Collective Spatial Keyword Querying With Auxiliary Information

Pengfei Zhang
Zhejiang University, China
zpf_2013zb@zju.edu.cn

Huaizhong Lin
Zhejiang University, China
linhz@zju.edu.cn

Dongming Lu
Zhejiang University, China
ldm@zju.edu.cn

ABSTRACT

Spatial keyword queries have received greatly attention recently due to the proliferation of spatial data. Given a location and a set of query keywords, collective spatial keyword query (CoSKQ) returns a group of objects which collectively cover given query keywords and have the smallest cost. However, CoSKQ only retrieves the objects that can cover query keywords, but without taking the auxiliary information (e.g., the level of attractions), which is critical for user to make decision, of keywords into consideration.

To tackle the spatial keyword query with auxiliary information of keyword, Keyword-aware group probability coverage query (KaGPC) is proposed in this work. KaGPC retrieves a group G of objects that collectively meet the user needs: 1) the probability that each query keyword covered by G not less than a given threshold θ; 2) the cost function is minimized. We map KaGPC query to the classic Weighted Set Cover (WSC) Problem, denoting that it is NP-hard. With the notion of contribution ratio, we design two approximation algorithms with provable approximation ratio and an exact algorithm. A comprehensive empirical evaluation over real and synthetic data sets demonstrates the effectiveness of our proposed algorithms.

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 [**?**, **?**, **?**, **?**], 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 [**?**, **?**] 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 **??** 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 threshold within a limited lifespan.

To address this problem, we propose a novel query called

OID	PX	PY	*Associated 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

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

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 [**?**]. 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 **??** with the objects presented in Table **??**. 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.

- 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.

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.

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 [**?**, **?**], 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:

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

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 $\lambda \in q.\omega$, $cov(G, \lambda) \geq q.\theta$;
- $\arg\min_{G} \sum_{o \in G} cd(o, q)$.

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.

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

In most cases, given a query q, we only need to visit a fraction of objects in that most of the objects have nothing to do with this query. Motivated by this observation, we introduce the pre-processing phase. We use the pre-processing result to accelerate the algorithms to be proposed. The task of the pre-processing is to construct the keywords inverted index structure which is organized as a hash table with ***perfect hashing technique***. For brevity, we will refer to them as KHT.

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

Keywords	OList	First Index	Second Index
mountain	o_1, o_3, o_7, o_{10}	5	0
forest	o_3, o_9	6	1
landscape	o_1, o_7	6	2
shore	o_2, o_4, o_5	6	1
temple	o_1, o_3, o_6	1	0
museum	o_2, o_7	2	0
architecture	o_5, o_6	5	1
drigtate	o_5, o_{10}	0	0
glacier	o_8, o_{10}	4	0

Table 4: KHT entries

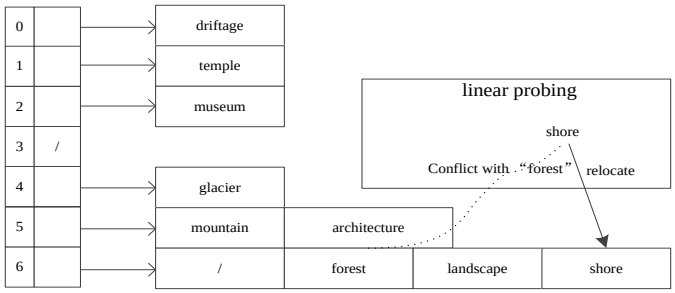


Figure 1: The KHT instance

- 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 $(\lambda, \lambda.olist)$, where λ represents a keyword and $\lambda.olist$ is the RO_λ. Table **??** elaborates the KHT entries, and Figure **??** shows the KHT of Table **??**. Since the limitation of space, we omit the olist for each entry and only show the keyword part in Figure **??**.

We map each distinct keyword λ into KHT with the two-levels hash function, the first level hash function generates a first index *fi* for each distinct keyword. For keywords possessing the identical first index, we utilize the second level hash function to specify a second index *si* for each of them. Note that, although the probability that two distinct keywords with the same fi and si is less than 0.5, which is demonstrated in [**?**], yet we should take this situation into consideration. To address the conflict problem, ***"linear probing"*** technique is applied in our work. From Table **??** we know that there is a conflict between "forest" and "shore". As illustrated in Figure **??**, when we try to insert the entry of "shore" into KHT, there is a entry "forest" in location $(6, 1)$, due to location $(6, 2)$ is occupied as well, with "linear probing", "shore" 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

We present the approximation algorithm CubeTree with provable approximation ratio in 4.1, in section 4.2 we show another approximation algorithm MaxMargin with better accuracy, finally, the exact algorithm MergeList is elaborated in section 4.3.

