $r(i)$'s values fall inside [0,5]; $IDCG_q$ is the discounted cumulative gain for the optimal ranking of query q's candidate networks, namely, *Ideal Discounted Cumulative Gain*(IDCG); $NDCG_q$ is the normalization result of DCG_q by $IDCG_q$, ranged from 0 to 1. The motivation of using NDCG@K is to pay more attention to the top-k results.
- **Precision at K:** P@K shows the fraction of the candidate networks ranked in top K results that are preferred by the user. In our settings, we define that a candidate network assessed with 3 or larger is preferred. The position of preferred candidate networks within top K is unconcerned. As the most intuitive metric, Precision@K measures the overall user satisfaction with the top K results.
- **11-point Precision/Recall:** For a query result, this metric reports the precision that is measured at the 11 recall levels of 0.0, 0.1, 0.2, …, 1.0. In our experiments, 11-pt Precision/Recall is the average result for all the testing queries.
- **F-measure:** Given K, we can compute the precision and the recall at K. Here, the candidate networks with 3 or larger points are preferred. F-measure is the harmonic mean of precision and recall and it is defined as $F\text{-}measure = \frac{(\alpha^2+1)precision \cdot recall}{\alpha^2(precision+recall)}$. Here, we set α as 1, then deduce that $F\text{-}measure = 2 \cdot \frac{precision \cdot recall}{precision+recall}$.

Effect of parameter t, α and λ of *Dynamic*. Set $T_{max} = 7$ and K=10. Table 3 shows the performances of SPP and SPP-PR with respect to NDCG@10 by varying the values of these parameters. Each line in Table 3 indicates that when fix the value of t and then vary α and λ, the best setting of α and λ and the corresponding result are presented (we do not show other settings here which only generate worse results). Observed from the results, we find t has an obvious impact on the performance of SPP but little influence on the effectiveness of SPP-PR. This is because we find the top-k *CN*s generated by SPP are almost identical but their ranked positions change significantly when t takes different values. Hence, NDCG@10-SPP will be affected obviously by t. As to SPP-PR, it will further rank the corresponding *JTT*s of every *CN* based on the PageRank principle, which causes the ranked positions of *JTT*s are almost irrelevant with that of *CN*s, so NDCG@10-SPP-PR almost remains the same when t takes different values. But when t=2, there is one different *CN* in the top-k *CN*s compared with t being other values, which leads to some *JTT*s appearing in final results, making the value of NDCG@10-SPP-PR become a little smaller. Observed from the table, we also conclude that when t (preference to larger frequent subtrees) is 4, NDCG@10-SPP and NDCG@10-SPP-PR both reaches the best values 0.890 and 0.912 respectively with $\alpha = 0.01$ and $\lambda = 0.1$.

Table 3
NDCG@10 for varying t, α and λ.

t	2	3	4	5	6
α	0.07	0.09	0.01	0.025	0.001
λ	0.7	0.5	0.1	0.1	0.3
NDCG@10-SPP	0.836	0.880	**0.890**	0.876	0.873
NDCG@10-SPP-PR	0.864	0.913	**0.913**	0.913	0.913

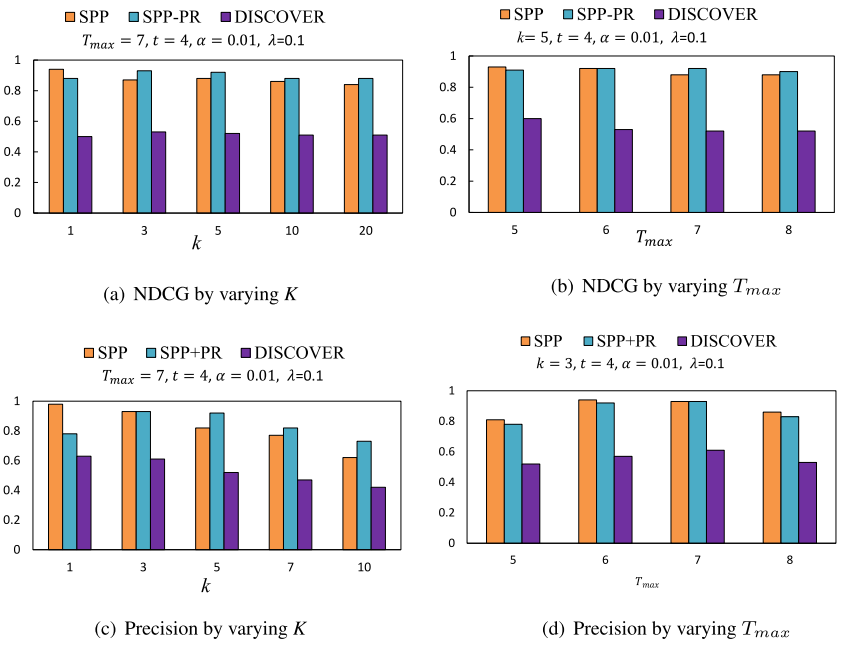
Effectiveness comparison for DISCOVER, SPP, and SPP-PR. We use DISCOVER as a baseline here as its scoring function directly corresponds to the "original score" part of the proposed new strategy. Fig. 5 shows that incorporating query log results in significant improvements over DISCOVER. In Fig. 5(a) ($T_{max} = 7$), when K increases, the performances of SPP and SPP-PR both decreases slightly. This is because there will be more rank orders of the results when the value of k increases, then the possibility of identifying the rank order accepted by most users from all rank orders will decrease. Even so, SPP and SPP-PR outperform DISCOVER by a considerable margin.

