A FRAMEWORK FOR PROCESSING K-BEST SITE QUERY

Yuan-Ko Huang* and Lien-Fa Lin

Department of Information Communication Kao-Yuan University;
Kaohsiung Country, Taiwan R.O.C.

ABSTRACT

A novel query in spatial databases is the K-Best Site Query (KBSQ for short). Given a set of objects O, a set of sites S, and a user-given value K, a KBSQ retrieves the K sites from S such that the total distance from each object to its closest site is minimized. The KBSQ is indeed an important type of spatial queries with many real applications. In this paper, we investigate how to efficiently process the KBSQ. We first propose a straightforward approach with a cost analysis, and then develop the K Best Site Query (KBSQ) algorithm combined with the existing spatial indexes to improve the performance of processing KBSQ. Comprehensive experiments are conducted to demonstrate the efficiency of the proposed methods.

KEYWORDS

spatial databases; K-Best Site Query; spatial indexes

1. INTRODUCTION

With the fast advances of positioning techniques in mobile systems, spatial databases that aim at efficiently managing spatial objects are becoming more powerful and hence attract more attention than ever. Many applications, such as mobile communication systems, traffic control systems, and geographical information systems, can benefit from efficient processing of spatial queries [1-7]. In this paper, we present a novel and important type of spatial query, namely the *K-Best Site Query* (*KBSQ* for short). Given a set of objects O, a set of sites S, and a user-given value K, a *KBSQ* retrieves the K sites $s_1, s_2, ..., s_K$ from S such that $\sum_{o_i \in O} d(o_i, s_j)$ is minimized, where $d(o_i, s_j)$ refers to the distance between object o_i and its closest site s_j. We term the sites retrieved by executing the *KBSQ* the *best sites* (or *bs* for short).

The *KBSQ* problem arises in many fields and application domains. As an example of real-world scenario, consider a set O of soldiers on the battlefields that is fighting the enemy. In order to immediately support the injured soldiers, we need to choose K sites from a set S of sites to build the emergicenters. Note that there are many soldiers fighting on the battlefields and many sites could be the emergicenters. To achieve the fastest response time, the sum of distances from each battlefield to its closest emergicenter should be minimized. Another real-world example is that the McDonald's Corporation may ask "what are the optimal locations in a city to open new McDonald's stores." In this case, the *KBSQ* can be used to find out the K best sites among a set S of sites so that every customer in set O can rapidly reach his/her closest store.

Let us use an example in Figure 1 to illustrate the *KBSQ* problem, where six objects o_1, o_2, ..., o_6 and four sites s_1, s_2, ..., s_4 are depicted as circles and rectangles, respectively. Assume that two best sites (i.e., 2*bs*) are to be found in this example. There are six combinations (s_1, s_2), (s_1, s_3), ..., (s_3, s_4), and one combination would be the result of *KBSQ*. As we can see, the sum of distances from objects o_1, o_2, o_3 to their closest site s_3 is equal to 9, and the sum of distances between objects o_4, o_5, o_6 and site s_1 is equal to 12. Because the combination (s_1, s_3) leads to the minimum total distance (i.e., 9 + 12 = 21), the two sites s_1 and s_3 are the 2*bs*.

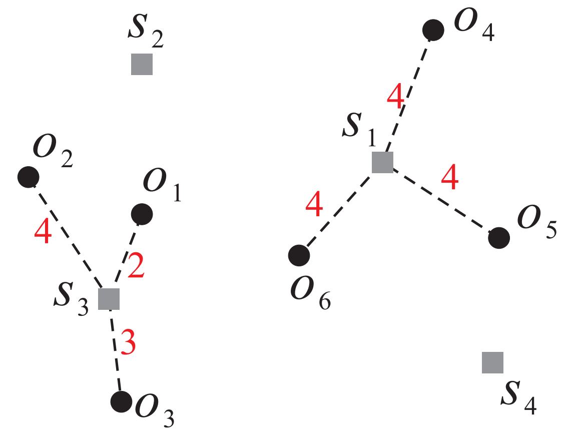

Figure 1. An example of the *KBSQ*

To process the *KBSQ*, the closest site for each object needs to be first determined and then the distance between object and its closest site is computed so as to find the best combination of *K* sites. When a database is large, it is crucial to avoid reading the entire dataset in identifying the *K* best sites. For saving CPU and I/O costs, we develop an efficient method combined with the existing spatial indexes to avoid unnecessary reading of the entire dataset. A preliminary version of this paper is [8], and the contributions of this paper are summarized as follows.