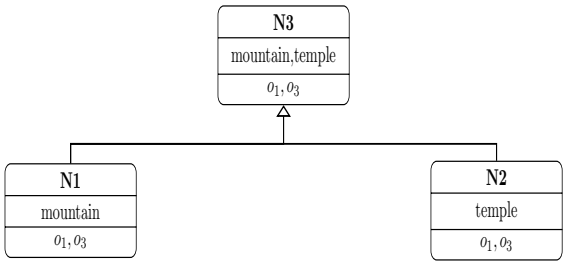


Figure 2: The cubeTree of query q of Table ??.

4.1 Approximation Algorithm CubeTree

Similar with the strategy of dynamical 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 following definition which will lay the foundation for the CubeTree algorithm.

Definition 5. (keywords cube) Given a query q and 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 that is in the descending order of *cr* and satisfies the following three conditions:

- For each keyword $\lambda \in \kappa$, $cov(Cube_\kappa, \lambda) \geq q.\theta$;
- For any $o_i \in Cube_\kappa$, $o_j \in RO_\kappa - Cube_\kappa$, $cr(o_i, q) \geq cr(o_j, q)$;
- There is not a subset of $Cube_\kappa$ satisfies the above two conditions.

Note that for an object o, it may appear in a few different keywords cubes. For brevity, we refer to keywords cube as cube. For ease to distinguish, we use cubeTree to denote the tree structure used in this section, and CubeTree as our algorithm.

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

The CubeTree algorithm builds the cubeTree in a bottom-up manner. For k query keywords, there are total $2^k - 1$ possible nodes to be computed, which is time consuming. We adopt a strategy which can balance the computation overhead and the approximation bound. In this paper, instead of computing all of the nodes, we compute a fraction of the them as follows.

In a nutshell, 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 this way, 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.* □

This lemma inspires us to combine two node with a simple strategy. In each iteration, we add the object with maximal cr into the cube of higher-level node instead of adding the object with maximal coverage probability.

We summary our discussion above as following three steps:

- Step 1 (Construct the bottom node): Construct the bottom node of cubeTree with KHT.
- Step 2 (Combine the low-level nodes): In this step, we retrieve 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 node. Then we take the cube of root node as our final solution.

Algorithm 1: *CubeTree*

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

```
1  G ⟵ ∅;
2  lcID ⟵ 0; rcID ⟵ 0; cID ⟵ 0;
3  for i ⟵ 0 to |q.ω| − 1 do
4      cID ⟵ 2^i;
5      κ ⟵ q.ω[i];
6      construct the Cube_κ of keywords set κ with
       KHT;
7      put the node (cID, κ, Cube_κ) into cubeTree;
8  for i ⟵ 2 to |q.ω| do
9      for j ⟵ 0 to |q.ω| − i − 1 do
10         if j == 0 then
11             seq ⟵ 2^i − 1;
12         else
13             seq ⟵ seq × 2;
14         cID ⟵ seq;
15         lcID ⟵ seq − 2^(i+j−1);
16         rcID ⟵ seq − 2^j;
17         κ ⟵ cubeTree[lcID].κ ∪ cubeTree[rcID].κ;
18         cubecID ⟵ combineCube(lcID, rcID);
19         put the node (cID, κ, cubecID) into
           cubeTree;
20 solutionID ⟵ 2^(|q·ω|−1);
21 G ⟵ the cube of cubeTree[solutionID];
22 return G as the final solution;
```

The pseudocode of CubeTree is outlined in Algorithm **??**. CubeTree takes a bottom-up manner to construct the cubeTree. CubeTree contains 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 retrieve 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 **??** elaborates the process of combining phase. We use *oidl* and *oidr* to record the current optimal object of $cubeTree[lcID]$ and $cubeTree[rcID]$ respectively (lines 4-5). If the *oidl* has larger *cr* than *oidr*, we test 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 next object. Otherwise, we add it into $cubecID$ and update the coverage probability 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 **??**.

