Best Keyword Cover Search
Ke Deng, Xin Li, Jiaheng Lu, and Xiaofang Zhou

Abstract—It is common that the objects in a spatial database (e.g., restaurants/hotels) are associated with keyword(s) to indicate their businesses/services/features. An interesting problem known as Closest Keywords search is to query objects, called keyword cover, which together cover a set of query keywords and have the minimum inter-objects distance. In recent years, we observe the increasing availability and importance of keyword rating in object evaluation for the better decision making. This motivates us to investigate a generic version of Closest Keywords search called Best Keyword Cover which considers inter-objects distance as well as the keyword rating of objects. The baseline algorithm is inspired by the methods of Closest Keywords search which is based on exhaustively combining objects from different query keywords to generate candidate keyword covers. When the number of query keywords increases, the performance of the baseline algorithm drops dramatically as a result of massive candidate keyword covers generated. To attack this drawback, this work proposes a much more scalable algorithm called keyword nearest neighbor expansion (keyword-NNE). Compared to the baseline algorithm, keyword-NNE algorithm significantly reduces the number of candidate keyword covers generated. The in-depth analysis and extensive experiments on real data sets have justified the superiority of our keyword-NNE algorithm.

Index Terms—Spatial database, Point of Interests, Keywords, Keyword Rating, Keyword Cover

1 INTRODUCTION

Driven by mobile computing, location-based services and wide availability of extensive digital maps and satellite imagery (e.g., Google Maps and Microsoft Virtual Earth services), the spatial keywords search problem has attracted much attention recently [3, 4, 5, 6, 8, 10, 15, 16, 18].

In a spatial database, each tuple represents a spatial object which is associated with keyword(s) to indicate the information such as its businesses/services/features. Given a set of query keywords, an essential task of spatial keywords search is to identify spatial object(s) which are associated with keywords relevant to a set of query keywords, and have desirable spatial relationships (e.g., close to each other and/or close to a query location). This problem has unique value in various applications because users’ requirements are often expressed as multiple keywords. For example, a tourist who plans to visit a city may have particular shopping, dining and accommodation needs. It is desirable that all these needs can be satisfied without long distance traveling.

Due to the remarkable value in practice, several variants of spatial keyword search problem have been studied. The works [5, 6, 8, 15] aim to find a number of individual objects, each of which is close to a query location and the associated keywords (or called document) are very relevant to a set of query keywords (or called query document). The document similarity (e.g., [14]) is applied to measure the relevance between the two sets of keywords. Since it is likely none of individual objects is associated with all query keywords, this motivates the studies [4, 17, 18] to retrieve multiple objects, called keyword cover, which together cover (i.e., associated with) all query keywords and are close to each other. This problem is known as m Closest Keywords (mCK) query in [17, 18]. The problem studied in [4] additionally requires the retrieved objects close to a query location.

This paper investigates a generic version of mCK query, called Best Keyword Cover (BKC) query, which considers inter-objects distance as well as keyword rating. It is motivated by the observation of increasing availability and importance of keyword rating in decision making. Millions of businesses/services/features around the world have been rated by customers through online business review sites such as Yelp, Citysearch, ZAGAT and Dianping, etc. For example, a restaurant is rated 65 out of 100 (ZAGAT.com) and a hotel is rated 3.9 out of 5 (hotels.com). According to a survey in 2013 conducted by Dimensional Research 1, www.dimensionalresearch.com

\[ \text{BKC returns } \{t_1, s_1, c_1\} \quad \text{mCK returns } \{t_2, s_2, c_2\} \]

Fig. 1. BKC vs. mCK

1. www.dimensionalresearch.com

1041-4347 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
an overwhelming 90 percent of respondents claimed that buying decisions are influenced by online business review/rating. Due to the consideration of keyword rating, the solution of BKC query can be very different from that of mCK query. Figure 1 shows an example. Suppose the query keywords are “Hotel”, “Restaurant” and “Bar”. mCK query returns \{t_2, s_2, c_2\} since it considers the distance between the returned objects only. BKC query returns \{t_1, s_1, c_1\} since the keyword ratings of object are considered in addition to the inter-objects distance. Compared to mCK query, BKC query supports more robust object evaluation and thus underpins the better decision making.

This work develops two BKC query processing algorithms, baseline and keyword-NNE. The baseline algorithm is inspired by the mCK query processing methods [17, 18]. Both the baseline algorithm and keyword-NNE algorithm are supported by indexing the objects with an R*-tree like index, called KRR*-tree. In the baseline algorithm, the idea is to combine nodes in higher hierarchical levels of KRR*-trees to generate candidate keyword covers. Then, the most promising candidate is assessed in priority by combining their child nodes to generate new candidates. Even though BKC query can be effectively resolved, when the number of query keywords increases, the performance drops dramatically as a result of massive candidate keyword covers generated.

To overcome this critical drawback, we developed much scalable keyword nearest neighbor expansion (keyword-NNE) algorithm which applies a different strategy. Keyword-NNE selects one query keyword as principal query keyword. The objects associated with the principal query keyword are principal objects. For each principal object, the local best solution (known as local best keyword cover (lbkc)) is computed. Among them, the lbkc with the highest evaluation is the solution of BKC query. Given a principal object, its lbkc can be identified by simply retrieving a few nearby and highly rated objects in each non-principal query keyword (2-4 objects in average as illustrated in experiments). Compared to the baseline algorithm, the number of candidate keyword covers generated in keyword-NNE algorithm is significantly reduced. The in-depth analysis reveals that the number of candidate keyword covers further processed in keyword-NNE algorithm is optimal, and each keyword candidate cover processing generates much less new candidate keyword covers than that in the baseline algorithm.

The remainder of this paper is organized as follows. The problem is formally defined in section 2 and the related work is reviewed in section 3. After that, section 4 discusses keyword rating R*-tree (KRR*-tree). The baseline algorithm is introduced in section 5 and keyword-NNE algorithm is proposed in section 6. Section 7 discusses the situation of weighted average of keyword ratings. An in-depth analysis is given in section 8. Then section 9 reports the experimental results. The paper is concluded in section 10.

2 PRELIMINARY

Given a spatial database, each object may be associated with one or multiple keywords. Without loss of generality, the object with multiple keywords are transformed to multiple objects located at the same location, each with a distinct single keyword. So, an object is in the form \(\langle id, x, y, \text{keyword}, \text{rating} \rangle\) where \(x, y\) define the location of the object in a 2-dimensional geographical space. No data quality problem such as misspelling exists in keywords.

Definition 1 (Diameter): Let \(O\) be a set of objects \(\{o_1, \cdots, o_n\}\). For \(o_i, o_j \in O\), \(\text{dist}(o_i, o_j)\) is the Euclidean distance between \(o_i, o_j\) in the 2-dimensional geographical space. The diameter of \(O\) is:

\[
diam(O) = \max_{o_i, o_j \in O} \text{dist}(o_i, o_j).\]  

The score of \(O\) is a function with respect to not only the diameter of \(O\) but also the keyword rating of objects in \(O\). Users may have different interests in keyword rating of objects. We first discuss the situation that a user expects to maximize the minimum keyword rating of objects in BKC query. Then we will discuss another situation in section 7 that a user expects to maximize the weighted average of keyword ratings.

The linear interpolation function [5, 16] is used to obtain the score of \(O\) such that the score is a linear interpolation of the individually normalized diameter and the minimum keyword rating of \(O\).

\[
O.\text{score} = \text{score}(A, B) = \alpha(1 - \frac{A}{\max_{o \in O} \text{dist}}) + (1 - \alpha) \frac{B}{\max_{o \in O} \text{rating}}.
\]  

(2)

where \(B\) is the minimum keyword rating of objects in \(O\) and \(\alpha (0 \leq \alpha \leq 1)\) is an application specific parameter. If \(\alpha = 1\), the score of \(O\) is solely determined by the diameter of \(O\). In this case, BKC query is degraded to mCK query.

