OID	PX	PY	$Associated \quad keywords$
o_1	159.0	246.0	mountain 4,landscape 1,temple 5
o_2	171.0	36.0	shore 2,museum 1
o_3	109.5	235.5	forest 4,mountain 1,temple 2
o_4	352.5	271.5	shore 1
o_5	97.5	276.0	driftage 1,shore 5,architecture 1
o_6	331.5	70.5	architecture 5,temple 2
o_7	259.5	177.0	museum 3,mountain 1,landscape 4
o_8	130.5	3.0	glacier 1
o_9	148.5	291.0	forest 4
o_{10}	204.0	58.5	driftage 3,mountain 1,glacier 1

Table 1: Spatial objects database

1. Introduction

With the development of information technique (e.g., GIS), large volumes of spatial data (e.g., tourist attractions, hotels and restaurants) are becoming available. Recently, increasing focus is being given to serve local content in response to users' queries, which facilitates the relevant researches on spatial keyword queries [7, 9, 19, 22], which take both of the locations and textual descriptions of content into consideration. A typical spatial keyword query takes a location and a set of keywords as arguments and returns the single spatial object that best matches these arguments.

In a wide spectrum of applications, however, the users' needs (expressed by keywords) cannot be satisfied with only one single object. Hence, the CoSKQ [3, 13] is proposed, and satisfies users' needs collectively by retrieving a group of objects. Unfortunately, all of these existing works regardless of the level information of keywords, which is critical for users to make decision. For instance, Table 1 shows a spatial object database. There is a integer value associated with each keyword to denote the corresponding level of attraction. For users who prefer to famous mountain may choose o_1, others may prefer to choose $o_3, o_7 or o_{10}$ due to the reason of cost or just for taking exercise. In this scenario, users prefer to find objects that best match the personal preferences with level information of keywords, instead of retrieving objects only to cover the query keywords. To address this problem, we introduce the notion of **weight vector** to capture the weight that query user assigns for each level. Without loss of generality, we hypothesize that the sum of each dimension of wv equals tp 1. We aim at retrieving a group G of objects, and for each query keyword, the weight sum for it can meet given threshold. By adjusting this threshold, users can control the size of result set flexibly and obtain the feasible succedaneous choice as the supplement. In addition, this query can also be used to find a consortium of partners that combine to offer the required capabilities for a given project. To accomplish a project, various kinds of capabilities are required, such as coding, information retrieval, text editing etc. Each partner may possess part of these capabilities, and we can utilize the integer value to denote the level of this ability. With the weight vector to capture the completion ratio for the partners with different level in a limited lifespan. In this situation, the leader may want to retrieve a consortium of partners that for each subtask e.g., coding, the total completion ratio not less than a

An example of $KaGPC$ query			
$q.\ell$	$q.\theta$	$q.P$	$q.\omega$
(31.5,50.0)	0.4	(0.1,0.3,0.2,0.3,0.1)	mountain,temple

SID	Solutions	Coverage Probability	WD_cost
S_1	o_1, o_3	0.4, 0.4	3746.83
S_2	o_1, o_3, o_6	0.4, 0.7	5851.73
S_3	o_1, o_3, o_7	0.5, 0.4	5834.71
S_4	o_1, o_3, o_{10}	0.5, 0.4	4610.38
S_5	o_1, o_6, o_7	0.4, 0.4	6530.98
S_6	o_1, o_6, o_{10}	0.4, 0.4	5306.65
S_7	o_1, o_3, o_6, o_7	0.5, 0.7	7939.61
S_8	o_1, o_3, o_6, o_{10}	0.5, 0.7	6715.28
S_9	o_1, o_3, o_7, o_{10}	0.6, 0.4	6698.26
S_{10}	o_1, o_6, o_7, o_{10}	0.5, 0.4	7394.53
S_{11}	$o_1, o_3, o_6, o_7, o_{10}$	0.6, 0.7	8803.15

Table 2: An example of $KaGPC$ query

threshold within a limited lifespan.

To address this problem, we propose a novel query called KaGPC in our work. We study the KaGPC in 2D Euclidean space. Similar with CoSKQ, the KaGPC query retrieves a group of objects with the minimal cost to cover the query keywords. Significantly, there are several major differences distinguish them. First, we attach the keyword with the level information to distinguish them, which is critical for users to make decision. In contrast, CoSKQ do not take this information into consideration. Second, in our work, we apply weight vector to measure the users' preference for each level and propose weight constraint. Third, we take cost distance as our cost function, not the maximum sum cost used in CoSKQ [3]. Given these differences, we cannot extend the methods of CoSKQ to tackle our problem.

Specifically, given a set of spatial objects O, and a query $q = (\ell, \theta, V, \omega)$, where ℓ is a spatial location and θ is a weight threshold. V is a weight vector. ω represents the query keywords. KaGPC query retrieves a group G of objects that meet the following two conditions:

- For each query keyword, the weight sum of G is not less than θ;
- The cost distance of G is minimized.

EXAMPLE 1: We illustrate an example of KaGPC query at the top of Table 2 with the objects presented in Table 1. We obtain all of the feasible solutions (e.g., S_1 to S_{11}). Even though all of these solutions satisfy the weight constraint, we take S_1 the one with the minimum cost distance as our optimal solution.

In this paper, we map the classic WSC to KaGPC, which denotes that KaGPC query is NP-hard. Due to the intrinsical challenges of NP-hard problem, we design two approximation algorithms with provable approximation ratio. Considering the scale of query keywords is limited, we also provide an exact algorithm in our work.

To summarize, we make the following contributions in this paper.

Notations	Explanations
q	The KaGPC query of form:$(\ell, \theta, P, \omega)$
RO_q	The relevant objects to query q
RO_λ	The relevant objects to keyword λ
$w(o)$	The weight of o
$wd(o,q)$	The weight distance cost of o to query q
$cp(o,\lambda)$	The coverage probability of λ covered by o
$cov(G,q)$	The coverage probability of q covered by G
$cr(o,q)$	The contribution ratio of o to q
$dcr_q^r(o)$	The dynamic contribution ratio of o to q

Table 3: Summary of the notations used.

- We introduce the weight vector and propose a novel type of queries, called KaGPC queries, retrieving a group of objects that cover the query keywords not less than a given threshold. We prove this problem is NP-hard. To the best of our knowledge, this is the first work to address this problem.
- We design two effective approximation algorithms based on the KHT index. Besides, we also propose an exact algorithm for this problem.
- We conduct comprehensive experiments to demonstrate the effectiveness of our proposed algorithms.

The rest of this paper is organized as follows. Section 2 formally defines the novel query and proves the NP-hard complexity of it. Section 3 presents how to construct the KHT index. Section 4 depicts two approximation algorithms and the exact algorithm. We show our experiment results in Section 5, and introduce related work in Section 6. Finally, We conclude our work in Section 7.

2. Problem Statement

In this section, we first introduce the fundamental notions used in this paper. Then we will prove the NP-hard complexity of our problem.

Let O denotes a database containing n spatial objects. Each object $o \in O$ is associated with a location $o.\ell$, a set of keywords $o.\omega$ to capture the tourist attractions or hotels etc. and a $|q.\omega|$ dimensions vector $o.\nu$ with the ith element being the level information of ith keywords in $|o.\omega|$. For an object $o \in O$, we refer to the cost of o as cost(o). For ease of presentation, we take the level vector as a multidimensional positive integer vector. And the upper bound of integer value is fixed (e.g., for attractions not larger than 5, that is to say only has 5 levels for attractions).

Definition 1. (Cost Distance) Given a query q and an object $o \in O$, the cost distance of o can be denoted as:

$$cd(o,q) = cost(o) \cdot dist(o,q) \tag{1}$$

In Equation(1), dist(o,q) refers to the Euclidean distance between o and q. Comparing with the cost function used in [3, 13], cost distance is more adaptive to real application scenarios in that, it not only take the distance between query point q and o but also the internal cost of o into consideration.