Algorithm 2: *combineCube*

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

```
1  cubecID ⟵ ∅;
2  κ ⟵ cubeTree[lcID].κ ∪ cubeTree[rcID].κ;
3  while exists λ ∈ κ and cov(cubecID, λ) < q.θ do
4      oidl ⟵ the oid with maximal cr in the cube of
       cubeTree[lcID];
5      oidr ⟵ the oid with maximal cr in the cube of
       cubeTree[rcID];
6      if cr(oidl, q) ≥ cr(oidr, q) then
7          if oidl ∈ cubecID then
8              delete oidl from the cube of
               cubeTree[lcID];
9              continue;
10         else
11             put oidl into cubecID;
12             delete oidl from the cube of
               cubeTree[lcID];
13             for each λ ∈ κ do
14                 update the cov(cubecID, λ);
15     else
16         if oidr ∈ cubecID then
17             delete oidr from the cube of
               cubeTree[rcID];
18             continue;
19         else
20             put oidr into the cubecID;
21             delete oidr from the cube of
               cubeTree[rcID];
22             for each λ ∈ κ do
23                 update the cov(cubecID, λ);
24 return cubecID;
```

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

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

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

and meanwhile,

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

*holds. Combining inequalities (**??**) and (**??**), we know that*

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

□

4.2 Approximation Algorithm MaxMargin

In this section, we elaborate the MaxMargin algorithm, which is inspired by the greedy strategy adopted by the WSC Problem [**?**]. For the WSC problem, the greedy strategy iteratively retrieves the current optimal subset and updates subsets that have not been visited. We can extend this strategy by iteratively selecting the object with maximal cr, and updating objects that have not been visited for our problem. The basic version of MaxMargin solves the problem in four steps with this strategy. We utilize G to reserve the solution.

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

Clearly, there are two major drawbacks degrade the performance of basic version: 1) lacking of efficient pruning strategy; 2) the Step 3 is time consuming in that all of the remaining objects to be updated in each iteration. To further boost the performance, we take a look at several notions which overcome aforementioned drawbacks.

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_q.

Each entry of the form (nid, oid, cr, cv), where nid and oid correspond to the identifier of entry and the object, respectively. cr represents the contribution ratio of oid and cv is a $|q.\omega|$ dimensions contribution vector to record the $cp(o, \lambda)$ 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 boosts the performance.

To address another drawback, we extend 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:

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

For dcr, we replace $cov(o, q)$ with $cov^r(o, q)$, where $cov^r(o, q)$ denotes the current contribution ratio of o after r time iterations.

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

PROOF. *After each iteration, the $dcr_q(o)$ will be updated due to the decreasing of $cov(o,q)$, which results in the decreasing of $dcr_q(o)$. If $m \leq n$, we conclude 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 2 guarantees we can always retrieve the current optimal object without updating the objects frequently.

With the aforementioned discussion, we present the major idea of the enhanced MaxMargin as follows. And we take G to reserve the final solution and use 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 has been in G, then continue to select next optimal object. Otherwise, we add o into G if $cov(o,q)$ less than RV for all the dimensions, else we update the cr of o and reinsert o into $KMPQ$.
- Step 3 (Iterative step): Repeat Step 2 until RV equals to 0. Then we take G as our final solution.

Algorithm **??** illustrates the detail of our MaxMargin algorithm.

MaxMargin utilizes the residual vector RV to record the difference between $q.\ell$ and the coverage probability of G for each dimension dynamically. And we initiate each dimension of RV to $q.\theta$. With RV, MaxMargin can update the *dcr* of o immediately once the coverage probability of o larger than RV in some dimensions. MaxMargin iteratively selects the optimal object o from PQ (lines 7-12). If o belongs to G, we continue to select next optimal object. Otherwise, we test whether to update the $dcr(o)$ (line 16) or not. If the contribution vector of $maxOID$ not larger than RV in all the dimensions, we add $maxOID$ into G and update the RV (lines 17-25), otherwise, we update the dcr of $maxOID$ and the PQ[indicator]. The algorithm terminates until RV equals to 0. Note that, in Algorithm **??**, 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 probability of o larger than RV in some dimensions instead of updating the remaining RO_q in basic version. Both of these two strategies significantly boost our MaxMargin algorithm.