If \(\alpha = 0\), the score of \(O\) only considers the minimum keyword rating of objects in \(O\) where \(\max_{o \in O} \text{dist}\) and \(\max_{o \in O} \text{rating}\) are used to normalize diameter and keyword rating into [0,1] respectively. \(\max_{o \in O} \text{dist}\) is the maximum distance between any two objects in the spatial database \(D\), and \(\max_{o \in O} \text{rating}\) is the maximum keyword rating of objects.

Lemma 1: The score is of monotone property.

Proof: Given a set of objects \(O_i\), suppose \(O_j\) is a subset of \(O_i\). The diameter of \(O_i\) must be not less than that of \(O_j\), and the minimum keyword rating of objects in \(O_i\) must be not greater than that of objects in \(O_j\). Therefore, \(O_i.\text{score} \leq O_j.\text{score}\). \(\square\)

Definition 2 (Keyword Cover): Let \(T\) be a set of keywords \(\{k_1, \cdots, k_n\}\) and \(O\) a set of objects \(\{o_1, \cdots, o_n\}\),
O is a keyword cover of T if one object in O is associated with one and only one keyword in T.

**Definition 3 (Best Keyword Cover Query (BKC)):**

Given a spatial database D and a set of query keywords T, BKC query returns a keyword cover O of T (O ⊂ D) such that \( \text{O.score} \geq \text{O'.score} \) for any keyword cover O' of T (O' ⊂ D).

The notations used in this work are summarized in Table 1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A spatial database.</td>
</tr>
<tr>
<td>T</td>
<td>A set of query keywords.</td>
</tr>
<tr>
<td>( O_k )</td>
<td>The set of objects associated with keyword k.</td>
</tr>
<tr>
<td>( o_k )</td>
<td>An object in ( O_k ).</td>
</tr>
<tr>
<td>( K_{C_k} )</td>
<td>The set of keyword covers in each of which ( o_k ) is a member.</td>
</tr>
<tr>
<td>( k_{rc} )</td>
<td>A keyword cover in ( K_{C_k} ).</td>
</tr>
<tr>
<td>( lbk_{rc} )</td>
<td>The local best keyword cover of ( o_k ), i.e., the keyword cover in ( K_{C_k} ) with the highest score.</td>
</tr>
<tr>
<td>( o_k.N_{N_k} )</td>
<td>( o_k )'s n\textsuperscript{th} keyword nearest neighbor in query keyword ( k_1 ).</td>
</tr>
<tr>
<td>KRR\textsuperscript{(*)}-tree</td>
<td>The keyword rating R*-tree of ( O_k ).</td>
</tr>
<tr>
<td>( N_k )</td>
<td>A node of KRR\textsuperscript{(*)}-tree.</td>
</tr>
</tbody>
</table>

### 3 RELATED WORK

#### Spatial Keyword Search

Recently, the spatial keyword search has received considerable attention from research community. Some existing works focus on retrieving individual objects by specifying a query consisting of a query location and a set of query keywords (or known as document in some context). Each retrieved object is associated with keywords relevant to the query keywords and is close to the query location [3, 5, 6, 8, 10, 15, 16]. The similarity between documents (e.g., [14]) are applied to measure the relevance between two sets of keywords.

Since it is likely no individual object is associated with all query keywords, some other works aim to retrieve multiple objects which together cover all query keywords [4, 17, 18]. While potentially a large number of object combinations satisfy this requirement, the research problem is that the retrieved objects must have desirable spatial relationship. In [4], authors put forward the problem to retrieve objects which 1) cover all query keywords, 2) have minimum inter-objects distance and 3) are close to a query location. The work [17, 18] study a similar problem called m Closet Keywords (mCK). mCK aims to find objects which cover all query keywords and have the minimum inter-objects distance. Since no query location is asked in mCK, the search space in mCK is not constrained by the query location. The problem studied in this paper is a generic version of mCK query by also considering keyword rating of objects.

#### Access Methods

The approaches proposed by Cong et al. [5] and Li et al. [10] employ a hybrid index that augments nodes in non-leaf nodes of an R/R*-tree with inverted indexes. The inverted index at each node refers to a pseudo-document that represents the keywords under the node. Therefore, in order to verify if a node is relevant to a set of query keywords, the inverted index is accessed at each node to evaluate the matching between the query keywords and the pseudo-document associated with the node.

In [18], bR*-tree was proposed where a bitmap is kept for each node instead of pseudo-document. Each bit corresponds to a keyword. If a bit is “1”, it indicates some object(s) under the node is associated with the corresponding keyword; “0” otherwise. A bR*-tree example is shown in Figure 2(a) where a non-leaf node \( N \) has four child nodes \( N_1, N_2, N_3, N_4 \). The bitmaps of \( N_1, N_2, N_3 \) are 111 and the bitmap of \( N_3 \) is 101. In specific, the bitmap 101 indicates some objects under \( N_3 \) are associated with keyword “hotel” and “restaurant” respectively, and no object under \( N_3 \) is associated with keyword “bar”. The bitmap allows to combine nodes to generate candidate keyword covers. If a node contains all query keywords, this node is a candidate keyword cover. If multiple nodes together cover all query keywords, they constitute a candidate keyword cover. Suppose the query keywords are 111. When \( N \) is visited, its child node \( N_1, N_2, N_3, N_4 \) are processed. \( N_1, N_2, N_4 \) are associated with all query keywords and \( N_3 \) is associated with two query keywords. The candidate keyword covers generated are \{\( N_1 \), \( N_2 \), \( N_3 \), \( N_1, N_2 \), \( N_1, N_3 \), \( N_2, N_3 \), \( N_3, N_4 \), \( N_1, N_2, N_3 \), \( N_1, N_3, N_4 \) and \( N_2, N_3, N_4 \). Among the candidate keyword covers, the one with the best evaluation is processed by combining their child nodes to generate more candidates. However, the number of candidates generated can be very large. Thus, the depth-first bR*-tree browsing strategy is applied in order to access the objects in leaf nodes as soon as possible. The purpose is to obtain the current best solution as soon as possible. The current best solution is used to prune the candidate keyword covers. In the same way, the remaining candidates are processed and the current best solution is updated once a better solution is identified. When all candidates have been pruned, the current best solution is returned to mCK query.

In [17], a virtual bR*-tree based method is introduced to handle mCK query with attempt to handle data set with massive number of keywords. Compared to the method in [18], a different index structure is utilized. In virtual bR*-tree based method, an R*-tree is used to index locations of objects and an inverted index is used to label the leaf nodes in the R*-tree associated with each keyword. Since only leaf nodes have keyword information the mCK query is processed by browsing index bottom-up. At first, m inverted lists corresponding to the query keywords are retrieved, and fetch all objects from the same leaf node to construct a virtual node in memory. Clearly, it has a counterpart in the original R*-tree. Each time a virtual node is constructed, it will be treated as a subtree which is browsed in the same way as in [18]. Compared to bR*-tree, the number of nodes in R*-tree has been greatly reduced such that the I/O cost is saved.
Fig. 2. (a) A bR*-tree. (b) The KRR*-tree for keyword “restaurant”.

As opposed to employing a single R*-tree embedded with keyword information, multiple R*-trees have been used to process multiway spatial join (MWSJ) which involves data of different keywords (or types). Given a number of R*-trees, one for each keyword, the MWSJ technique of Papadias et al. [13] (later extended by Mamoulis and Papadias [11]) uses the synchronous R*-tree approach [2] and the window reduction (WR) approach [12]. Given two R*-tree indexed relations, SRT performs two-way spatial join via synchronous traversal of the two R*-trees based on the property that if two intermediate R*-tree nodes do not satisfy the spatial join predicate, then the MBRs below them will not satisfy the spatial predicate either. WR uses window queries to identify spatial regions which may contribute to MWSJ results.