The influence of T_{max} on the ranking effectiveness of three approaches is shown in Fig. 5(b). When we vary T_{max}, we find that the gap between our proposed algorithms and DISCOVER remains significant; meanwhile, SPP and DISCOVER both progress with a downward trend when T_{max} increases, that is, the overlarge value of T_{max} will decay the performance of SPP and DISCOVER. However, the varying value of T_{max} has no obvious impact on NDCG of SPP-PR, this is because SPP-PR will further rank the *JTT*s based on the selected *CN*s, which can reduce the impact of T_{max} on the performance of SPP-PR only if the selected top-k *CN*s remain the same.

Fig. 5(c) shows the precisions of three approaches by varying k. From the results, we find the precisions of SPP and DISCOVER both decrease when the value of k increases, but the precision of SPP-PR increases first and then decreases. This because the results returned by SPP are *CN*s, and the optimal *CN*s are easily to be identified according to the user feedback when k takes smaller values. But as to SPP-PR, it returns *JTT*s, the smaller granularity results for users, which leads to the identification of the results satisfying different users' preferences becomes more difficult if k is too small. For instance, when k equals to 1, we need to return the optimal *JTT* for all users, but the accuracy rate of determining the optimal result is not high. When the value of k grows, the top-k results could have more possibility to contain the *JTT*s meeting the users' preferences; but this probability will decrease if the value of k continue to increase.

To verify the above analysis, we give an example that shows the results returned by SPP-PR on our employed DBLP data set in Table 4. For the second query (Q_2: *sigmod, xiaofang*) in Table 2, we illustrate the results returned by SPP-PR based on the *CN* "Author^{xiaofang} ⋈ Paper ⋈ Conference^{sigmod}". Since the information of "Author" and "Conference" has been given, we only show the selected papers in Table 4. When k=1, some participants deem that the result does not conform to their preferences, so the precision is not high. But this situation becomes better when the value of k increases, e.g., most participants view their preferred papers are contained in the results when k=4. However, as k continues to grow, the results also include some undesired ones to some participants. For instance, "Sampling dirty data for matching attributes" is a needless result to most participants when k is set to 7.

We also evaluate the impact of T_{max} on the precisions of three approaches in Fig. 5(d). The results show that the precisions of SPP and SPP-PR are obviously higher than that of DISCOVER, which also verifies



(a) NDCG by varying K

(b) NDCG by varying T_{max}

(c) Precision by varying K

(d) Precision by varying T_{max}

Fig. 5. Effectiveness comparison for DISCOVER, SPP, and SPP-PR.

Table 4
The results of the query Q_2.

	Q_2: sigmod, *xiaofang*
k = 1	Spark: top-k keyword query in relational databases
k = 4	Spark: top-k keyword query in relational databases **Searching trajectories by locations: an efficiency study** **Monitoring path nearest neighbor in road networks** **K-nearest neighbor search for fuzzy objects**
k = 7	Spark: top-k keyword query in relational databases **Searching trajectories by locations: an efficiency study** **Monitoring path nearest neighbor in road networks** **K-nearest neighbor search for fuzzy objects** *Towards Effective Indexing for Very Large Video Sequence Database.* *Sampling dirty data for matching attributes.* *Effective data co-reduction for multimedia similarity search.*

that the ranking effectiveness is actually enhanced when we incorporate the user preferences. Another case we observe is that the precisions of three approaches first increase slightly and then decrease slowly when the value of T_{max} grows. The reason caused this phenomenon is that some longer *CN*s will be included by the top-k ones and these longer *CN*s probably exactly conform to the preferences of participants; but when the value of T_{max} keeps increasing, more longer *CN*s will appear in the top-k ones but some of they are not the results meeting participants' requirements, which results in the precisions of these approaches take a downward trend.