- We present a novel query, namely the *K* Best Site Query, which is indeed an important type of spatial queries with many real applications.
- We propose a straightforward approach to process the *KBSQ* and also analyze the processing cost required for this approach.
- An efficient algorithm, namely the *K Best Site Query* (*KBSQ*) algorithm, operates by the support of R*-tree [9] and Voronoi diagram [10] to improve the performance of *KBSQ*.
- A comprehensive set of experiments is conducted. The performance results manifest the efficiency of our proposed approaches.

The rest of this paper is organized as follows. In Section 2, we discuss some related works on processing spatial queries similar to the *KBSQ*, and point out their differences. In Section 3, the straightforward approach and its cost analysis is presented. Section 4 describes the *KBSQ* algorithm with the used indexes. Section 5 shows extensive experiments on the performance of our approaches. Finally, Section 6 concludes the paper with directions on future work.

2. Related Work

In recent years, some queries similar to the *KBSQ* are presented, including the Reverse Nearest Neighbor Query (*RNNQ*) [11], the Group Nearest Neighbor Query (*GNNQ*) [12], and the Min-Dist Optimal-Location Query (*MDOLQ*) [13]. Several methods have been designed to efficiently process these similar queries. However, the query results obtained by executing these queries are quite different from that of the KBSQ. Also, the proposed methods cannot be directly used to answer the KBSQ. In the following, we investigate why the existing methods for processing the similar queries cannot be applied to the KBSQ separately.

2.1. Methods For RNNQ

Given a set of object O and a site s, a *RNNQ* can be used to retrieve a set S of objects contained in O whose closest site is s. Each object o in S is termed a *RNN* of s. An intuitive way for finding the query result of *KBSQ* is to utilize the *RNNQ* to find the *RNNs* for each site. Then, the K sites having the maximum number of *RNNs* (meaning that they are closer to most of the objects) are chosen to be the K best sites.

Taking Figure 2 as an example, the *RNNs* of site s_1 can be determined by executing the *RNNQ* and its *RNNs* are objects o_4 and o_6. Similarly, the *RNN*s of sites s_2, s_3, and s_4 are determined as o_1 and o_2, o_3, and o_5, respectively. As sites s_1 and s_2 have the maximum number of *RNNs*, they can be the 2*bs* for the *KBSQ*. However, sites s_1 and s_2 lead to the total distance 24 (i.e., $d(o_4, s_1) + d(o_5, s_1) + d(o_6, s_1) + d(o_1, s_2) + d(o_2, s_2) + d(o_3, s_2)$), which is greater than the total distance 22 as sites s_1 and s_3 are chosen to be the 2*bs*. As a result, the intuition of using the *RNNQ* result to be the *KBSQ* result is infeasible.

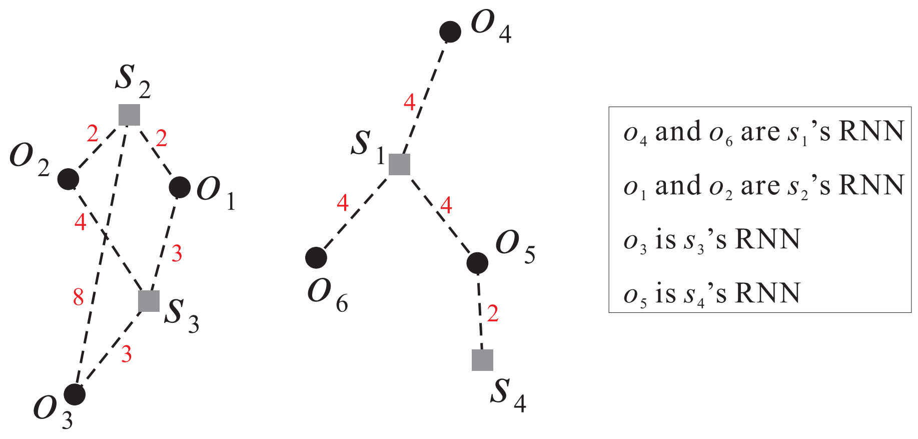

Figure 2. An example of the *RNNQ*

2.2. Methods For GNNQ

A *GNNQ* retrieves a site s from a set of sites S such that the total distance from s to all objects is the minimum among all sites in S. Here, the result s of *GNNQ* is called a *GNN*. To find the K best sites, we can repeatedly evaluate the *GNNQ* K times so as to retrieve the first K *GNNs*. It means that the sum of distances between these K *GNNs* and all objects is minimum, and thus they can be the K *bs*. However, in some cases the result obtained by executing the *GNNQ* K times is still different from the exact result of KBSQ.