4 INDEXING KEYWORD RATINGS

To process BKCl query, we augment R*-tree with one additional dimension to index keyword ratings. Keyword rating dimension and spatial dimension are inherently different measures with different ranges. It is necessary to make adjustment. In this work, a 3-dimensional R*-tree called keyword rating R*-tree (KRR*-tree) is used. The ranges of both spatial and keyword rating dimensions are normalized into [0,1]. Suppose we need construct a KRR*-tree over a set of objects D. Each object \( o \in D \) is mapped into a new space using the following mapping function:

\[
f : o(x, y, rating) \rightarrow o\left(\frac{x}{\max_x}, \frac{y}{\max_y}, \frac{\text{rating}}{\max_{\text{rating}}}\right).
\]

where \( \max_x, \max_y, \max_{\text{rating}} \) are the maximum value of objects in \( D \) on \( x \), \( y \) and keyword rating dimensions respectively. In the new space, KRR*-tree can be constructed in the same way as constructing a conventional 3-dimensional R*-tree. Each node \( N \) in KRR*-tree is defined as \( N(x, y, r, l_x, l_y, l_r) \) where \( x \) is the value of \( N \) in \( x \) axe close to the origin, i.e., \((0,0,0,0,0,0)\), and \( l_x \) is the width of \( N \) in \( x \) axe, so do \( y \), \( l_y \) and \( r \), \( l_r \). The Figure 2 (b) gives an example to illustrate the nodes of KRR*-tree indexing the objects in keyword “restaurant”.

In [17, 18], a single tree structure is used to index objects of different keywords. In the similar way as discussed above, the single tree can be extended with an additional dimension to index keyword rating. A single tree structure suits the situation that most keywords are query keywords. For the above mentioned example, all keywords, i.e., “hotel”, “restaurant” and “bar”, are query keywords. However, it is more frequent that only a small fraction of keywords are query keywords. For example in the experiments, only less than 5% keywords are query keywords. In this situation, a single tree is poor to approximate the spatial relationship between objects of few specific keywords. Therefore, multiple KRR*-trees are used in this work, each for one keyword. The KRR*-tree for keyword \( k_i \) is denoted as KRR*-tree \( k_i \).

Given an object, the rating of an associated keyword is typically the mean of ratings given by a number of customers for a period of time. The change does happen but slowly. Even though dramatic change occurs, the KRR*-tree is updated in the standard way of R*-tree update.

5 BASELINE ALGORITHM

The baseline algorithm is inspired by the mCK query processing methods [17, 18]. For mCK query processing, the method in [18] browses index in top-down manner while the method in [17] does bottom-up. Given the same hierarchical index structure, the top-down browsing manner typically performs better than the bottom-up since the search in lower hierarchical levels is always guided by the search result in the higher hierarchical levels. However, the significant advantage of the method in [17] over the method in [18] has been reported. This is because of the different index structures applied. Both of them use a single tree structure to index data objects of different keywords. But the number of nodes of the index in [17] has been greatly reduced to save I/O cost by keeping keyword information with inverted index separately. Since only leaf nodes and their keyword information are maintained in the inverted index, the bottom-up index browsing manner is used. When designing the baseline algorithm for BKCl query processing, we take the advantages of both methods [17, 18]. First, we apply multiple KRR*-trees which contain no keyword information in inverted index separately. Since only leaf nodes and their keyword information are maintained in the inverted index, the number of nodes of the index is not more than that of the index in [17]; second, the top-down index browsing method can be applied since each keyword has own index.

Suppose KRR*-trees, each for one keyword, have been constructed. Given a set of query keywords \( T = \{k_1, \ldots, k_n\} \), the child nodes of the root of KRR*-tree \( k_i \) (\( i \leq i \leq n \)) are retrieved and they are combined to generate candidate keyword covers. Given a candidate keyword cover \( O = \{N_{k_1}, \ldots, N_{km}\} \) where \( N_{ki} \) is a node of
KRR*_{k_i}-tree.

\[ O.score = score(A, B). \]
\[ A = \max_{N_i, N_j \in O} dist(N_i, N_j) \]
\[ B = \min_{N \in O}(N.maxrating). \]

Where N.maxrating is the maximum value of objects under N in keyword rating dimension; \( dist(N_i, N_j) \) is the minimum Euclidean distance between \( N_i \) and \( N_j \) in the 2-dimensional geographical space defined by \( x \) and \( y \) dimensions.

**Lemma 2:** Given two keyword covers \( O \) and \( O' \), \( O' \) consists of objects \( \{o_{k_1}, ..., o_{k_n}\} \) and \( O \) consists of nodes \( \{N_{k_1}, ..., N_{k_n}\} \). If \( o_{k_i} \) is under \( N_{k_i} \) in KRR*_{k_i}-tree for \( 1 \leq i \leq n \), it is true that \( O'.score \leq O.score \).

**Algorithm 1:** Baseline(\( T, Root \))

**Input:** A set of query keywords \( T \), the root nodes of all KRR*-trees \( Root \).

**Output:** Best Keyword Cover.

1. \( bkc \leftarrow \emptyset; \)
2. \( H \leftarrow \text{Generate_Candidate}(T, \text{Root}, bkc); \)
3. while \( H \) is not empty do
   4. \( can \leftarrow \) the candidate in \( H \) with the highest score;
   5. Remove \( can \) from \( H \);
   6. Depth_First_Tree_Browsing(\( H, T, can, bkc \);
   7. foreach \( candidate \in H \) do
      8. if \( \text{candidate.score} \leq \text{bkc.score} \) then
         9. remove \( candidate \) from \( H \);
5. return \( bkc \);

**Algorithm 2:** Depth_First_Tree_Browsing(\( H, T, can, bkc \))

**Input:** A set of query keywords \( T \), a candidate \( can \), the candidate set \( H \), and the current best solution \( bkc \).

1. if \( can \) consists of leaf nodes then
   2. \( S \leftarrow \) objects in \( can \);
   3. \( bkc' \leftarrow \) the keyword cover with the highest score identified in \( S \);
   4. if \( \text{bkc.score} < \text{bkc'.score} \) then
      5. \( bkc \leftarrow bkc' \);
   6. else
      7. \( \text{New.Cans} \leftarrow \text{Generate_Candidate}(T, can, bkc) \);
      8. Replace \( can \) by \( \text{New.Cans} \) in \( H \);
      9. \( can \leftarrow \) the candidate in \( \text{New.Cans} \) with the highest score;
   10. Depth_First_Tree_Browsing(\( H, T, can, bkc \);

Algorithm 1 shows the pseudo-code of the baseline algorithm. Given a set of query keywords \( T \), it first generates candidate keyword covers using Generate_Candidate function which combines the child nodes of the roots of KRR*_{k_i}-trees for all \( k_i \in T \) (line 2). These candidates are maintained in a heap \( H \). Then, the candidate with the highest score in \( H \) is selected and its child nodes are combined using Generate_Candidate function to generate more candidates. Since the number of candidates can be very large, the depth-first KRR*_{k_i}-tree browsing strategy is applied to access the leaf nodes as soon as possible (line 6). The first candidate consisting of objects (not nodes of KRR*-tree) is the current best solution, denoted as \( bkc \), which is an intermediate solution. According to Lemma 2, the candidates in \( H \) are pruned if they have score less than \( \text{bkc.score} \) (line 8). The remaining candidates are processed in the same way and \( bkc \) is updated if the better intermediate solution is found. Once no candidate is remained in \( H \), the algorithm terminates by returning current \( bkc \) to BKC query.