Example 2: Lets go back to the KaGPC query q depticts in Table **??**. As illustrated in Figure **??**, 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 depticted in Figure **??**(a) and **??**(b).
- 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 **??**(c).
- Step 3 (Delete o_3): As depticted in Figure **??**(c) and **??**(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 **??**(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.

Algorithm 3: *MaxMargin*

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

```
1  G ⟵ ∅; maxMargin ⟵ 0.0; indicator ⟵ −1;
2  for i ⟵ 0 to |q.ω| − 1 do
3      RV[i] ⟵ q.ℓ;
4      construct the KMPQ PQ[i] for q.ω[i] with KHT;
5  r ⟵ 0;
6  while exists a dimension i satisfies RV[i] > 0.0 do
7      for each valid PQ[j] ∈ PQ do
8          o ⟵ PQ[j].top();
9          if dcr^r(o) > maxMargin then
10             maxMargin ⟵ dcr_q^r(o);
11             indicator ⟵ j;
12             maxOID ⟵ o;
13     if maxOID ∈ G then
14         PQ[indicator].dequeue();
15         continue;
16     if each dimension i satisfies
       maxOID.cv[i] < RV[i] then
17         G ⟵ G ∪ maxOID;
18         for j ⟵ 0 to |q.ω| − 1 do
19             if RV[j] ≥ maxOID.cv[j] then
20                 RV[j] ⟵ RV[j] − maxOID.cv[j];
21             else
22                 RV[j] ⟵ 0;
23                 set the PQ[j] to be invalid;
24         PQ[indicator].dequeue();
25         r++;
26     else
27         for k ⟵ 0 to |q.ω| − 1 do
28             if RV[j] < maxOID.cv[j] then
29                 maxOID.cv[j] ⟵ RV[j];
30         recompute the dcr_q^r(maxOID);
31         PQ[indicator].dequeue();
32         PQ[indicator].enqueue(maxOID);
33 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 [**?**], we provide the approximation ratio proof of MaxMargin here. We use m, n to denote the number of $|q.\omega|$ and $|RO_q|$ respectively. We define a $m \times n$ matrix $P = (p_{ij})$ by*

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

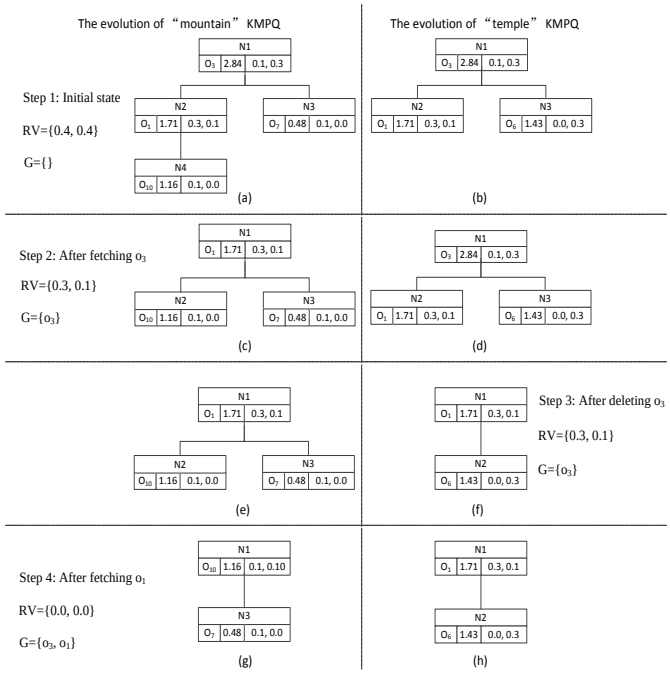


Figure 3: KMPQ for query q of Table ??

According to the definition of P, we know that the n columns of P is n coverage probability 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:

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

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

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

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

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

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

and such that

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

for then

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

as desired.

The numbers $y_1, y_2, ..., y_m$ satisfying (??) and (??) 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 probability 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

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

A simple idea of an 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 minimal weight cost, as our optimal solution. This yields an exponential running time in terms of the number of RO_q, which is also unacceptable for us.

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

Firstly, we sort the objects of RO_q in ascending order of the weight distance cost. 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 weight distance cost. 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 weight distance cost. We know that each object behind 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. That is ***"all nonempty subsets of a frequent itemset must also be frequent"***. In the following, 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 compute.*

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 need not to construct any superset G'. Since, G is superior to G' anyway.

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

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 weight distance cost.
- Step 2 (Add o into *ecs*): For object o in RO_q, we test 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 Lemma 3 is met.

Algorithm 4: *MergeList*

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