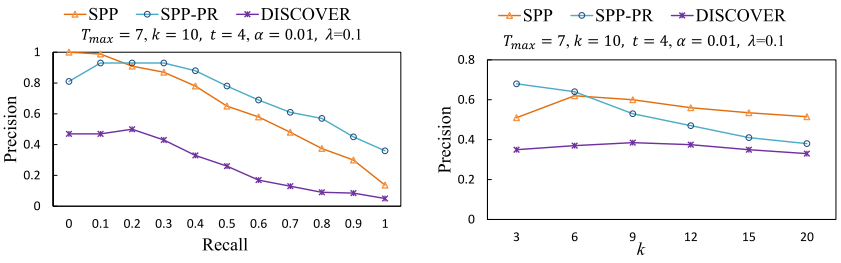
Overall ranking effectiveness comparison. Fig. 6(a) shows the 11-points precision/recall graph for DISCOVER, SPP, and SPP-PR, in which the precision generally goes down with recall growing. In the global perspective, SPP behaves well with points (0.1,0.988), …, (0.9,0.230), but the precision rate drops sharply with the recall rate increasing. As to SPP-PR, its precision/recall curve also presents a downward trend in general, but it declines more smoothly than the precision/recall curve of SPP. The same as the above evaluations, DISCOVER takes the worst performance. Meanwhile, as an auxiliary, Fig. 6(b) presents the F-measure value by varying k. The three preserve some differences as in the previous case. As the value of k increases, the precisions of SPP and SPP-PR both decline significantly but their recalls will become greater, which leads to the F-measure curves of SPP and SPP-PR go down more smoothly.

Search efficiency evaluation. Since our work involves the problems of detecting isomorphic graphs, identifying the best proper partition, and calculating final results (*JTT*s), so it is essential to evaluate the time costs of relevant algorithms. Due to the isomorphic graph matching is a basic step to the identification of the best proper partition, we thereby compare the performances of MatchGraph and Ullmann, which both can be used for detecting the isomorphic graphs in Fig. 7(a). Here, we first select five generated *CN*s and construct a set containing different numbers of LF-subtrees. For every *CN*, we make MatchGraph and Ullmann detect which LF-subtrees in this set are contained by the specified *CN*, and then keep the processing time for each *CN*. We then average the time costs of processing five *CN*s as the final result shown in Fig. 7(a). The observation displays that MatchGraph outperforms significantly than Ullmann regardless of the number of LF-subtrees to be detected, this is because MatchGraph only needs to detect whether a LF-subtree is isomorphic to a subgraph of a *CN* but Ullmann has to discover all isomorphic subgraphs to the LF-subtree from the *CN*. This evaluation testifies that MatchGraph can reduce a great deal of calculations.

In Fig. 7(b), we test the performance of LF-CN-M for identifying the best proper partition for a *CN* by varying T_{max}. Here, we randomly select three users labeled as "*User1*", "*User2*", and "*User3*", and then process one query submitted by every user. In this process, we census the time of identifying the best proper partition for every *CN* and then take the average value of these times for every user respectively. From Fig. 7(b), we find that T_{max} has an obvious impact on the performance of LF-CN-M, because the greater value of T_{max} will generate *CN*s with larger sizes, then more nodes need to be detected when identifying the best proper partition for these *CN*s.