Definition 2. (Object coverage weight) Given a keyword λ, a multidimensional level vector V and an object $o \in O$. We use the notation $o.\nu_\lambda$ to denotes the corresponding level value of keyword λ in $o.\nu$. Then the weight that λ covered by o can be represented as:

$$cw(o, \lambda) = P[o.\nu_\lambda] \tag{2}$$

Differing with CoSKQ in which a keyword either covered by object o or not, in this work we take the coverage weight cw as our measurement. Note that if λ is not contained by $o.\omega$, we set $cw(o, \lambda) = 0$. We use $cov(o, q) = \sum_{\lambda \in q.\omega} cp(o, \lambda)$ and $cov(G, \lambda) = \sum_{o \in G} cp(o, \lambda)$ to denote the weight that q is covered by o and the weight that λ is covered by G, respectively.

Definition 3. (Contribution ratio) Given a query q and an object o, we define the contribution ratio of the object o to q as follows:

$$cr(o, q) = \frac{cov(o, q)}{cd(o, q)} \tag{3}$$

By taking both of the coverage weight and the cost distance into consideration, $cr(o, q)$ can fit in with the real application scenarios better than only use $cov(o, q)$ to evaluate the contribution of o to q.

Definition 4. (KaGPC queries) The KaGPC queries $q = (\ell, \theta, P, \omega)$, where ℓ is a spatial location and θ is a threshold. V is a vector model to capture level information and ω represents the query keywords. KaGPC queries aim at retrieving a group G of objects that collectively satisfy the following two conditions:

- For each keyword $\lambda \in q.\omega$, $cov(G, \lambda) \geq q.\theta$;
- $\underset{G}{\arg\min} \sum_{o \in G} cd(o, q)$.

Given a KaGPC queries q, we claim an object o is ***relevant*** to q if o contains at least one keyword $\lambda \in q.\omega$. We use notation RO_λ and RO_q to denote the set of objects that relevant to λ and query q in O, respectively. It is sufficient to take only RO_q instead of O into consideration for a specific query q. If a group G of objects can satisfy the weight constraint of definition 4, we say that G is a ***feasible solution*** of query q. Put differently, KaGPC queries return the feasible solution with minimum costs.

Theorem 1. *The KaGPC queries is NP-hard.*

Proof: We can reduce the classic WSC Problem to the KaGPC problem. A typical instance of the WSC problem of the form $< U, S >$, where $U = \{1, 2, 3, ..., n\}$ of n elements and a family of sets $S = \{S_1, S_2, S_3, ..., S_m\}$, where $S_i \subseteq U$ and each S_i is associated with a positive cost C_{S_i}. The decision problem is to decide if we can find a subset F of S such that $\cup_{S_i \in F} S_i = U$ and such that its cost $\sum_{S_i \in F} C_{S_i}$ is minimized.

Entry	Keywords	OList	FI	SI
e_1	$mountain$	o_1, o_3, o_7, o_{10}	5	0
e_2	$forest$	o_3, o_9	6	1
e_3	$landscape$	o_1, o_7	6	2
e_4	$shore$	o_2, o_4, o_5	6	1
e_5	$temple$	o_1, o_3, o_6	1	0
e_6	$museum$	o_2, o_7	2	0
e_7	$architecture$	o_5, o_6	5	1
e_8	$drigtate$	o_5, o_{10}	0	0
e_9	$glacier$	o_8, o_{10}	4	0

Table 4: KHT entries

To reduce the WSC problem to KaGPC queries q. We map each element of U corresponding to a query keyword, that each set S_i corresponding to a spatial object o_i, and each positive cost C_{S_i} as the $wd(o_i, q)$. For each keyword associated with o_i, we set the integer value to 1. Besides we hypothesize the preference vector $P = \{1, 0, 0, 0, 0\}$ and the threshold is 1. Clearly, there is a solution to weighted set cover problem if and only if there is solution to query q.

3. Pre-Processing

As claimed in Section 2, given a query q it's sufficient to only tackle the relevant objects in RO_q, due to other objects have no contribution to satisfy the user's needs. To alleviate the unnecessary computation and boost the search efficiency, in this section, we introduce the keyword hash table(KHT) index, which is organized as a hash table with ***perfect hashing technique***.

In a nutshell, the KHT consists of two major components:

- Distinct keywords: A vocabulary of all the distinct keywords appearing in the object database.
- OID list: For each distinct keyword λ, there is a posting list which records the RO_λ.

Each entry of KHT of the form $(\lambda, \lambda.olist, FI, SI)$, where λ represents a keyword and $\lambda.olist$ is the RO_λ. FI and SI correspond to the first and second index for λ in KHT. Table 4 elaborates the KHT entries, and Figure 1 shows the KHT structure of Table 1.

To balance the search efficiency and storage space, we combine the two-level index technique and perfect hashing technique[8] in our work. We assign each entry a index (Fi, Si), with which we can retrieve a KHT entry in nearly $\Theta(1)$ time.

As illustrated in Figure 2, we determine the index of an entry in two steps. In the first step, we first map the keyword w of entry to the FTemp with the string hash function BKDRHash. It's worth to note that, although we apply BKDRHash as hash function others string hash function can also be applied. And then, we obtain the Fi by performing a modulus on FTemp with first-level index length M. In the second step, we hash the FTemp with a integer hash function and then perform a modulus on STemp with second-level index length of Fi to get Si. After these two steps, each KHT entry

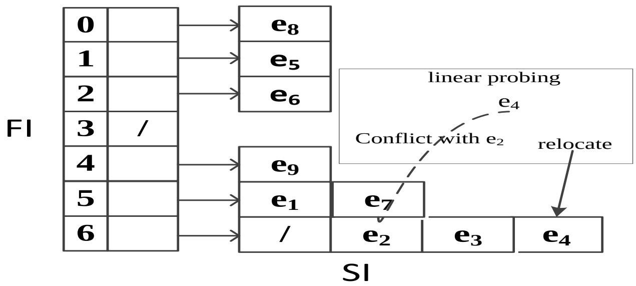


Figure 1: The KHT instance

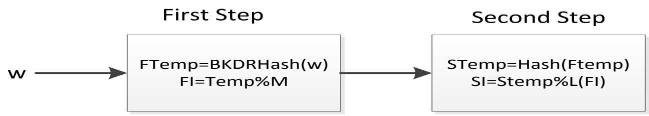


Figure 2: The Hash Function

has a index (Fi, Si). Note that, a delicate situation arises when two or more entries share the same index (Fi, Si) whose probability less than 0.5, which is demonstrated in [8]. In this case, ***"linear probing"*** technique is utilized to tackle this problem. From Table 4 we know that there is a conflict situation between "forest" and "shore". As illustrated in Figure 1, when we try to insert the entry e_4 into KHT, e_2 has already stayed in location $(6, 1)$. Due to location $(6, 2)$ is occupied as well, with "linear probing", e_4 is inserted into $(6, 3)$ ultimately, just as the solid arrow shown.

With KHT, for a specific query q we can retrieve the RO_q in nearly $\Theta(|q.\omega|)$, and prune the unnecessary visit hugely.

4. Algorithms

Considering the inherent complexity of NP-hard problem, we design two approximate algorithms, namely CubeTree and MaxMargin with provable approximation ratio in section 4.1 and 4.2 respectively. Besides, we also provide the exact algorithm MergeList to compare.

4.1. Approximation Algorithm CubeTree

Similar with the strategy of dynamic programming, CubeTree retrieves the final solution by combining the solutions of the sub-queries in a bottom-up manner. Before delving more into CubeTree algorithm, we introduce the notion of *keywords cube* which will lay the foundation for the CubeTree algorithm.