In Generate_Candidate function, it is unnecessary to actually generate all possible keyword covers of input nodes (or objects). In practice, the keyword covers are generated by incrementally combining individual nodes (or objects). An example in Figure 3 shows all possible combinations of input nodes incrementally generated bottom up. There are three keywords \( k_1, k_2 \) and \( k_3 \) and each keyword has two nodes. Due to the monotonic property in Lemma 1, the idea of Apriori algorithm [1] can be applied. Initially, each node is a combination with score = \( \infty \). The combination with the highest score is always processed in priority to combine one more input node in order to cover a keyword, which is not covered yet. If a combination has score less than \( \text{bkc.score} \), any superset of it must have score less than \( \text{bkc.score} \). Thus, it is unnecessary to generate the superset. For example, if \( \{N_{k_2}, N_{k_3}\}.score < \text{bkc.score} \), any superset of \( \{N_{k_2}, N_{k_3}\} \) must have score less than \( \text{bkc.score} \). So, it is not necessary to generate \( \{N_{k_2}, N_{k_3}, N_{k_1}\} \) and \( \{N_{k_2}, N_{k_2}, N_{k_2}\} \).

6 KEYWORD NEAREST NEIGHBOR EXPANSION (KEYWORD-NNE)

Using the baseline algorithm, BKC query can be effectively resolved. However, it is based on exhaustively
combining objects (or their MBRs). Even though pruning techniques have been explored, it has been observed that the performance drops dramatically, when the number of query keywords increases, because of the fast increase of candidate keyword covers generated. This motivates us to develop a different algorithm called keyword nearest neighbor expansion (keyword-NNE).

We focus on a particular query keyword, called principal query keyword. The objects associated with the principal query keyword are called principal objects. Let \( k \) be the principal query keyword. The set of principle objects is denoted as \( O_k \).

Definition 4 (Local Best Keyword Cover): Given a set of query keywords \( T \) and the principal query keyword \( k \in T \), the local best keyword cover of a principal object \( o_k \) is

\[
\text{lbkc}_{o_k} = \{ k | k \in K \text{C}_{o_k} \}
\]

\[
\text{lbkc}_{o_k}.score = \max_{k \in K \text{C}_{o_k}} k \text{c}.score \).
\]

Where \( K \text{C}_{o_k} \) is the set of keyword covers in each of which the principal object \( o_k \) is a member.

For each principal object \( o_k \in O_k \), \( \text{lbkc}_{o_k} \) is identified. Among all principal objects, the \( \text{lbkc}_{o_k} \) with the highest score is called global best keyword cover (\( GBKC_k \)).

Lemma 3: GBKC_k is the solution of BKC query.

Proof: Assume the solution of BKC query is a keyword cover \( k \) other than GBKC_k, i.e., \( k \text{c}.score > GBKC_k.score \). Let \( o_k \) be the principal object in \( k \). By definition, \( lbkc_{o_k}.score \geq k \text{c}.score \), and \( GBKC_k.score \geq lbkc_{o_k}.score \). So, \( GBKC_k.score \geq k \text{c}.score \) must be true. This conflicts to the assumption that \( BKC \) is a keyword cover \( k \) other than \( GBKC_k \).

The sketch of keyword-NNE algorithm is as follows:

**Sketch of Keyword-NNE Algorithm**

1. One query keyword \( k \in T \) is selected as the principal query keyword;
2. For each principal object \( o_k \in O_k \), \( \text{lbkc}_{o_k} \) is computed;
3. In \( O_k \), \( GBKC_k \) is identified;
4. return \( GBKC_k \).

Conceptually, any query keyword can be selected as the principal query keyword. Since computing \( lbkc \) is required for each principal object, the query keyword with the minimum number of objects is selected as the principal query keyword in order to achieve high performance.

### 6.1 LBKC Computation

Given a principal object \( o_k \), \( lbkc_{o_k} \) consists of \( o_k \) and the objects in each non-principal query keyword which is close to \( o_k \) and have high keyword ratings. It motivates us to compute \( lbkc_{o_k} \) by incrementally retrieving the keyword nearest neighbors of \( o_k \). 

#### 6.1.1 Keyword Nearest Neighbor

**Definition 5 (Keyword Nearest Neighbor (Keyword-NN)):**

Given a set of query keywords \( T \), the principal query keyword is \( k \in T \) and a non-principal query keyword is \( k_i \in T / \{ k \} \). \( O_{k_i} \) is the set of principal objects and \( O_{k_i} \) is the set of objects of keyword \( k_i \). The keyword nearest neighbor of a principal object \( o_k \in O_k \) in keyword \( k_i \) is \( o_{k_i} \in O_{k_i} \) iff \( \{ o_k, o_{k_i} \}.score \geq \{ o_k, o'_{k_i} \}.score \) for all \( o'_{k_i} \in O_{k_i} \).

The first keyword-NN of \( o_k \) in keyword \( k_i \) is denoted as \( o_k.nn_1 \) and the second keyword-NN is \( o_k.nn_2 \), and so on. These keyword-NNs can be retrieved by browsing \( KRR^{*}_{k_i} \)-tree. Let \( N_{k_i} \) be a node in \( KRR^{*}_{k_i} \)-tree.

\[
\{ o_k, N_{k_i} \}.score = \text{score}(A, B),
\]

\[
A = \text{dist}(o_k, N_{k_i}),
\]

\[
B = \min(N_{k_i}.maxrating, o_k.rating).
\]

where \( \text{dist}(o_k, N_{k_i}) \) is the minimum distance between \( o_k \) and \( N_{k_i} \) in the 2-dimensional geographical space defined by \( x \) and \( y \) dimensions, and \( N_{k_i}.maxrating \) is the maximum value of \( N_{k_i} \) in keyword rating dimension.

**Lemma 4:** For any object \( o_{k_i} \) under node \( N_{k_i} \) in \( KRR^{*}_{k_i} \)-tree,

\[
\{ o_k, N_{k_i} \}.score \geq \{ o_k, o_{k_i} \}.score.
\]

**Proof:** It is a special case of Lemma 2.

To retrieve keyword-NNs of a principal object \( o_k \) in keyword \( k_i \), \( KRR^{*}_{k_i} \)-tree is browsed in the best-first strategy [9]. The root node of \( KRR^{*}_{k_i} \)-tree is visited first by keeping its child nodes in a heap \( H \). For each node \( N_{k_i} \in H \), \( \{ o_k, N_{k_i} \}.score \) is computed. The node in \( H \) with the highest score is replaced by its child nodes. This operation is repeated until an object \( o_{k_i} \) (not a \( KRR^{*}_{k_i} \)-tree node) is visited. \( \{ o_k, o_{k_i} \}.score \) is denoted as current_best and \( o_k \) is the current best object. According to Lemma 4, any node \( N_{k_i} \in H \) is pruned if \( \{ o_k, N_{k_i} \}.score \leq \text{current_best} \). When \( H \) is empty, the current best object is \( o_k.nn_1 \) and \( j > 1 \) can be identified.

#### 6.1.2 lbkc Computing Algorithm

Computing \( lbkc_{o_k} \) is to incrementally retrieve keyword-NNs of \( o_k \) in each non-principal query keyword. An example is shown in Figure 4 where query keywords

![Fig. 3. Generating candidates.](image-url)
are “bar”, “restaurant” and “hotel”. The principal query keyword is “bar”. Suppose we are computing lbkc3. The first keyword-NN of t3 in “restaurant” and “hotel” are c2 and s3 respectively. A set S is used to keep t3, s3, c2. Let kc be the keyword cover in S which has the highest score (the idea of Apriori algorithm can be used, see section 5). After step 1, kc.score = 0.3. In step 2, “hotel” is selected and the second keyword-NN of t3 in “hotel” is retrieved, i.e., s2. Since \{t3, s2\}.score < kc.score, s2 can be pruned and more importantly all objects not accessed in “hotel” can be pruned according to Lemma 5. In step 3, the second keyword-NN of t3 in “restaurant” is retrieved, i.e., c3. Since \{t3, c3\}.score > kc.score, c3 is inserted into S. As a result, kc is updated to 0.4. Then, the third keyword-NN of t3 in “restaurant” is retrieved, i.e., c4. Since \{t3, c4\}.score < kc.score, c4 and all objects not accessed yet in “restaurant” can be pruned according to Lemma 5. To this point, the current kc is lbkc3.