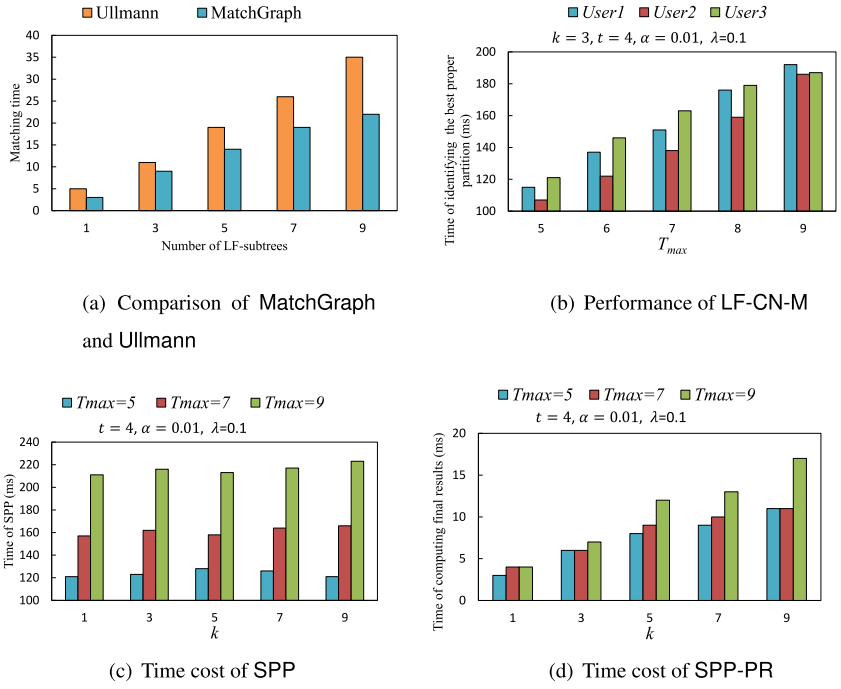
The search efficiency of SPP is evaluated in Fig. 7(c). The time cost of SPP includes the time of identifying the best proper partition for every *CN* as well as the time of ranking all *CN*s. When the value of T_{max} is fixed, we find that the time cost of SPP almost remains the same as k takes different values. The reason is that, regardless of the value of k, SPP always needs to identify the best proper partition for all *CN*s and rank them. However, the search efficiency of SPP is heavily affected by T_{max} because the average size of *CN*s will become greater when the value of T_{max} grows and the *CN*s with larger size need more time to be processed.

Finally, we test the efficiency of SPP-PR in Fig. 7(d), where the processing time refers to the time of generating *JTT*s based on *CN*s,



(a) 11-pt Precision/Recall (b) F-measure

Fig. 6. Overall effectiveness comparison.



(a) Comparison of MatchGraph and Ullmann (b) Performance of LF-CN-M

(c) Time cost of SPP (d) Time cost of SPP-PR

Fig. 7. Search efficiency evaluation.

calculating PageRank scores of *JTT*s, and determining the top-k results. The tests show that the cost of SPP-PR almost increases linearly when k grows. The reason is that more *CN*s will be selected by SPP as k takes greater values, which will cause more *JTT*s to be generated and ranked, hence the search time of SPP-PR is notable on the rise. When k is fixed, we observe that the consumed time of SPP-PR is also slightly influenced by T_{max}.

6. Conclusion and future work

Existing work on keyword search in databases has considered the problem of improving the search effectiveness extensively. However, few works have explicitly taken user preferences into the consideration when ranking query results. Additionally, they usually separate the usages of the schema graph and the data graph when processing keyword queries on databases. In this paper, by introducing user feedback to the problem of ranking *CN*s, we propose a new ranking strategy to adapt to user preferences. In this new ranking strategy, we come across a NP-hard problem and provide a complete solution. Based on the selected top-k *CN*s, we further designed a data graph based algorithm to improve the ranking effectiveness of *JTT*s with the PageRank principle. Finally, we evaluated the proposed approaches on the DBLP dataset via a user study, which verifies the effectiveness of our strategy.

In the future, beyond adopting the frequent subtrees to embody user preferences, we will study the problem of mining more valuable information from user query logs, and meanwhile optimize the ranking strategy based on the mined information. Additionally, user query logs usually keep growing, so we also plan to explore an incremental score function that can update the scores of different *CN*s when the scale of user query logs gradually increases.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 61702217), the Primary Research and Development Plan of Shandong Province (No. 2017GGX10144, No. 2018GGX101048, No. 2017CXGC0701), National Natural Science Foundation of China (No. 61873324, No. 61772231, No. 61771230), the Shandong Provincial Natural Science Foundation (No. ZR2017MF025), the Project of Shandong Provincial Social Science Program (No. 18CHLJ39), and the Project of Shandong Province Higher Educational Science and Technology Program (No. J16LN07).

References

Agrawal, S., Chaudhuri, S., Das, G., 2002. DBXplorer: A system for keyword-based search over relational databases. In: ICDE.

Alavi, F., Hashemi, S., 2015. DFP-SEPSF: A dynamic frequent pattern tree to mine strong emerging patterns in streamwise features. Eng. Appl. Artif. Intell. 37, 54–70.