Definition 5. (keywords cube) Given a query q, a set of keywords $\kappa \subseteq q.\omega$ and $\kappa \neq \emptyset$. We define the keywords cube $Cube_\kappa$ as a subset of RO_κ which is in the descending order of $cr(o, q)$ and satisfies the following three conditions:

- For each keyword $\lambda \in \kappa$, $cov(Cube_\kappa, \lambda) \geq q.\theta$;

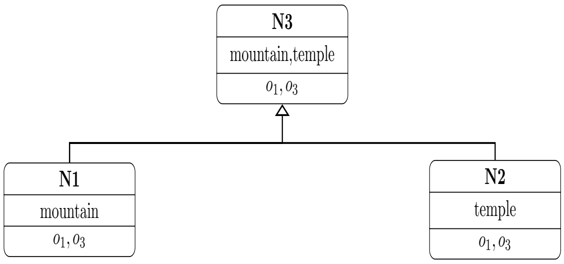


Figure 3: The cubeTree of query q of Table 2.

- For any $o_i \in Cube_\kappa$, $o_j \in RO_\kappa - Cube_\kappa$, $cr(o_i, q) \geq cr(o_j, q)$;
- No subset of $Cube_\kappa$ satisfies the above two conditions.

For brevity, we refer to keywords cube as cube. Note that for an object o, it may exist in several cubes for different keywords set. In this paper hereafter, we use CubeTree to denote the Algorithm and utilize cubeTree to represent tree structure described below, whenever there is no ambiguity.

Each node of cubeTree of the form $(NID, \kappa, Cube_\kappa)$, where NID is the identifier of the cubeTree node. κ and $Cube_\kappa$ denote the keywords set and the corresponding cube, respectively. Figure 3 depicts the cubeTree for query of Table 2.

In a nutshell, CubeTree algorithm builds the cubeTree iteratively in a bottom-up manner. For k query keywords, there are up to $2^k - 1$ nodes to be computed, which results in vast computation overhead. To alleviate the overhead, we adopt a strategy which can balance the overhead and the approximation bound. In this paper, instead of computing all of the nodes, we only compute a fraction of the them as follows.

CubeTree iteratively constructs higher-level node by combining two adjacent low-level nodes, until there is only one single node in current level, which is the root node of cubeTree. In doing so, we only need to compute $\frac{k \cdot (k+1)}{2}$ nodes, which significantly reduces the computation overhead.

As illustrated in Figure 2. In the bottom of cubeTree, there are two nodes corresponding to the two query keywords, then we combine the adjacent two nodes, namely, N1 and N2 to form the higher-lever nodes N3. We return the cube of N3 as our approximation solution directly.

Lemma 1. *If* $\frac{cov(o_l,q)}{wd(o_l,q)} \geq \frac{cov(o_m,q)}{wd(o_m,q)} \geq \frac{cov(o_n,q)}{wd(o_n,q)}$*,then we have* $\frac{cov(o_l,q)+(cov(o_m,q)}{wd(o_l,q)+wd(o_m,q)} \geq \frac{cov(o_n,q)}{wd(o_n,q)}$.

PROOF. *It's obvious with the basic knowledge of fraction, due to the limitation of space, we omit the proof here.*

With this lemma, we can construct the cubeTree with a simple strategy. In each iteration, we add the object with maximum $cr(o, q)$ into the cube of higher-level node instead of adding the object with maximum coverage weight.

We summary our discussion above by three steps:

- Step 1 (Construct the bottom node): Construct the bottom nodes of cubeTree with KHT.

- Step 2 (Combine the low-level nodes): In this step, we obtain the higher-level node by combining two low-level adjacent nodes.
- Step 3 (Iterative step): Repeat Step 2 until there is only one single node in current level and we take this node as root. Then we return the cube of root as our final solution.

Algorithm 1: *CubeTree*

Input : The KHT and the query q.
Output: A group G of objects as approximation solution.

1 $\mathsf{G} \longleftarrow \emptyset$;
2 $\mathsf{lcID} \longleftarrow 0$; $\mathsf{rcID} \longleftarrow 0$; $\mathsf{cID} \longleftarrow 0$;
3 **for** $i \longleftarrow 0$ **to** $|q.\omega| - 1$ **do**
4   cID $\longleftarrow 2^i$;
5     $\kappa \longleftarrow q.\omega[i]$;
6   construct the $Cube_\kappa$ of keywords set κ with KHT;
7   put the node $(cID, \kappa, Cube_\kappa)$ into $cubeTree$;
8 **for** $i \longleftarrow 2$ **to** $|q.\omega|$ **do**
9   **for** $j \longleftarrow 0$ **to** $|q.\omega| - i$ **do**
10   **if** $j == 0$ **then**
11             $seq \longleftarrow 2^i - 1$;
12   **else**
13             $seq \longleftarrow seq \times 2$;
14         $\mathsf{cID} \longleftarrow seq$;
15         $\mathsf{lcID} \longleftarrow seq - 2^{i+j-1}$;
16         $\mathsf{rcID} \longleftarrow seq - 2^j$;
17         $\kappa \longleftarrow cubeTree[lcID].\kappa \cup cubeTree[rcID].\kappa$;
18         $cubecID \longleftarrow combineCube(lcID, rcID)$;
19   put the node $(cID, \kappa, cubecID)$ into $cubeTree$;
20 $solutionID \longleftarrow 2^{|q.\omega|} - 1$;
21 $G \longleftarrow$ the cube of $cubeTree[solutionID]$;
22 return G as the final solution;

The pseudocode of CubeTree is outlined in Algorithm 1. CubeTree takes a bottom-up strategy to construct the cubeTree. Generally, CubeTree consists of two phases. In the first phase (lines 3-7), it builds the bottom nodes with KHT. Then, in the second phase (lines 8-19), by combining two low-level adjacent nodes (line 18), namely $cubeTree[lcID]$ and $cubeTree[rcID]$ to form a higher-level node $cubeTree[cID]$ in an iterative manner, until there is only one single node in this level, which is the root node. Finally, we return the cube of root as our final solution.

Algorithm 2 elaborates the process of the second phase. We use *oidl* and *oidr* to record the current optimal object of $cubeTree[lcID]$ and $cubeTree[rcID]$ respectively (lines 4-5). If *oidl* with larger *cr* than *oidr*, we verify whether *oidl* contained by $cubecID$ or not (line 7). If *oidl* has been in $cubecID$, then we delete it from $cubeTree[ldID]$ and continue to select the next object. Otherwise, we add it into $cubecID$ and update the coverage weight of keywords contained by κ. Otherwise, if *oidl* has smaller *cr* than *oidr*, we tackle the *oidr* as the way of *oidl*. Finally, we return the cube of node cID to Algorithm 1.

Algorithm 2: *combineCube*

Input : Two low-level node lcID and rlID.
Output: The cube of the node cID.

$cubecID \longleftarrow \emptyset$;
$\kappa \longleftarrow cubeTree[lcID].\kappa \cup cubeTree[rcID].\kappa$;
while *exists* $\lambda \in \kappa$ *and* $cov(cubecID, \lambda) < q.\theta$ **do**
  $oidl \longleftarrow$ the oid with maximal *cr* in the cube of $cubeTree[lcID]$;
  $oidr \longleftarrow$ the oid with maximal *cr* in the cube of $cubeTree[rcID]$;
  if $cr(oidl, q) \geq cr(oidr, q)$ **then**
    if $oidl \in cubecID$ **then**
      delete *oidl* from the cube of $cubeTree[lcID]$;
      continue;
    else
      put *oidl* into $cubecID$;
      delete *oidl* from the cube of $cubeTree[lcID]$;
      for *each* $\lambda \in \kappa$ **do**
        update the $cov(cubecID, \lambda)$;
  else
    if $oidr \in cubecID$ **then**
      delete *oidr* from the cube of $cubeTree[rcID]$;
      continue;
    else
      put *oidr* into the $cubecID$;
      delete *oidr* from the cube of $cubeTree[rcID]$;
      for *each* $\lambda \in \kappa$ **do**
        update the $cov(cubecID, \lambda)$;
return $cubecID$;

Theorem 2. *The approximation ratio of CubeTree is not larger than* $\frac{l_{max} \cdot cr_{max}}{cr_{min} \cdot q.\theta}$.