Lemma 5: If kc.score > \{ok, ok.nn_{n1}^{ki}\}, ok.nn_{n2}^{ki} and ok.nn_{n3}^{ki} (t’ > t) must not be in lbkcok.

Proof: By definition, kc.score ≤ lbkcok.score. Since \{ok, ok.nn_{n1}^{ki}\}.score < kc.score, we have \{ok, ok.nn_{n2}^{ki}\}.score < lbkcok.score and in turn \{ok, ok.nn_{n3}^{ki}\}.score < lbkcok.score. If ok.nn_{n1}^{ki} is in lbkcok, \{ok, ok.nn_{n1}^{ki}\}.score ≥ lbkcok.score according to Lemma 1, so is ok.nn_{n1}^{ki}. Thus, ok.nn_{n1}^{ki} and ok.nn_{n3}^{ki} must not be in lbkcok.

For each non-principal query keyword \ki, after retrieving the first t keyword-NNs of ok in keyword \ki, we use \ki.score to denote \{ok, ok.nn_{n1}^{ki}\}.score. For example in Figure 4, “restaurant”.score is 0.7, 0.6 and 0.35 after retrieving the 1st, 2nd and 3rd keyword-NN of t3 in “restaurant”. From Lemma 5,}

\[
\text{Lemma 6: lbkcok} \equiv \text{kc once kc.score} > \max_{k_i \in T/(k)} (k_i \text{.score})
\]

\[
\text{Algorithm 4: Local_Best_Keyword_Cover(ok, T)}
\]
\[
\text{Input: A set of query keywords T, a principal object ok}
\]
\[
\text{Output: lbkcok}
\]
\[
1 \text{foreach non-principal query keyword } k_i \in T \text{ do}
2 \quad S \gets \text{retrieve } ok.nn_{n1}^{ki};
3 \quad k_i \text{.score} \gets \{ok, ok.nn_{n1}^{ki}\}.score;
4 \quad k.i.n = 1;
5 \quad kc \gets \text{the keyword cover in } S;
6 \quad \text{while } T \neq \emptyset \text{ do}
7 \quad \quad \text{Find } k_i \in T/\{k\}, k_i \text{.score} = \max_{k_j \in T/(k)} (k_j \text{.score});
8 \quad \quad k.i.n \gets k.i.n + 1;
9 \quad \quad S \gets S \cup \text{retrieve } ok.nn_{n1}^{ki};
10 \quad \quad k_i \text{.score} \gets \{ok, ok.nn_{n1}^{ki}\}.score;
11 \quad \quad \text{temp} \_kc \gets \text{the keyword cover in } S;
12 \quad \quad \text{if } \text{temp} \_kc \_score > kc \_score \text{ then}
13 \quad \quad \quad kc \gets \text{temp} \_kc;
14 \quad \quad \quad \text{foreach } k_i \in T/\{k\} \text{ do}
15 \quad \quad \quad \quad \text{if } k_i \text{.score} \leq kc \_score \text{ then}
16 \quad \quad \quad \quad \quad \text{remove } k_i \text{ from } T;
17 \quad \quad \text{return } kc;
\]

Algorithm 4 shows the pseudo-code of lbkcok computing algorithm. For each non-principal query keyword \ki, the first keyword-NN of ok is retrieved and \ki.score = \{ok, ok.nn_{n1}^{ki}\}.score. They are kept in S and the best keyword cover kc in S is identified using Generate_Candidate function in Algorithm 3. The objects in different keywords are combined. Each time the most promising combination are selected to further do further combination until the best keyword cover is identified. When the second keyword-NN of ok in \ki is retrieved, \ki.score is updated to \{ok, ok.nn_{n2}^{ki}\}.score, and so on. Each time one non-principal query keyword is selected to search next keyword-NN in it. Note that we always select keyword \ki ∈ T/{k} where \ki.score = \max_{k_j \in T/(k)} (k_j \text{.score}) to minimize the number of keyword-NNs retrieved (line 7). After the next keyword-NN of ok in this keyword is retrieved, it is inserted into S and kc is updated. If \ki.score < kc.score, all objects in \ki can be pruned by deleting \ki from T according to Lemma 5. When T is empty, kc is returned to lbkcok according to Lemma 6.

6.2 Keyword-NNE Algorithm

In keyword-NNE algorithm, the principal objects are processed in blocks instead of individually. Let k be the principal query keyword. The principal objects are indexed using KRR*_k-tree. Given a node N_k in KRR*_k-tree, also known as a principal node, the local best keyword cover of

![Diagram](http://example.com/diagram.png)

Fig. 4. An example of lbkc computation.
$N_k$, $lbkc_{N_k}$, consists of $N_k$ and the corresponding nodes of $N_k$ in each non-principal query keyword.

Definition 6 (Corresponding Node): $N_k$ is a node of KRR*-k-tree at the hierarchical level $i$. Given a non-principal query keyword $k_i$, the corresponding nodes of $N_k$ are nodes in KRR*-k-tree at the hierarchical level $i$.

The root of a KRR*-tree is at hierarchical level 1, its child nodes are at hierarchical level 2, and so on. For example, if $N_k$ is a node at hierarchical level 4 in KRR*-tree, the corresponding nodes of $N_k$ in keyword $k_i$ are these nodes at hierarchical level 4 in KRR*-k-tree. From the corresponding nodes, the keyword-NNs of $N_k$ are retrieved incrementally for computing $lbkc_{N_k}$.

Lemma 7: If a principal object $o_k$ is an object under a principal node $N_k$ in KRR*-k-tree

$$lbkc_{N_k}.score \geq lbkc_{o_k}.score.$$ 

Proof: Suppose $lbkc_{N_k} = \{N_k, N_{k1}, ..., N_{kn}\}$ and $lbkc_{o_k} = \{o_k, o_{k1}, ..., o_{kn}\}$. For each non-principal query keyword $k_i$, $o_{ki}$ is under a corresponding node of $N_k$, say $N_{ki}$. Note that $N_{ki}$ can be in $lbkc_{N_k}$ or not. By definition,

$$lbkc_{N_k}.score \geq \{N_k, N_{k1}, ..., N_{kn}\}.score.$$ 

According to Lemma 2

$$\{N_k, N_{k1}, ..., N_{kn}\}.score \geq lbkc_{o_k}.score.$$ 

So, we have

$$lbkc_{N_k}.score \geq lbkc_{o_k}.score.$$ 

The lemma is proved.

The pseudo-code of keyword-NNE algorithm is presented in Algorithm 5. Keyword-NNE algorithm starts by selecting a principal query keyword $k \in T$ (line 2). Then, the root node of KRR*-k-tree is visited by keeping its child nodes in a heap $H$. For each node $N_k$ in $H$, $lbkc_{N_k}.score$ is computed (line 5). In $H$, the one with the maximum score, denoted as $H.head$, is processed. If $H.head$ is a node of KRR*-k-tree (line 8-14), it is replaced in $H$ by its child nodes. For each child node $N_k$, we compute $lbkc_{N_k}.score$. Correspondingly, $H.head$ is updated. If $H.head$ is a principal object of $o_k$ rather than a node in KRR*-k-tree (line 15-21), $lbkc_{o_k}$ is computed. If $lbkc_{o_k}$ is greater than the score of the current best solution $bkc$ ($bkc.score = 0$ initially), $bkc$ is updated to be $lbkc_{o_k}$. For any $N_k \in H$, $N_k$ is pruned if $lbkc_{N_k}.score \leq lbkc_{o_k}.score$ since $lbkc_{o_k}.score \leq lbkc_{o_k}$ for every $o_k$ under $N_k$ in KRR*-k-tree according to Lemma 7. Once $H$ is empty, $bkc$ is returned to BKC query.