```
1  COS ⟵ ∅;
2  SS ⟵ ∅;
3  minCost ⟵ INFINITE_MAX;
4  RO_q ⟵ compute the relevant object set to query q
   with KHT;
5  sort RO_q in ascending order of weight distance cost;
6  put empty set ∅ into SS;
7  for each object o ∈ RO_q do
8      if wd(o, q) ≥ minCost then
9          break;
10     for each ecs ∈ SS do
11         if the cost sum of ecs and o not less than
           minCost then
12             delete ecs from SS;
13             continue;
14         tempSet ⟵ ecs ∪ o;
15         if tempSet is a feasible solution then
16             COS ⟵ tempSet;
17             minCost ⟵ the cost of tempSet;
18             delete ecs from SS;
19         else
20             put tempSet into SS;
21     if SS == ∅ then
22         break;
23 G ⟵ COS;
24 return G as the final solution;
```

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

OID	o_1	o_2	o_3	o_4	o_5
WD cost 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 4: An example of MergeList

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 **??**(a) illustrates the RO_q and the weight distance cost to q and the corresponding coverage probability. We illustrate the process of MergeList in Figure **??**(b). There are total six steps to answer this query.

- Step 1 (Initiation): We initiate the SS, *minCost* and COS in this step.
- Step 2 (Visit o_1): Due to RO_q in Figure **??**(a) has been sorted, we visit the object according to the order in Figure **??**(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 total $2^n - 1$ candidate sets. Every 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, its 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

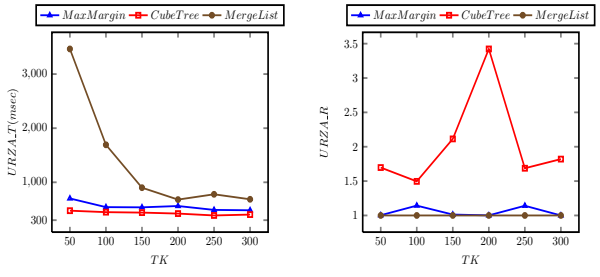
In this section, we study the performance of our proposed three algorithms, namely CubeTree, MaxMargin and MergeList with both real and synthetic data sets in two aspects, namely, response time and approximation ratio. All three algorithms were implemented in C/C++ and run on a Intel(R) Core(TM)2 Quad CPU Q8400 @2.66Hz with 4GB RAM.

5.1 Experiments on Synthetic Data Sets

We study the performance of our three algorithms in synthetic data sets with five parameters, the TK, DS, QK, KD and TS. To study the performance of algorithms in data sets following different distribution, for each parameter, the synthetic data generator generates three types of data set following the uniform, random and zipf distribution respectively. It is worth noting that we utilize the URZ average response time ($URZA\_T$) and URZ average approximation ratio ($URZA\_R$) to measure the performance, instead of showing the experiment results of random, uniform and zipf respectively. There are three major factors account for this strategy: 1) If we show our experiments respectively, it will take up a huge space, which significantly weakens the presentation for other parts; 2) By taking average results we can study the performance of our algorithms better from a global perspective, without being disturbed by exceptional case occurred in a special distribution data set; 3) Last but not least, the average results can reflect the general performance and is convenient for us to evaluate the algorithms. $URZA\_T = \frac{Tu+Tr+Tz}{3}, URZA\_R = \frac{Ru+Rr+Rz}{3}$, where Tu, Tr, Tz and Ru, Rr, Rz represent the response time and approximation ratio of uniform, random and zipf data sets, respectively. In the following, we take a look at these five parameters aforementioned.

- TK: the total number of keywords in the spatial object database.
- DS: the data size.
- QK: the number of query keywords.