Proof. *Assuming that G is the solution returned by CubeTree. We use $cw_{max} = max_{\lambda \in q.\omega} cov(G, \lambda)$ to denote the maximum coverage weight of keyword in $q.\omega$. The maximum and minimum contribution ratio of object in G are cr_{max} and cr_{min}, respectively. We know that the cost of optimal solution:*

$$Cost(Opt) \geq \frac{|q.\omega| \cdot q.\theta}{cr_{max}} \tag{4}$$

and meanwhile,

$$Cost(G) \leq \frac{|q.\omega| \cdot cw_{max}}{cr_{min}} \tag{5}$$

holds. Combining inequalities (4) and (5), we know that

$$\frac{Cost(G)}{Cost(Opt)} \leq \frac{cw_{max} \cdot cr_{max}}{cr_{min} \cdot q.\theta}.$$

4.2. Approximation Algorithm MaxMargin

In section 4.1 we show how to utilize the CubeTree algorithm to handle our problem. Although CubeTree can solve our problem with provable approximation ratio, however, as presented in Theorem 1, the approximation ratio of CubeTree is unstable due to the relevance with final solution. In this section, we elaborate the MaxMargin algorithm, which is inspired by the greedy strategy adopted by the WSC Problem [5], which with more stable performance in terms of approximation ratio. To address WSC problem, the greedy strategy iteratively selects the current optimal subset and updates subsets that have not been visited yet. We modify this strategy by iteratively selecting the object with maximum $cr(o, q)$, and updating objects that have not been visited for our problem. The naive version of MaxMargin solves the problem by four steps. We use G to reserve the solution.

- Step 1 (Construct the RO_q): Construct the relevant set RO_q with KHT and sort the objects of RO_q in descending order of $cr(o, q)$.
- Step 2 (Select the optimal object): In this step, MaxMargin adds the current optimal object o with maximum $cr(o, q)$ into G. Then, o is deleted from RO_q.
- Step 3 (Update RO_q): After adding o into G, we update the $cr(o, q)$ of remaining objects in RO_q.
- Step 4 (Iterative step): Repeat Step 2 and Step 3 until the weight constraint in definition 4 is satisfied. Then we take G as our final solution.

Obviously, two major drawbacks degrade the performance of naive version: 1) lacking of efficient pruning strategies; 2) updating all the remaining objects for each iteration in Step 3 is time consuming. To further boost the search efficiency, we take a look at several notions which settle aforementioned drawbacks collectively.

Definition 6. (Keyword Max Priority Queue): For a given keyword λ, we define the Keyword Max Priority Queue (KMPQ) of λ as the max priority queue of RO_λ according to the value of $cr(o, q)$.

Each element of KMPQ with the form (nid, oid, cr, cv), where nid and oid correspond to the identifier of element and the object, respectively. cr represents the contribution ratio of oid and cv is a $|q.\omega|$ dimensions contribution vector to record the $cw(o, \lambda)$ for each keyword in $q.\omega$. Instead of constructing the RO_q, we construct the KMPQ for each query keywords. Once $cov(G, \lambda)$ reaches $q.\theta$, we can prune objects in the KMPQ of λ safely, which significantly exalts the performance.

To address the second drawback, we expand the notion of cr to fit in with our algorithm.

Definition 7. (Dynamic contribution ratio): Given a query q and an object o, we define the dynamic contribution ratio (dcr) of object o as follows:

$$dcr_q^r = \frac{cov^r(o, q)}{wd(o, q)} \tag{6}$$

Note that the major difference between dcr_q^r and $cr(o,q)$ is that, we replace $cov(o,q)$ with $cov^r(o,q)$, where $cov^r(o,q)$ denotes the last contribution ratio of o after r-th objects added into result set. We do not update dcr for object o in each iteration r until o is chosen as current optimal object.

Lemma 2. *Given a query q, an object o, two integer m, n and* $m \leq n$*, then* $dcr_q^n(o) \leq dcr_q^m(o)$.

PROOF. *After each iteration r, the value of* $cov^r(o,q)$ *is comparable or below than the former iteration. Besides,* $wd(o,q)$ *is constant after each iteration, which results in the decreasing of* $dcr_q(o)$*. If* $m \leq n$*, we can safely draw the conclusion that the* $dcr_q^n(o)$ *is not larger than* $dcr_q^m(o)$.

Instead of updating all of the remaining objects after each iteration, we update the object until it is chosen as the optimal object. Lemma 4.2 indicates that we can always choose the newly updated objects with maximum $dcr_q(o)$ as current optimal object, and objects whose dcr less than $dcr_q(o)$ unnecessary to be updated, which significantly prune the updating overhead.

We summarize the aforementioned discussion by following three steps for enhanced MaxMargin algorithm. And we use G to reserve the final solution and RV to record the residual.

- Step 1 (Construct the KMPQ): In this step, we construct the KMPQ for each keyword of $q.\omega$.
- Step 2 (Select the optimal object): Select the current optimal object o from all the KMPQ. If o already in G, then continue to select next optimal object. Otherwise, we add o into G if $cov^r(o,q)$ less than RV for all the dimensions. Otherwise, we update the $cr(o,q)$ of o and reinsert it into KMPQ.
- Step 3 (Iterative step): Repeat Step 2 until RV equals to 0. Then we take G as our final solution.

Algorithm 3 illustrates the details of MaxMargin algorithm.

MaxMargin utilizes the residual vector RV to record the difference between $q.\theta$ and the coverage weight of G for each dimension dynamically. And we initiate each dimension of RV to be $q.\theta$. There are two different cases when processing the selected optimal object. Case 1: If all dimensions satisfy $maxOID.cv[i] < RV[i]$ (lines 16-25), it need not to update maxOID and we update RV. Case 2: Otherwise, we update the dcr of $maxOID$ and the PQ[indicator]. The algorithm terminates until RV equals to 0. Note that, in Algorithm 3, we set $PQ[j]$ to invalid if $RV[j]$ is 0 (line 23). Put differently, we prune all of the objects in $PQ[j]$ once the $cov(G, q.\omega[j])$ reaches the threshold $q.\theta$. What's more, we update the object $maxOID$ (lines 27-32) only when the coverage weight of o larger than RV in some dimensions instead of updating the remaining RO_q in naive version. Both of these two strategies significantly improve our MaxMargin algorithm.

Example 2: Lets go back to the KaGPC query q depicts in Table 2. As illustrated in Figure 4, we answer the query q in four steps, as follows.

- Step 1 (Initial state): In this step, we construct the KMPQ for "mountain" and "temple", and initiate the $RV = \{0.4, 0.4\}$ and $G = \emptyset$ as depicted in Figure 4(a) and 4(b).

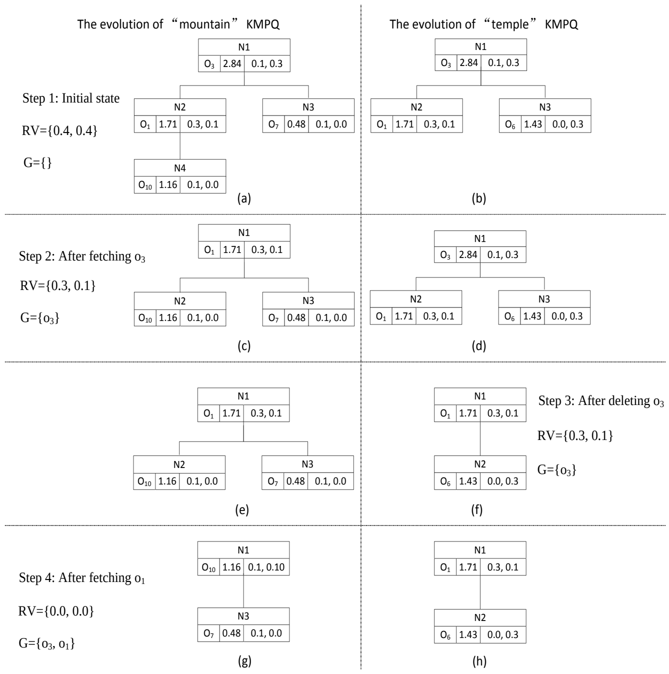


Figure 4: KMPQ for query q of Table 2

- Step 2 (Fetch o_3): Due to the two KMPQs have common optimal object o_3, we can select o_3 from any of these two KMPQs. And we select o_3 from "mountain". After fetching o_3 from "moutain", the "KMPQ" of "moutain" is reorganized and the RV and G is updated as Figure 4(c).
- Step 3 (Delete o_3): As depicted in Figure 4(c) and 4(d). After fetching o_3, the current optimal object is o_3 of "temple". However, o_3 has been in G. So we delete it from "temple" directly, as illustrated in Figure 4(f).
- Step 4 (Fetch o_1): Again, the two KMPQs have common optimal object. We select o_1 from "mountain", and put it into G. Up to now, RV is equal to 0. We terminate our procedure and return G as the final solution.