### Algorithm 5: Keyword-NNE(T, D)

Input: A set of query keywords $T$, a spatial database $D$ 
Output: Best Keyword Cover

1. $bkc.score \leftarrow 0$;
2. $k \leftarrow$ select the principal query keyword from $T$;
3. $H \leftarrow$ child nodes of the root in KRR*-k-tree;
4. foreach $N_k \in H$ do
   5. Compute $lbkc_{N_k}.score$;
   6. $H.head \leftarrow N_k$ with $\max_{N_k \in H} lbkc_{N_k}.score$;
7. while $H \neq 0$ do
   8. while $H.head$ is a node in KRR*-k-tree do
      9. $N \leftarrow$ child nodes of $H.head$;
      10. foreach $N_k$ in $N$ do
          11. Compute $lbkc_{N_k}.score$;
          12. Insert $N_k$ into $H$;
          13. Remove $H.head$ from $H$;
      14. $H.head \leftarrow N_k$ with $\max_{N_k \in H} lbkc_{N_k}.score$;
     ="/ H.head is a principal object (i.e., not a node in KRR*-k-tree) */
      15. $o_k \leftarrow H.head$;
      16. Compute $lbkc_{o_k}.score$;
      17. if $bkc.score < lbkc_{o_k}.score$ then
          18. $bkc \leftarrow lbkc_{o_k}$;
      19. foreach $N_k$ in $H$ do
          20. if $lbkc_{N_k}.score \leq bkc.score$ then
              21. Remove $N_k$ from $H$;
22. return $bkc$;

where $W_Average(O)$ is defined as:

$$W_Average(O) = \frac{\sum_{o_k \in O} w_{ki} * o_k.rating}{|O|}. \quad (9)$$

where $w_{ki}$ is the weight associated with the query keyword $k_i$ and $\sum_{k_i \in T} w_{ki} = 1$. For example, a user may give higher weight to “hotel” but lower weight to “restaurant” in a BKC query. Given the score function in Equation (8), the baseline algorithm and keyword-NNE algorithm can be used to process BKC query with minor modification. The core is to maintain the property in Lemma 1 and in Lemma 2 which are the foundation of the pruning techniques in the baseline algorithm and the keyword-NNE algorithm.

However, the property in Lemma 1 is invalid given the score function defined in Equation (9). To maintain this property, if a combination does not cover a query keyword $k_i$, this combination is modified by inserting a virtual object associated with $k_i$. This virtual object does not change the diameter of the combination, but it has the maximum rating of $k_i$ (for the combination of nodes, a virtual node
is inserted in the similar way). The $W_{\text{Average}}(O)$ is redefined to $W_{\text{Average}}^*(O)$.

$$W_{\text{Average}}^*(O) = \frac{E + F}{T}. \tag{10}$$

where $T/O.T$ is the set of query keywords not covered by $O$, $O_{kj}.\text{maxrating}$ is the maximum rating of objects in $O_{kj}$. For example in Figure 1, suppose the query keywords are “restaurant”, “hotel” and “bar”. For a combination $O = \{t_1, c_1\}$, $W_{\text{Average}}(O) = w_t\cdot t_1.\text{rating} + w_c\cdot c_1.\text{rating}$ while $W_{\text{Average}}^*(O) = w_t\cdot t_1.\text{rating} + w_c\cdot c_1.\text{rating}$ + $w_s\cdot \text{max rating}_s$ where $w_t$, $w_c$ and $w_s$ are the weights assigned to “bar”, “restaurant” and “hotel” respectively, and $\text{max rating}_s$ is the highest keyword rating of objects in “hotel”.

Given $O.\text{score}$ with $W_{\text{Average}}^*(O)$, it is easy to prove that the property in Lemma 1 is valid. Note that the purpose of $W_{\text{Average}}^*(O)$ is to apply the pruning techniques in the baseline algorithm and keyword-NNE algorithm. It does not affect the correctness of the algorithms. In addition, the property in Lemma 2 is valid no matter which of $W_{\text{Average}}^*(O)$ and $W_{\text{Average}}(O)$ is used in $O.\text{score}$.

8 ANALYSIS

To help analysis, we assume a special baseline algorithm $BF$-baseline which is similar to the baseline algorithm but the best-first KRR*-tree browsing strategy is applied. For each query keyword, the child nodes of the KRR*-tree root are retrieved. The child nodes from different query keywords are combined to generate candidate keyword covers (in the same way as in the baseline algorithm, see section 5) which are stored in a heap $H$. The candidate $kc \in H$ with the maximum $\text{score}$ is processed by retrieving the child nodes of $kc$. Then, the child nodes of $kc$ are combined to generate more candidates which replace $kc$ in $H$. This process continues until a keyword cover consisting of objects only is obtained. This keyword cover is the current best solution $bkc$. Any candidate $kc \in H$ is pruned if $kc.\text{score} \leq bkc.\text{score}$. The remaining candidates in $H$ are processed in the same way. Once $H$ is empty, the current $bkc$ is returned to $BKC$ query. In $BF$-baseline algorithm, only if a candidate keyword cover $kc$ has $kc.\text{score} > BKC.\text{score}$, it is further processed by retrieving the child nodes of $kc$ and combining them to generate more candidates.

8.1 Baseline

However, $BF$-baseline algorithm is not feasible in practice. The main reason is that $BF$-baseline algorithm requires to maintain $H$ in memory. The peak size of $H$ can be very large because of the exhaustive combination until the first current best solution $bkc$ is obtained. To release the memory bottleneck, the depth-first browsing strategy is applied in the baseline algorithm such that the current best solution is obtained as soon as possible (see section 5). Compared to the best-first browsing strategy which is global optimal, the depth-first browsing strategy is a kind of greedy algorithm which is local optimal. As a consequence, if a candidate keyword cover $kc$ has $kc.\text{score} > BKC.\text{score}$, $kc$ is further processed by retrieving the child nodes of $kc$ and combining them to generate more candidates. Note that $bkc.\text{score}$ increases from 0 to $BKC.\text{score}$ in the baseline algorithm. Therefore, the candidate keyword covers which are further processed in the baseline algorithm can be much more than that in $BF$-baseline algorithm.

Given a candidate keyword cover $kc$, it is further processed in the same way in both the baseline algorithm and $BF$-baseline algorithm, i.e., retrieving the child nodes of $kc$ and combine them to generate more candidates using $\text{Generate Candidate}$ function in Algorithm 3. Since the candidate keyword covers further processed in the baseline algorithm can be much more than that in $BF$-baseline algorithm, the total candidate keyword covers generated in the baseline algorithm can be much more than that in $BF$-baseline algorithm.

Note that the analysis captures the key characters of the baseline algorithm in $BKC$ query processing which are inherited from the methods for $mCK$ query processing [17, 18]. The analysis is still valid if directly extending the methods [17, 18] to process $BKC$ query as introduced in section 4.

8.2 Keyword-NNE

In keyword-NNE algorithm, the best-first browsing strategy is applied like $BF$-baseline but large memory requirement is avoided. For the better explanation, we can imagine
all candidate keyword covers generated in BF-baseline algorithm are grouped into independent groups. Each group is associated with one principal node (or object). That is, the candidate keyword covers fall in the same group if they have the same principal node (or object). Given a principal node \( N_k \), let \( G_{N_k} \) be the associated group. The example in Figure 5 shows \( G_{N_k} \) where some keyword covers such as \( kc_1, kc_2 \) have score greater than \( BKC.score \), denoted as \( G_{N_{k1}} \), and some keyword covers such as \( kc_3, kc_4 \) not greater than \( BKC.score \), denoted as \( G_{N_{k2}} \). In BF-baseline algorithm, \( G_{N_k} \) is maintained in \( H \) before the first current best solution is obtained, and every keyword cover in \( G_{N_k} \) needs to be further processed.