Bergamaschi, S., Guerra, F., Interlandi, M., Trillo-Lado, R., Velegrakis, Y., 2016. Combining user and database perspective for solving keyword queries over relational databases. Inform. Syst. 55, 1–19.

Chi, Y., Yang, Y., Muntz, R., 2003. Indexing and mining frequent subtrees. In: ICDE.
Coffman, J., Weaver, A.C., 2014. An empirical performance evaluation of relational keyword search techniques. IEEE Trans. Knowl. Data Eng. 26 (1), 30–42.
Duong, H., Truong, T., Le, B., 2018. Efficient algorithms for simultaneously mining concise representations of sequential patterns based on extended pruning conditions. Eng. Appl. Artif. Intell. 67, 197–210.
Fakas, G., Cai, Z., Mamoulis, N., 2015. Diverse and proportional size-l object summaries for keyword search. In: SIGMOD. ACM, pp. 363–375.
Gan, W., Lin, J.C.-W., Fournier-Viger, P., Chao, H.-C., Zhan, J., 2017. Mining of frequent patterns with multiple minimum supports. Eng. Appl. Artif. Intell. 60, 83–96.
Gao, L., Yu, X., Liu, Y., 2011. Keyword query cleaning with query logs. In: WAIM. pp. 31–42.
He, H., Wang, H., Yang, J., Yu, P.S., 2007. BLINKS: Ranked keyword searches on graphs. In: SIGMOD. pp. 305–316.
Hristidis, V., Gravano, L., Papakonstantinou, Y., 2003. Efficient IR-style keyword search over relational databases. In: VLDB. pp. 850–861.
Hristidis, V., Papakonstantinou, Y., 2002. DISCOVER: Keyword search in relational databases. In: VLDB.
Hulgeri, A., Nakhe, C., 2002. Keyword searching and browsing in databases using banks. In: ICDE.
Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H., 2005. Bidirectional expansion for keyword search on graph databases. In: VLDB. pp. 505–516.
Lin, J.C.-W., Ren, S., Fournier-Viger, P., Pan, J.-S., Hong, T.-P., 2018. Efficiently updating the discovered high average-utility itemsets with transaction insertion. Eng. Appl. Artif. Intell. 72, 136–149.
Lin, Z., Xue, Q., Lai, Y., 2016. PACOKS: Progressive ant-colony-optimization-based keyword search over relational databases. In: WAIM. Springer, pp. 107–119.
Liu, Z.-Y., Qiao, H., 2014. GNCCP-Graduated nonconvexity and concavity procedure. IEEE Trans. Pattern Anal. Mach. Intell. 36 (6), 1258–1267.
Liu, F., Yu, C., Meng, W., Chowdhury, A., 2006. Effective keyword search in relational databases. In: SIGMOD. pp. 563–574.
Luo, Y., Lin, X., Wang, W., Zhou, X., 2007. SPARK: Top-k keyword query in relational databases. In: SIGMOD. pp. 115–126.
Luo, Y., Wang, W., Lin, X., Zhou, X., Wang, J., Li, K., 2011. Spark2: Top-k keyword query in relational databases. TKDE 23 (12), 1763–1780.
Peng, Z., Zhang, J., Wang, S., Wang, C., 2009. Bring user feedback into keyword search over databases. In: Proceeding of the 3rd Workshop on Electronic Government Technology and Application, pp. 210–214.
Qiao, S., Han, N., Zhou, J., Li, R.-H., Jin, C., Gutierrez, L.A., 2018. SocialMix: A familiarity-based and preference-aware location suggestion approach. Eng. Appl. Artif. Intell. 68, 192–204.
Yagci, A.M., Aytekin, T., Gurgen, F.S., 2017. Scalable and adaptive collaborative filtering by mining frequent item co-occurrences in a user feedback stream. Eng. Appl. Artif. Intell. 58, 171–184.
Yang, M., Ding, B., Chaudhuri, S., Chakrabarti, K., 2014. Finding patterns in a knowledge base using keywords to compose table answers. Proc. VLDB Endowment 7 (14), 1809–1820.
Yu, X., Shi, H., 2012. CI-Rank: Ranking keyword search results based on collective importance. In: ICDE.
Zeng, Z., Bao, Z., Ling, T.W., Lee, M.L., 2012. Isearch: An interpretation based framework for keyword search in relational databases. In: KEYS. pp. 3–10.
Zhou, J., Liu, Y., Yu, Z., 2015. Improving the effectiveness of keyword search in databases using query logs. In: WAIM. Springer, pp. 193–206.