Theorem 3. *The approximation ratio of MaxMargin is not larger than* $\frac{H(\lfloor cov+1 \rfloor)}{q.\theta}$, *where* cov *is the largest* $cov(o_j, q)$ *for all* $o_j \in RO_q$.

PROOF. *Inspired by the proof in [5], we provide the approximation ratio proof of MaxMargin here. We use* m, n *to denote the number of elements in* $|q.\omega|$ *and* $|RO_q|$ *respectively. We define a* $m \times n$ *matrix* $P = (p_{ij})$ *by*

$$p_{ij} = \begin{cases} cw(o_j, q.\omega[i]) & \text{if } q.\omega[i] \in o_j.\omega, \\ 0 & \text{otherwise.} \end{cases}$$

According to the definition of P, we know that the n columns of P is n coverage weight vectors of n objects. The goal of MaxMargin is to retrieve a group G of objects. And *we utilize the incidence vector $x = (x_j)$ to denote the cover set. Clearly, the incidence vector x of an arbitrary cover satisfies:*

$$\sum_{j=1}^{n} p_{ij}x_j \geq q.\theta \quad \textit{for all } i,$$
$$x_j \in \{0,1\} \quad \textit{for all } j.$$

For ease of presentation, in the following, we refer to the cost distance of o_j as c_j. And we claim that these inequations imply

$$\sum_{j=1}^{n} H(\lfloor cov_j^1 + 1\rfloor)c_j x_j \geq q.\theta \sum (c_j : \textit{where } o_j \in G) \tag{7}$$

for the cover G returned by the greedy heuristic. Once (7) is proved, the theorem will follow by letting x be the incidence vector of an optimal cover.

To prove (7), it will suffice to exhibit nonnegative numbers $y_1, y_2, ..., y_m$ such that

$$\sum_{i=1}^{m} p_{ij}y_i \leq H(\sum_{i=1}^{m} p_{ij})c_j \quad \textit{for all } j \tag{8}$$

and such that

$$\sum_{i=1}^{m} y_i = \sum (c_j : \textit{where } o_j \in G) \tag{9}$$

for then

$$\begin{aligned}
\sum_{j=1}^{n} H(\sum_{i=1}^{m} p_{ij})c_j x_j &\geq \sum_{j=1}^{n}(\sum_{i=1}^{m} p_{ij}y_i)x_j \\
&= \sum_{i=1}^{m}(\sum_{j=1}^{n} p_{ij}x_j)y_i \\
&\geq q.\theta \sum_{i=1}^{m} y_i \\
&= q.\theta \sum (c_j : \textit{where } o_j \in G)
\end{aligned}$$

as desired.

The numbers $y_1, y_2, ..., y_m$ satisfying (8) and (9) have a simple intuitive interpretation: each y_i can be interpreted as the cost paid by MaxMargin for covering the keyword $q.\omega[i]$. We use cov_j^r to denote the coverage weight of query q covered by object o_j at the beginning of iteration r. Without loss of generality, we may assume that G is $\{o_1,o_2,...,o_r\}$ after r iteration, and so

$$\frac{cov_r^r}{c_r} \geq \frac{cov_j^r}{c_j}$$

for all r and j. If there are t iterations altogether then

$$\sum(c_j : j \in G) = \sum_{j=1}^{t} c_j,$$

and

$$y_i = \sum_{r=1}^{t} \frac{c_r \cdot cp(o_r, q.\omega[i])}{cov_r^r}.$$

We know that

$$\sum_{i=1}^{m} y_i = \sum_{i=1}^{m}\sum_{r=1}^{t} \frac{c_r \cdot cp(o_r, q.\omega[i])}{cov_r^r} = \sum_{r=1}^{t} c_r$$

For any o_j in RO_q, we know that the cov_j^r decrease as the iteration continues. We assume s is the largest superscript such that $cov_j^s > 0$ then

$$\begin{aligned}
\sum_{i=1}^{m} p_{ij} y_i &= \sum_{r=1}^{s} (cov_j^r - cov_j^{r+1}) \cdot \frac{c_r}{cov_r^r} \\
&\leq c_j \sum_{r=1}^{s} \frac{cov_j^r - cov_j^{r+1}}{cov_j^r} \\
&= c_j \sum_{r=1}^{s} \frac{cov_j^r - cov_j^{r+1}}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \frac{\lfloor cov_j^r + 1\rfloor - \lfloor cov_j^{r+1}\rfloor}{\lfloor cov_j^r + 1\rfloor} \\
&= c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1\rfloor}^{\lfloor cov_j^r + 1\rfloor} \frac{1}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1\rfloor}^{\lfloor cov_j^r + 1\rfloor} \frac{1}{l} \\
&= c_j H(\lfloor cov_j^1 + 1\rfloor) - c_j H(\lfloor cov_j^s + 1\rfloor) \\
&\leq c_j H(\lfloor cov_j^1 + 1\rfloor)
\end{aligned}$$

4.3. The Exact Algorithm MergeList

For many real application scenarios, the number of query keywords submitted by user is limited. Motivated by this observation, we devise an exact algorithm MergeList in this section.

The straightforward method for the exact algorithm is to enumerate all of the subsets of RO_q and return the subset, which covers the query keywords not less than a given threshold and with minimum cost, as our optimal solution. This yields an exponential time complexity in terms of the number of RO_q, which is also unacceptable for us.

To further prune the search space, we delve into several efficient pruning strategies below.

Firstly, we sort the objects of RO_q in ascending order of the cost distance. And we record the current optimal solution and the its cost with notation COS and $minCost$. Instead of enumerating all of the subsets of RO_q randomly, we construct the candidate subsets whose cost less than minCost by adding object into existing candidate subsets progressively.

Lemma 3. *Given a sorted RO_q which in ascending order of the cost distance. If for each existing candidate set **ecs**, the cost sum of ecs and current visitorial object **cvo** in RO_q is not less than minCost, then COS is our optimal solution.*

PROOF. *Since RO_q is sorted in ascending order of the cost distance. We know that each object behind in RO_q has a higher cost than cvo. If the cost sum of any ecs and cvo is not less than minCost, we can conclude that any ecs can not be the optimal solution, so we can terminate our procedure immediately.*

Secondly, there is a ***"Apriori property"*** in the data mining field. ***"All nonempty subsets of a frequent itemset must also be frequent"***. In the sequel, we present a similar pruning strategy.

Lemma 4. *If the cost sum of a ecs and cvo larger than the minCost, we can prune the ecs safely and any of its superset ecs need not to be computed.*

PROOF. *If the cost sum of a ecs and cvo larger than the minCost, we know that ecs cannot be the optimal solution and any superset of ecs neither can be the optimal solution, so we can prune them safely.*

Further, only if G is a feasible solution, we can filter out any superset G', since G is superior to G' anyway.

In a word, Lemma 3 allows us terminate the procedure earlier and Lemma 4 provides significant pruning ability. We elaborate the MergeList in algorithm 4.

The MergeList algorithm can be summarized as following three steps.