In keyword-NNE algorithm, the keyword cover in \( G_{N_k} \) with the highest score, i.e., \( lbkc_{N_k} \), is identified and maintained in memory. That is, each principal node (or object) keeps its \( lbkc \) only. The total number of principal nodes (or objects) is \( O(n \log n) \) where \( n \) is the number of principal objects. So, the memory requirement for maintaining \( H \) is \( O(n \log n) \). The (almost) linear memory requirement makes the best-first browsing strategy practical in keyword-NNE algorithm. Due to the best-first browsing strategy, \( lbkc_{N_k} \) is further processed in keyword-NNE algorithm only if \( lbkc_{N_k}.score > BKC.score \).

### 8.2.1 Instance Optimality

The instance optimality [7] corresponds to the optimality in every instance, as opposed to just the worst case or the average case. There are many algorithms that are optimal in a worst-case sense, but are not instance optimal. An example is binary search. In the worst case, binary search is guaranteed to require no more than \( \log N \) probes for \( N \) data items. By linear search which scans through the sequence of data items, \( \log N \) probes are required in the worst case. However, binary search is not better than linear search in all instances. When the search item is in the very first position of the sequence, a positive answer can be obtained in one probe and a negative answer in two probes using linear search. The binary search may still require \( \log N \) probes.

Instance optimality can be formally defined as follows: for a class of correct algorithms \( A \) and a class of valid input \( D \) to the algorithms, \( cost(A, D) \) represents the amount of a resource consumed by running algorithm \( A \in A \) on input \( D \in D \). An algorithm \( B \in A \) is instance optimal over \( A \) and \( D \) if \( cost(B, D) = O(cost(A, D)) \) for \( \forall A \in A \) and \( \forall D \in D \). This cost could be running time of algorithm \( A \) over input \( D \).

**Theorem 1:** Let \( D \) be the class of all possible spatial databases where each tuple is a spatial object and is associated with a keyword. Let \( A \) be the class of any correct \( BKC \) processing algorithm over \( D \in D \). For all algorithms in \( A \), multiple \( KRR* \)-trees, each for one keyword, are explored by combining nodes at the same hierarchical level until leaf node, and no combination of objects (or nodes of \( KRR* \)-trees) has been pre-processed. keyword-NNE algorithm is optimal in terms of the number of candidate keyword covers which are further processed.

**Proof:** Due to the best-first browsing strategy, \( lbkc_{N_k} \) is further processed in keyword-NNE algorithm only if \( lbkc_{N_k}.score > BKC.score \). In any algorithm \( A \in A \), a number of candidate keyword covers need to be generated and assessed since no combination of objects (or nodes of \( KRR* \)-trees) has been pre-processed. Given a node (or object) \( N \), the candidate keyword covers generated can be organized in a group if they contain \( N \). In this group, if one keyword cover has score greater than \( BKC.score \), the possibility exists that the solution of \( BKC \) query is related to this group. In this case, \( A \) needs to process at least one keyword cover in this group. If \( A \) fails to do this, it may lead to an incorrect solution. That is, no algorithm in \( A \) can process less candidate keyword covers than keyword-NNE algorithm.

### 8.2.2 Candidate Keyword Covers Processing

Every candidate keyword cover in \( G_{N_k} \) is further processed in BF-baseline algorithm. In the example in Figure 5, \( kc_1 \) is further processed, so does every \( kc \in G_{N_k} \). Let us look closer at \( kc_1 = \{ N_k, N_{k1}, N_{k2} \} \) processing. As introduced in section 4, each node \( N \) in \( KRR* \)-tree is defined as \( N(x, y, r, l_x, l_y, l_r) \) which can be represented with 48 bytes. If the disk pagesize is 4096 bytes, the reasonable fan-out of \( KRR* \)-tree is 40-50. That is, each node in \( kc_1 \) (i.e., \( N_k, N_{k1} \), and \( N_{k2} \)) has 40-50 child nodes. In \( kc_1 \) processing in BF-baseline algorithm, these child nodes are combined to generate candidate keyword covers using Algorithm 3.

In keyword-NNE algorithm, one and only one keyword cover in \( G_{N_k} \), i.e., \( lbkc_{N_k} \), is further processed. For each child node \( cN_k \) of \( N_k \), \( lbkc_{cN_k} \) is computed. For computing \( lbkc_{cN_k} \), a number of keyword-NNs of \( cN_k \) are retrieved and combined to generate more candidate keyword covers using Algorithm 3. The experiments on real data sets illustrate that only 2-4 keyword-NNs in average in each non-principal query keyword are retrieved in \( lbkc_{cN_k} \) computation.

When further processing a candidate keyword cover, keyword-NNE algorithm typically generates much less new candidate keyword covers compared to BF-baseline algorithm. Since the number of candidate keyword covers is independent of the principal query keyword since the analysis does not apply any constraint on the selection strategy of principal query keyword.

### 9 Experiment

In this section we experimentally evaluate keyword-NNE algorithm and the baseline algorithm. We use four real data sets, namely Yelp, Yellow Page, AU, and DE. Specifically, Yelp is a data set extracted from Yelp Academic Dataset (www.yelp.com) which contains 7707 POIs (i.e., points
of interest, which are equivalent to the objects in this work) with 27 keywords where the average, maximum and minimum number of POIs in each keyword are 285, 1353 and 120 respectively. Yellow Page is a data set obtained from yellowpage.com.au in Sydney which contains 30444 POIs with 26 keywords where the average, maximum and minimum number of POIs in each keyword are 1170, 10284 and 154 respectively. All POIS in Yelp have been rated by customers from 1 to 10. About half of the POIs in Yellow Page have been rated by Yelp, the unrated POIs are assigned average rating 5. AU and US are extracted from a public source \(^3\). AU contains 678581 POIs in Australia with 187 keywords where the average, maximum and minimum number of POIs in each keyword are 3728, 53956 and 403 respectively. US contains 1541124 POIs with 237 keywords where the average, maximum and minimum number of POIs in each keyword are 6502, 122669 and 400. In AU and US, the keyword ratings from 1 to 10 are randomly assigned to POIs. The ratings are in normal distribution where the mean \(\mu = 5\) and the standard deviation \(\sigma = 1\). The distribution of POIs in keywords are illustrated in Figure 6. For each data set, the POIs of each keyword are indexed using a KRR*-tree.

Figure 6. The distribution of keyword size in test data sets.

Fig. 6. The distribution of keyword size in test data sets.

Fig. 7. Baseline, Virtual bR*-tree and bR*-tree.

We are interested in 1) the number of candidate keyword covers generated, 2) \(BKC\) query response time, 3) the maximum memory consumed, and 4) the average number of keyword-NNs of each principal node (or object) retrieved for computing \(lbkc\) and the number of \(lbkes\) computed for answering \(BKC\) query. In addition, we test the performance in the situation that the weighted average of keyword ratings is applied as discussed in section 7. All algorithms are implemented in Java 1.7.0. and all experiments have been performed on a Windows XP PC with 3.0 Ghz CPU and 3 GB main memory.

In Figure 7, the number of keyword covers generated in baseline algorithm is compared to that in the algorithms directly extended from [17, 18] when the number of query keywords \(m\) changes from 2 to 9. It shows that the baseline algorithm has better performance in all settings. This is consistent with the analysis in section 5. The test results on Yellow Page and Yelp data sets are shown in Figure 7 (a) which represents data sets with large number of keywords. The test results on AU and US data sets are shown in Figure 7 (b) which represents data set with small number of keywords. As observed, when the number of keywords in a data set is small, the difference between baseline algorithm and the directly extended algorithms is reduced. The reason is that the single tree index in the directly extended algorithms has more pruning power in this case (as discussed in section 4).