- KD: the upper bound of the number of keywords associated with each object, instead of fixing the number of keywords in [**?**].
- TS: the threshold of query q.

In this part, we adjust the five parameters to verify the performance of our algorithms, and we set the default value for each parameter as TK(300), DS(10^4), QK(3), KD(7) and TS(0.2). Note that, we only adjust one parameter and others take the default value for each sub-experiment.

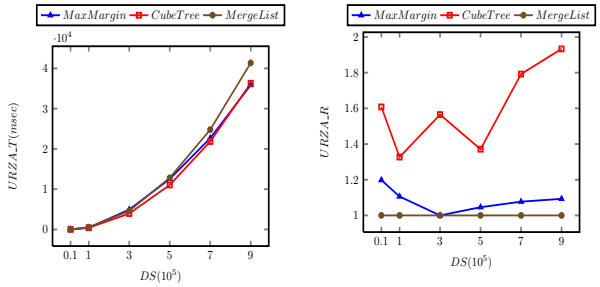


(a) (b)

Figure 5: Effect of TK

Studying on TK. To study the effect of TK. We increase the TK from 50 to 300. Figure **??** shows the results of our algorithms. Clearly, Figure **??**(a) shows that with the increasing of TK, the average response time of our algorithms

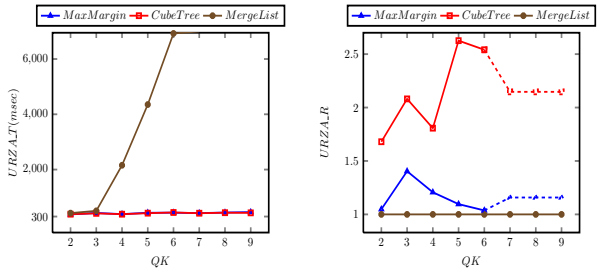
decreases. The reason accounts for this phenomenon is that under a fixed data size, the larger the TK, the smaller the number of RO_q. The response time of MergeList decreases drastically, which denotes that MergeList is more sensitive to TK than CubeTree and MaxMargin. Figure **??**(b) shows the $URZA\_R$ of our algorithms. In Figure **??**(b), we keep the approximation ratio of our exact algorithm MergeList always equal to 1, which is also adopted by the Figure following. MaxMargin algorithm achieves a nicer approximation ratio than CubeTree. This is because MaxMargin always take the optimal object dynamically and update the contribution ratio, that is the reason why MaxMargin slower than CubeTree in Figure **??**(a).



(a) (b)

Figure 6: Effect of DS

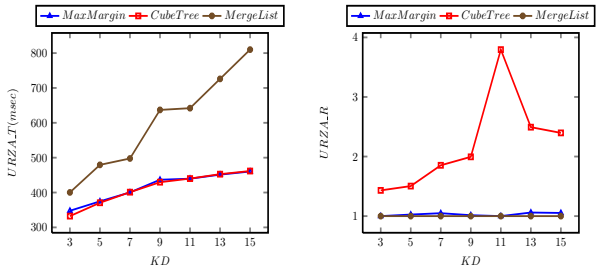
Studying on DS. The objective of this sub-experiment is to evaluate the influence of DS to the performance of our algorithms. We evaluate the influence of DS by testing data sets whose size range from 10,000 to 900,000. As can be seen from Figure **??**(a) that all of the three algorithms scale well with the size of dataset and the growth approaches to linear. 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 **??**(b) shows that MaxMargin outperforms CubeTree in terms of approximation ratio and changes slightly.



(a) (b)

Figure 7: Effect of QK

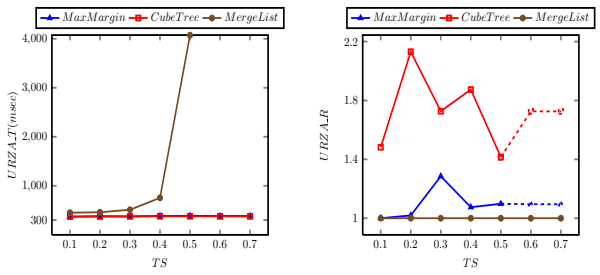
Studying on QK. In this sub-experiment, we vary the QK from 2 to 9. The experiment results are shown in Figure **??**. In Figure **??**(a), we shows the response time of three algorithms. For MergeList, we do not show the response time of algorithm if it runs out of memory (e.g., we omit the response time of MergeList in Figure **??**(a)). We can see from Figure **??**(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 **??**(b) shows the approximation ratio of our algorithms. Due to we lack the experiment result of MergeList when the QK equals to 7, 8 and 9. So the approximation ratio of CubeTree and MaxMargin can not 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 **??**(b), we show the default value with the dotted line. These strategies also can be used for the Figure following.



(a) (b)

Figure 8: Effect of KD

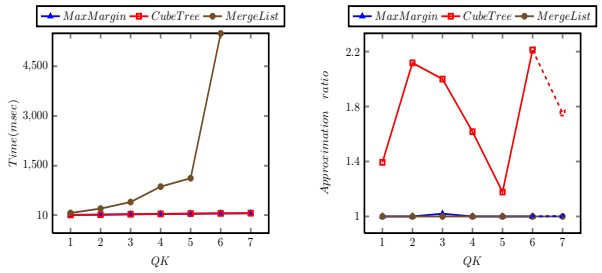
Studying on KD. In [**?**], zhang. study the performance of KD by set a fixed number of keywords for each object. Differing with it, in our work, we take the real application scenarios into consideration. In real application scenarios, the number of keywords associated with object is random. We set a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



(a) (b)

Figure 9: Effect of TS