- Step 1 (Construct RO_q): In this step, we construct the RO_q and sort it in ascending order of cost distance.
- Step 2 (Add o into *ecs*): For object o in RO_q, we verify for each ecs whether delete it from candidate sets SS or combine it with o and put it into SS.
- Step 3 (Iterative step): Repeat Step 2, until the terminal condition in Lemma 3 is met.

As discussed above, in MergeList algorithm, we construct the candidate set by adding the object into ecs progressively. We use SS to store all the ecs, and initiate SS only with the empty set (line 6). For each object o in RO_q, if the cost of o not less than minCost then we terminate our procedure (lines 8-9). Otherwise, for each ecs satisfies Lemma 4, we prune it directly (lines 11-13). If *tempSet* is a feasible solution, we use it to update the COS and $minCost$ (lines 15-18), else we add it into SS.

Example 3: Consider a query q with two keywords $q.\omega = \{\lambda_1, \lambda_2\}$. Figure 5(a) illustrates the RO_q and the cost distance to q and the corresponding coverage weight. We illustrate the process of MergeList in Figure 5(b). There are total six steps to answer this query.

OID	o_1	o_2	o_3	o_4	o_5
Cost Distance to q	2	2.5	4	5	7
CP	0.1, 0.0	0.1, 0.3	0.3, 0.1	0, 0.3	0.1, 0.3

(a)

SID	Action	SS	minCost	COS
S_0	Initiation	$\emptyset$	$+\infty$	$\emptyset$
S_1	Visit o_1	$\emptyset$,$\{o_1\}$	$+\infty$	$\emptyset$
S_2	Visit o_2	$\emptyset$,$\{o_1\}$,$\{o_2\}$,$\{o_1, o_2\}$	$+\infty$	$\emptyset$
S_3	Visit o_3	$\emptyset$,$\{o_1\}$,$\{o_2\}$,$\{o_1, o_2\}$,$\{o_3\}$,$\{o_1, o_3\}$	6.5	$\{o_2, o_3\}$
S_4	Visit o_4	$\emptyset$,$\{o_4\}$	6.5	$\{o_2, o_3\}$
S_5	Visit o_5	$\emptyset$,$\{o_4\}$	6.5	$\{o_2, o_3\}$

(b)

Figure 5: An example of MergeList

- Step 1 (Initiation): We initiate the SS, $minCost$ and COS in this step.
- Step 2 (Visit o_1): Due to RO_q in Figure 5(a) has been sorted, we visit the object according to the order in Figure 5(a). We first visit o_1, and merge it with ecs in SS.
- Step 3 (Visit o_2): Object o_2 is merged with ecs in SS.
- Step 4 (Visit o_3): In this step, we obtain a feasible solution $COS = \{o_2, o_3\}$. And to filter candidate sets with Lemma 4.
- Step 5 (Visit o_4): In this step, ecs which satisfies Lemma 4 is pruned by COS.
- Step 6 (Visit o_5): Because the cost of o_7 is larger than $minCost$, so Lemma 3 is met and COS is the optimal solution.

Theorem 4. *(Correctness of MergeList): The MergeList algorithm always produces the correct result set.*

Proof. *Assuming the number of objects of RO_q is n, hence,there are up to $2^n - 1$ candidate sets. Each ecs either used to update the COS or pruned by the COS. If the cost of ecs less then minCost, we take ecs as our COS, otherwise, we prune ecs and any superset of ecs. Hence, it is suffice to show that MergeList never prune any feasible solution whose cost less than minCost (false negatives), and never maintain the ecs whose cost larger than minCost (false positives). So the MergeList algorithm always produce the correct result.*

5. EMPIRICAL STUDY

In this section, we experimentally study the performance of our propose algorithms through extensive experiments on both real and synthetic data sets. We describe the experimental settings in Section 5.1, and report the performance of our proposed algorithms for KaGPC queries on Synthetic and real data sets in Section 5.2 and 5.3 respectively.

5.1. *Experimental Setup*

Algorithms. We evaluate the performance of the following proposed algorithms: the approximate CubeTree in Section 4.1, the approximate MaxMargin in Section 4.2, and the exact algorithm MergeList in Section 4.3.

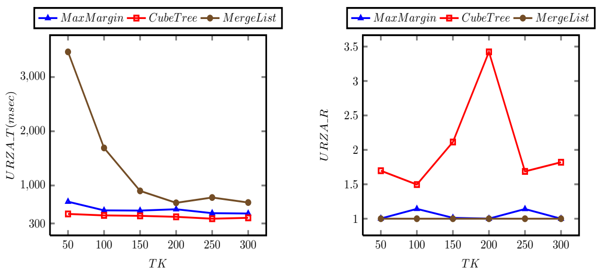
Data and queries. We conduct experiment with two datasets, namely synthetic dataset and real-life dataset.

Our experiments were conducted primarily on synthetic data sets to test the scalability and sensitivity of our algorithms to five major factors, namely, 1) *the total number of keywords in the spatial object database* (TK), 2) *data size* (DS), 3) *the number of query keywords* (QK), 4) *the upper bound of the number of keywords associated with each object* (KD), not the fixed number of keywords in [22], 5) *the threshold of query* q (TS). Synthetic data generator generates three types of data set following the uniform, random and zipf distribution respectively. We also included real data set CA, which was collected from the U.S. Board on Geographic Names(geonames.usgs.gov).

All three algorithms were implemented in C/C++ and run on a Intel(R) Core(TM)2 Quad CPU Q8400 @2.66Hz with 4GB RAM.

5.2. *Results on Synthetic Data Sets*

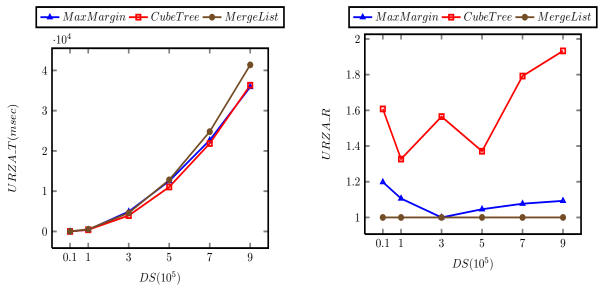
To measure the comprehensive performance of our algorithms with different types of data, in this section, we employ the URZ average response time ($URZA\_T$) and URZ average approximation ratio ($URZA\_R$) as the measurement, where $URZA\_T = \frac{T_u+T_r+T_z}{3}, URZA\_R = \frac{R_u+R_r+R_z}{3}$, Tu, Tr, Tz and Ru, Rr, Rz represent the response time and approximation ratio of uniform, random and zipf data sets, respectively. Table 5 illustrates all the possible values for the factors. Note that, we use bold font to denote the default value. We repeat query five times for each data set, and we take the average time as the final result.



(a) (b)

Figure 6: Effect of TK

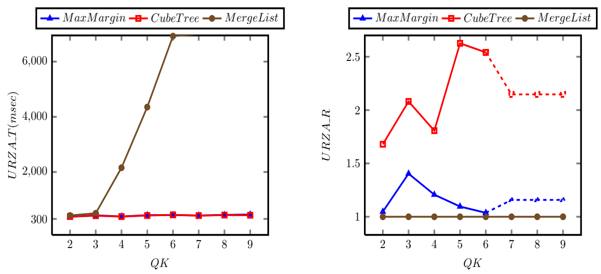
Studying on TK. Figure 6 shows the effect of TK on our algorithms. As can be seen from Figure 6(a), the response time of MergeList decreases dramatically, which denotes that MergeList is more sensitive to TK than CubeTree and MaxMargin. The reason is that under a fixed data size, the larger the TK, the smaller the number of RO_q. Figure 6(b) shows the $URZA\_R$ of our algorithms. In Figure 6(b), we set the approximation ratio of our exact algorithm MergeList equal to 1, which is also adopted by the experiment following. MaxMargin algorithm is superior to CubeTree in terms of approximation ratio. This is because MaxMargin always take the optimal object dynamically and update the contribution ratio.