9.1 Effect of \(m\)

The number of query keywords \(m\) has significant impact to query processing efficiency. In this test, \(m\) is changed from 2 to 9 when \(\alpha = 0.4\). Each BKC query is generated by randomly selecting \(m\) keyword from all keywords as the query keywords. For each setting, we generate and perform 100 BKC queries, and the averaged results are reported \(^4\). Figure 8 shows the number of candidate keyword covers generated for \(BKC\) query processing. When \(m\) increases, the number of candidate keyword covers generated increases dramatically in the baseline algorithm. In contrast, keyword-NNE algorithm shows much better scalability. The reason has been explained in section 8.

Figure 9 reports the average response time of \(BKC\) query when \(m\) changes. The response time is closely related to the candidate keyword covers generated during the query processing. In the baseline algorithm, the response time increases very fast when \(m\) increases. This is consistent with the fast increase of the candidate keyword covers generated when \(m\) increases. Compared to the baseline algorithm, the keyword-NNE algorithm shows much slower increase when \(m\) increases.

For processing the \(BKC\) queries at the same settings of \(m\) and \(\alpha\), the performances are different on datasets US, AU, YELP and Yellow Page as shown in Figure 8 and Figure 9. The reason is that the average number of objects in one keyword in datasets US, AU, YELP and Yellow Page are 6502, 3728, 1170 and 285 respectively as shown in Figure 6; in turn, the average numbers of objects in one keyword in the \(BKC\) queries on dataset US, AU, YELP and Yellow Page as shown in Figure 8.


4. In this work, all experimental results are obtained in the same way.
AU, YELP and Yellow Page are expected to be the same. The experimental results show the higher average number such as on dataset US leads to the more candidate keyword covers and the more processing time.

9.2 Effect of $\alpha$

This test shows the impact of $\alpha$ to the performance. As shown in Equation (2), $\alpha$ is an application specific parameter to balance the weight of keyword rating and the diameter in the score function. Compared to $m$, the impact of $\alpha$ to the performance is limited. When $\alpha = 1$, $BKC$ query is degraded to $mKC$ query where the distance between objects is the sole factor and keyword rating is ignored. When $\alpha$ changes from 1 to 0, more weight is assigned to keyword rating. In Figure 10, an interesting observation is that with the decrease of $\alpha$ the number of keyword covers generated in both the baseline algorithm and keyword-NNE algorithm shows a constant trend of slight decrease. The reason behind is that KRR*-tree has a keyword rating dimension. Objects close to each other geographically may have very different ratings and thus they are in different nodes of KRR*-tree. If more weight is assigned to keyword ratings, KRR*-tree tends to have more pruning power by distinguishing the objects close to each other but with different keyword ratings. As a result, less candidate keyword covers are generated. Figure 11 presents
the average response time of queries which are consistent with the number of candidate keyword covers generated.

**BKC** query provides robust solutions to meet various practical requirements while **mCK** query cannot. Suppose we have three query keywords in Yelp dataset, namely, “bars”, “hotels & travel”, and “fast food”. When \( \alpha = 1 \), the BKC query (equivalent to mCK query) returns Pita House, Scottsdale Neighborhood Trolley, and Schlotzskys (the names of the selected objects in keyword “bars”, “hotels & travel”, and “fast food” respectively) where the lowest keyword rating is 2.5 and the maximum distance is 0.045km. When \( \alpha = 0.4 \), the BKC query returns The Attic, Enterprise Rent-A-Car and Chick-Fil-A where the lowest keyword rating is 4.5 and the maximum distance is 0.662km.

### 9.3 Maximum Memory Consumption

The maximum memory consumed by the baseline algorithm and keyword-NNE algorithm are reported in Figure 12 (the average results of 100 BKC queries on each of four data sets). It shows that the maximum memory consumed in keyword-NNE algorithm is up to 0.5MB in all settings of \( m \) while it increases very fast when \( m \) increases in the baseline algorithm. As discussed in section 8, keyword-NNE algorithm only maintains the principal nodes (or objects) in memory while the baseline algorithm maintains candidate keyword covers in memory.

![Fig. 12. Maximum memory consumed vs. \( m \) (\( \alpha=0.4 \)).](image)

### 9.4 Keyword-NNE

The high performance of keyword-NNE algorithm is due to that each principal node (or object) only retrieves a few keyword-NNs in each non-principal query keyword. Suppose all retrieved keyword-NNs in keyword-NNE algorithm are kept in a set \( S \). In Figure 13 (a), the average size of \( S \) is shown. The data sets are randomly sampled so that the number of objects in each query keyword in a BKC query is from 100 to 3000. It illustrates that the impact of the number of objects in query keywords to the size of \( S \) is limited. On the contrary, it shows that the size of \( S \) is clearly influenced by \( m \). When \( m \) increases from 2 to 9, \( S \) increases linearly. In average, a principal node (or object) only retrieves 2-4 keyword-NNs in each non-principal query keyword. Figure 13 (b) shows the number of lbkcs computed in query processing. We can see less than 10% of principal nodes (or objects) need to compute their lbkcs in different sizes of data sets. In other words, 90% of the overall principal nodes (or objects) are pruned during the query processing.

![Fig. 13. Features of keyword-NNE (\( \alpha=0.4 \)).](image)

### 9.5 Weighted Average of Keyword Ratings

The tests compare the weighted average of keyword rating and the minimum keyword rating to performance. The average experimental results of 100 BKC queries on each of four data sets are reported in Figure 14. We can see the difference between these two situations is trivial. This is because the score computation in the situation of the minimum keyword rating is fundamentally equivalent to that in the situation of weight average. In the former situation, if a combination \( O \) of objects (or their MBRs) does not cover a keyword, the rating of this keyword used for computing \( O.score \) is 0 while it is the maximum rating of this keyword in the latter situation.

![Fig. 14. Weighted average vs. minimum (\( \alpha=0.4 \)).](image)

### 10 Conclusion

Compared to the most relevant mCK query, BKC query provides an additional dimension to support more sensible decision making. The introduced baseline algorithm is inspired by the methods for processing mCK query. The baseline algorithm generates a large number of candidate keyword covers which leads to dramatic performance drop when more query keywords are given. The proposed keyword-NNE algorithm applies a different processing strategy, i.e., searching local best solution for each object in a certain query keyword. As a consequence, the number of candidate keyword covers generated is significantly reduced. The analysis reveals that the number of candidate

---

1041-4347 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
keyword covers which need to be further processed in keyword-NNE algorithm is optimal and processing each keyword candidate cover typically generates much less new candidate keyword covers in keyword-NNE algorithm than in the baseline algorithm.

REFERENCES


Ke Deng was awarded a PhD degree in computer science from The University of Queensland, Australia, in 2007 and a Master degree in Information and Communication Technology from Griffith University, Australia, in 2001. His research background include high performance database system, spatiotemporal data management, data quality control and business information system. His current research interest focuses on big spatiotemporal data management and mining.

Xin Li is a Master candidate in Computer Science at Renmin University of China. He received the Bachelor degree from School of Information at Renmin University of China in 2007. His current research focuses on spatial database and geo-positioning.

Jiaheng Lu is a professor in Computer Science at the Renmin University of China. He received the Ph.D. degree from National University of Singapore in 2007 and M.S. from Shanghai Jiaotong University in 2001. His research interests span many aspects of data management. His current research focuses on developing an academic search engine, XML data management, and big data management. He has served in the organization and program committees for various SIGMOD, VLDB, ICDE and CIKM.

Xiaofang Zhou received the BS and MS degrees in computer science from Nanjing University, China, in 1984 and 1987, respectively, and the PhD degree in computer science from The University of Queensland, Australia, in 1994. He is a professor of computer science with The University of Queensland. He is the head of the Data and Knowledge Engineering Research Division, School of Information Technology and Electrical Engineering. He is also an specially appointed Adjunct Professor at Soochow University, China. From 1994 to 1999, he was a senior research scientist and project leader in CSIRO. His research is focused on finding effective and efficient solutions to managing integrating and analyzing very large amounts of complex data for business and scientific applications. His research interests include spatial and multimedia databases, high performance query processing, web information systems, data mining, and data quality management. He is a senior member of the IEEE.