Studying on TS. We vary the TS from 0.1 to 0.7 in this sub-experiment. Figure **??**(a) denotes that CubeTree and MaxMargin have a good adaption to TS, however, the response time of MergeList increases drastically as TS increases. When TS increases, much more objects needed to reach the threshold specified by the query q, therefor the scale of candidate sets of MergeList significantly increases



(a) (b)

Figure 10: Effect of QK

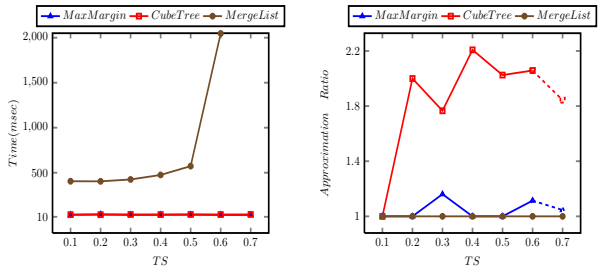
and can be pruned by Lemma 4 efficiently, which results in the overflow of memory. Figure **??**(b), shows a similar result with the approximation ratio result discussed above.

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

Table 5: The real data of CA

5.2 Experiments on Real Data Sets

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



(a) (b)

Figure 11: Effect of TS

Each object in 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 overcome this problem, we combine these objects neighbouring, which can be organized as a set S, as one new object o. By specifying a threshold T, we can combine objects whose distance not larger than 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 auxiliary information. We take the real data set of California as our test data sets. Table **??** 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 **??** and Figure **??** present the experiment result of QK and TS, respectively. Due to the experiment reslut of real data set is similar 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 [**?**, **?**] 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 a few 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**] 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 [**?**, **?**] (IR-tree), [**?**] (SKI), [**?**, **?**] (S2I). [**?**, **?**] study the top-k spatial keyword queries over trajectory data. Cong et al. [**?**] proposes the (Bck-tree) to facilitate the query processing of top-k trajectories. Wu et al. [**?**] handles the joint top-k spatial keyword queries utilizing the W-IR-Tree index. Zhang et al. [**?**] 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. [**?**] 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 of the query keywords. It has a few variants. Tao et al. [**?**] develops a new access method called the SI-index to cope with multidimensional data, which overcomes the drawback of IR2-Tree [**?**]. Lu et al. [**?**] 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 query [**?**] 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. [**?**] 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 [**?**], 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 [**?**], and the corresponding system of KOR is provided in their subsequent work [**?**].

6.2 Collective Keyword Queries

All the works aforementioned return objects that can meet the user needs solely. However, in real life applications, it is common to satisfy the users needs collectively by a group of objects. The mCK queries [**?**, **?**] 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 [**?**, **?**]. With the maximum sum cost function, Cao et al. [**?**] provides approximation algorithms as well as an exact algorithm. To further improve the performance, Long et al. [**?**] 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 auxiliary information of keywords into consideration, which is critical for user to make decision; 2) we take the combination of space distance and auxiliary information of keywords 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 user needs, however, all of the exiting works regardless of the useful auxiliary 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 auxiliary 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.