(a) (b)

Figure 7: Effect of DS

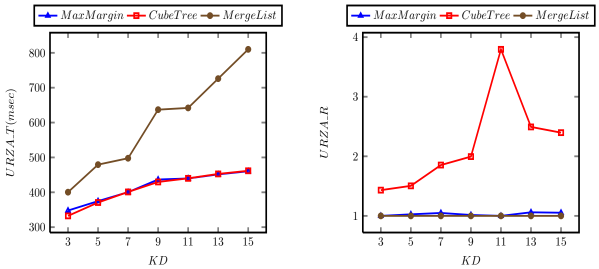
Studying on DS. The objective of this sub-experiment is to evaluate the influence of DS to the performance of our algorithms. We evaluate the influence of DS by testing data sets whose size range from 10,000 to 900,000. As can be seen from Figure 7(a) that all of the three algorithms scale well with the size of dataset. The exact algorithm achieves a nice response time due to the use of Lemma 3 and Lemma 4. Even though CubeTree outperforms MarMaxgin in terms of response time, however, Figure 7(b) shows that MaxMargin outperforms CubeTree in terms of approximation ratio and changes slightly.



(a) (b)

Figure 8: Effect of QK

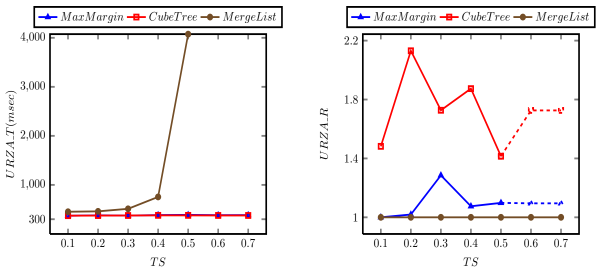
Studying on QK. In this sub-experiment, we vary the QK from 2 to 9. The experiment results are shown in Figure 8. Figure 8(a) shows the response time of three algorithms. We do not show the response time of algorithm if it runs out of memory (e.g., we omit the response time of MergeList in Figure 8(a)). We can see from Figure 8(a) that QK has little influence on both MaxMargin and CubeTree, and MergeList significantly increases in terms of response time due to the huge increasing of candidate sets to be computed. Figure 8(b) shows the approximation ratio of our algorithms. Due to out of memory, the results of MergeList when the QK equals to 7, 8 and 9 are missing. So the approximation ratio of CubeTree and MaxMargin cannot be computed when the QK equals to 7, 8 and 9. In this situation, to maintain the integrity of the experiment result curve, we take the average approximation ratio as the default value when QK equals to 7, 8 and 9, as can be seen in Figure 8(b), we show the default value with the dotted line. This strategy also be used for the experiment following.



(a) (b)

Figure 9: Effect of KD

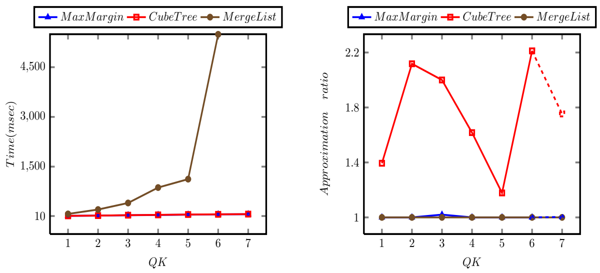
Studying on KD. In [22], zhang. studies the performance of KD by fixing the number of keywords for each object. In contrast, in our work, we take into account the real application scenarios. In real application scenarios, the number of keywords associated with object is random. We set a upper bound for KD in our work, which ranges from 3 to 15. For each object, we generate k keywords for it, where k is a random positive integer less than the upper bound. Figure 9(a) shows MaxMargin and CubeTree scale better than MergeList, both of CubeTree and MaxMargin change slightly. However, Figure 9(b) indicates the approximation ratio of CubeTree is unstable.



(a) (b)

Figure 10: Effect of TS

Studying on TS. We vary the TS from 0.1 to 0.7 in this sub-experiment. Figure 10(a) denotes that CubeTree and MaxMargin are adaptive to TS, however, the response time of MergeList increases dramatically as the value of TS is increased. As TS increases, much more objects are needed to reach the threshold specified by the query q, therefor the scale of candidate sets of MergeList significantly increases and can be pruned by



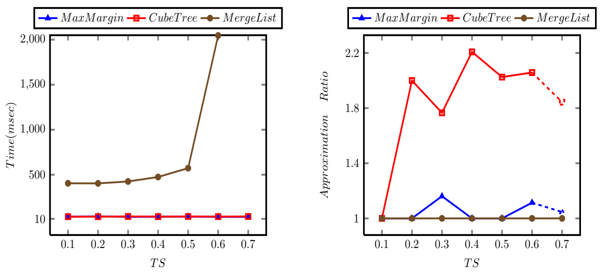
(a) (b)

Figure 11: Effect of QK

Lemma 4 efficiently, which results in the overflow of memory. Figure 10(b) shows a similar result with the approximation ratio result discussed above.

5.3. Results on Real Data Sets

In this section, we mainly study the response time and approximation ratio of our proposed algorithms on real data set GN, which was collected from the U.S. Board on Geographic Names(geonames.usgs.gov).



(a) (b)

Figure 12: Effect of TS

Each object of GN is a 2D location which is associated with a set of items describing it (e.g., a geographic name like Locate). We use the *feature class* as our keyword. Since there is only one unique Feature Class associated with each object, which is differ with our assumption that each object is associated with a few keywords. To address this problem, we combine these objects neighbouring, which can be organized as a set S, as a new object o. By specifying a threshold T, we can combine objects whose distance within T into a set S, we take the average coordinates of S as the new coordinates of o and the keyword set of S as the keyword set of o. For each keyword of o, we assign a integer number (e.g., 1,2,3,4,5) randomly for it as the level information. We take the real data set of California as our test data sets. Table 6 shows the details of this data set. Note that, the number of combined objects denotes the number of combined spatial objects that obtained by combining neighbouring objects.

Since the TK, DS and KD are fixed for real data set. We study the influence of QK and TS for real data set. Figure 11 and Figure 12 present the experiment result of QK and TS, respectively. Due to the experiment result of real data set is consistent with the result of synthetic data set, we do not analysis the results anymore. In the nutshell, similar with the result of synthetic data sets, CubeTree and MaxMargin run faster than MergeList by many times. And the CubeTree outperforms MaxMargin in terms of running time slightly. However, MaxMargin is superior to CubeTree in terms of approximation ratio. And at most of the time, MaxMargin returns result whose approximation ratio approaches to 1.

6. RELATED WORK

In this section, we mainly overview the existing work related to our KaGPC queries, focusing mostly on conventional spatial keyword queries and collective spatial keyword queries.

6.1. Conventional Spatial Keyword Queries

The conventional spatial keyword queries [6, 9] take a location and a set of keywords as arguments, and return objects that can satisfy the users needs solely. There are lots of efforts on conventional spatial keyword queries. By combining with existing queries in database community, there are several variants of conventional spatial keyword queries. We will review these queries as follows.

Combining with top-k queries. By combining with the top-k queries, the top-k spatial keyword queries [6, 10, 12, 16, 17, 19, 20] retrieve k objects with the highest ranking scores by utilizing the ranking function, which takes both location and the relevance of textual descriptions into consideration. To address the top-k spatial keyword queries efficiently, various hybrid indexes have been explored. This branch includes [6, 12] (IR-tree), [4] (SKI), [15, 16] (S2I). [7, 17] studies the top-k spatial keyword queries over trajectory data. Cong et al. [7] proposes the (Bck-tree) to facilitate the query processing of top-k trajectories. Wu et al. [19] handles the joint top-k spatial keyword queries utilizing the W-IR-Tree index. Zhang et al. [24] demonstrates that I^3 index, which adopts the Quadtree structure to hierarchically partition the data space into cells, is superior to IR-tree and S2I. Gao et al. [10] studies the reverse top-k boolean spatial keyword queries on the road network with count tree.

Combining with NN queries. The spatial keyword NN queries retrieve object that closed to the query location and contains the query keywords. Several variants has been explored. Tao et al. [18] develops a new access method called the SI-index to cope with multidimensional data, which overcomes the drawback of IR2-Tree [9]. Lu et al. [14] studies the RSTkNN query, finding the objects that take a specified query object as one of their k most spatial-textual similar objects.

Combining with route queries. The conventional route queries [11] in spatial database search the shortest route that starts at location s, passes through as least one object from each category in C and ends at t. Yao et al. [21] proposes the multi-approximate-keyword routing(MARK) query, which searches for the route with the shortest length such that it covers at least one matching object per keyword with the similarity larger than the corresponding threshold value. The problem of keyword-aware optimal route search (KOR) is studied in [2], to find the route which can cover a set of user-specified keywords, a specified budget constrain is satisfied, and an objective score of the route is optimal. Three algorithms are proposed for this problem in [2], and the corresponding system of KOR is provided in their subsequent work [1].

6.2. Collective Keyword Queries

All the works aforementioned return objects that can meet the users' needs solely. However, in real life applications, it is common to satisfy the users' needs collectively by a group of objects. The mCK queries [22, 23] return a set of objects to cover the query keywords. However, in the context of mCK, each object associates with only a single keyword and it only take keywords into consideration. The most similar works to ours is CoSKQ queries [3, 13]. With the maximum sum cost function, Cao et al. [3] provides approximation algorithms as well as an exact algorithm. To further improve the performance, Long et al. [13] proposes a distance owner-driven approach, besides they also propose a new cost measurement called diameter cost and design an exact and an approximate algorithm for it.

Although both CoSKQ and our queries retrieve a group of objects as result and take the keywords and location into consideration, however, our work differs with CoSKQ mainly in two aspects: 1) we take the useful level information of keywords into consideration, which is critical for user to make decision; 2) we take the combination of space distance and cost of object as our cost function, which is more closer to real life application scenarios. Due to these differences between CoSKQ and our KaGPC query, the methods adopted by CoSKQ can not be extended to solve our problem.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce and solve a novel type of queries, namely, KaGPC query. Although CoSKQ can retrieve a group of objects to collectively satisfy the users' needs, however, all of the exsiting works regardless of the useful level information of keywords, in this work, we take this kind of information into consideration. To address this novel problem, we design two approximation algorithms, namely, MaxMargin and CubeTree with provable approximation ratio. Besides, we also propose an exact algorithm MergeList for this problem. Extensive experiments with both real and synthetic data sets were conducted to verify the performance of our proposed algorithms.

In the future work, there are several interesting research directions. One is to research the KaGPC problem in the road network scenario. Another direction is to take multidimensional level information of keywords into consideration, which can provide the user more accuracy query result. It is also interesting to study other forms of cost function for this problem.

References

[1] X. Cao, L. Chen, G. Cong, J. Guan, N.-T. Phan, and X. Xiao. Kors: Keyword-aware optimal route search system. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1340–1343. IEEE, 2013.

[2] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *Proceedings of the VLDB Endowment*, 5(11):1136–1147, 2012.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 373–384. ACM, 2011.

[4] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *Scientific and Statistical Database Management*, pages 87–95. Springer, 2010.

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[6] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[7] G. Cong, H. Lu, B. C. Ooi, D. Zhang, and M. Zhang. Efficient spatial keyword search in trajectory databases. *arXiv preprint arXiv:1205.2880*, 2012.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[9] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *IEEE 24th International Conference on Data Engineering, 2008. (ICDE)*, pages 656–665. IEEE, 2008.

[10] Y. Gao, X. Qin, B. Zheng, and G. Chen. Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

[11] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases*, pages 273–290. Springer, 2005.

[12] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(4):585–599, 2011.

[13] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In *Proceedings of the 2013 ACM SIGMOD International conference on Management of data*, pages 689–700. ACM, 2013.

[14] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 349–360. ACM, 2011.

[15] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *Advances in Spatial and Temporal Databases*, pages 205–222. Springer, 2011.

[16] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *Proceedings of the 15th international conference on extending database technology (EDBT*, pages 168–179. ACM, 2012.

[17] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 156–167. ACM, 2012.

[18] Y. Tao and C. Sheng. Fast nearest neighbor search with keywords. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(4):878–888, 2014.

[19] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(10):1889–1903, 2012.

[20] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 541–552. IEEE, 2011.

[21] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in gis data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 201–210. ACM, 2011.

[22] D. Zhang, Y. M. Chee, A. Mondal, A. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *IEEE 25th International Conference on Data Engineering, 2009. ICDE'09.*, pages 688–699. IEEE, 2009.

[23] D. Zhang, B. C. Ooi, and A. Tung. Locating mapped resources in web 2.0. In *2010 IEEE 26th International Conference on Data Engineering (ICDE)*, pages 521–532. IEEE, 2010.

[24] D. Zhang, K.-L. Tan, and A. K. Tung. Scalable top-k spatial keyword search. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 359–370. ACM, 2013.

Algorithm 3: *MaxMargin*

Input : The KHT and the query q.

Output: A group G of objects as approximation solution.

```
G ⟵ ∅; maxMargin ⟵ 0.0; indicator ⟵ −1;
for i ⟵ 0 to |q.ω| − 1 do
    RV[i] ⟵ q.θ;
    construct the KMPQ PQ[i] for q.ω[i] with KHT;
r ⟵ 0;
while exists a dimension i satisfies RV[i] > 0.0 do
    for each valid PQ[j] ∈ PQ do
        o ⟵ PQ[j].top();
        if dcr_q^r(o) > maxMargin then
            maxMargin ⟵ dcr_q^r(o);
            indicator ⟵ j;
            maxOID ⟵ o;
    if maxOID ∈ G then
        PQ[indicator].dequeue();
        continue;
    if all dimensions satisfy maxOID.cv[i] < RV[i] then
        G ⟵ G ∪ {maxOID};
        for j ⟵ 0 to |q.ω| − 1 do
            if RV[j] ≥ maxOID.cv[j] then
                RV[j] ⟵ RV[j] − maxOID.cv[j];
            else
                RV[j] ⟵ 0;
                set the PQ[j] to be invalid;
        PQ[indicator].dequeue();
        r++;
    else
        for k ⟵ 0 to |q.ω| − 1 do
            if RV[j] < maxOID.cv[j] then
                maxOID.cv[j] ⟵ RV[j];
        recompute the dcr_q^r(maxOID);
        PQ[indicator].dequeue();
        PQ[indicator].enqueue(maxOID);
return G as the final solution;
```

Algorithm 4: *MergeList*

Input : The KHT and the query q.
Output: A group G of objects as resulting solution.

$COS \longleftarrow \emptyset$;
$SS \longleftarrow \emptyset$;
$minCost \longleftarrow$ INFINITE_MAX;
$RO_q \longleftarrow$ compute the relevant object set to query q with KHT;
sort RO_q in ascending order of weight distance cost;
put empty set Ø into SS;
for *each object* $o \in RO_q$ **do**
    if $wd(o, q) \geq minCost$ **then**
        break;
    for *each* $ecs \in SS$ **do**
        if *the cost sum of ecs and o not less than minCost* **then**
            delete ecs from SS;
            continue;
        $tempSet \longleftarrow ecs \cup \{o\}$;
        if *tempSet is a feasible solution* **then**
            $COS \longleftarrow tempSet$;
            $minCost \longleftarrow$ the cost of $tempSet$;
            delete ecs from SS;
        else
            put $tempSet$ into SS;
    if $SS ==$ Ø **then**
        break;
$G \longleftarrow COS$;
return G as the final solution;

Factors	Instance Value
TK	50,100,150,200,250,**300**
DS(10^4)	1,**10**,30,50,70,90
QK	2,**3**,4,5,6,7,8,9
KD	3,5,**7**,9,11,13,15
TS	0.1,**0.2**,0.3,0.4,0.5,0.6,0.7

Table 5: The real data of CA

Items Of CA	The Scale Of Items
Number of objects (or keywords)	2761823
Number of unique keywords	63
Number of combined objects	20694

Table 6: The real data of CA