Let us consider an example shown in Figure 3, where 2*bs* are required. As shown in Figure 3(a), the first and second *GNNs* are sites s_3 and s_1, respectively. As such, the 2*bs* are s_3 and s_1, and the total distance $d(o_1, s_1) + d(o_2, s_1) + d(o_4, s_1) + d(o_3, s_3) + d(o_5, s_3) + d(o_6, s_3) = 23$. However, another combination (s_2, s_4) shown in Figure 3(b) can further reduce the total distance to 13. Therefore, using the way of executing *GNNQ* *K* times to find the *K* best sites could return incorrect result.

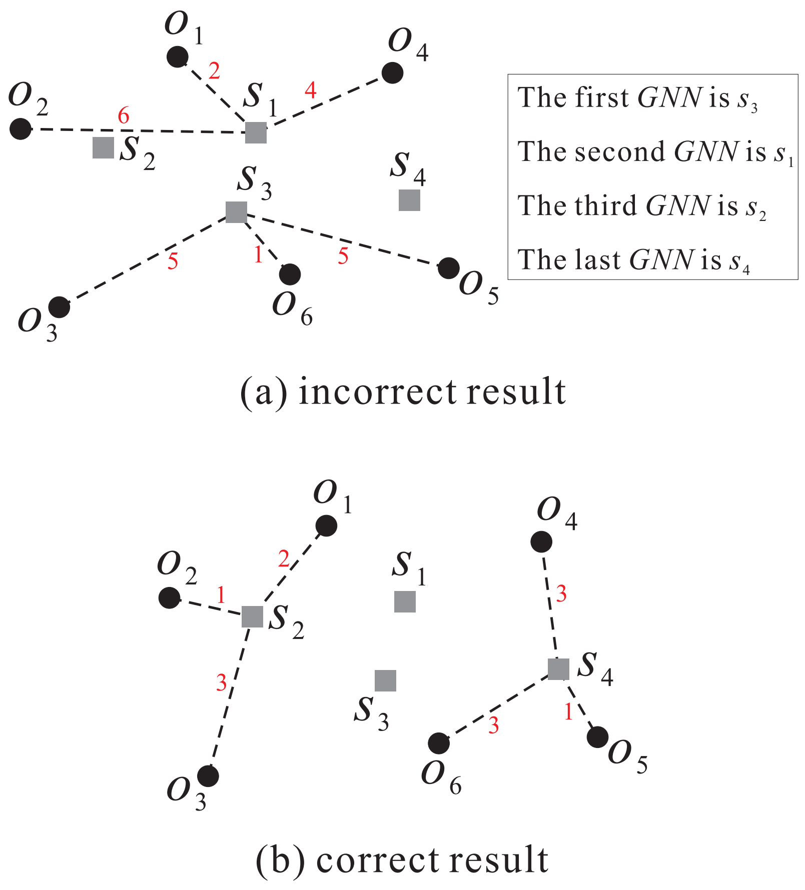

Figure 3. An example of the *GNNQ*

2.3. Methods For MDOLQ

Given a set of objects O and a set of sites S, a *MDOLQ* returns a location which, if a new site s not in S is built there, minimizes $\sum_{o_i \in O} d(o_i, s_j)$ where $d(o_i, s_j)$ is the distance between object o_i and its closest site $s_j \in S \cup \{s\}$. At first glance, the *MDOLQ* is more similar to the *KBSQ* than the other queries mentioned above. However, using the *MDOLQ* to obtain the *K* best sites may still lead to incorrect result.

Consider an example of using *MDOLQ* to find the *K* best sites in Figure 4. As 2*bs* are to be found, we can evaluate the *MDOLQ* two times to obtain the result. In the first iteration (as shown in Figure 4(a)), the site s_1 becomes the first *bs* because it has the minimum total distance to all objects. Then, the *MDOLQ* is executed again by taking into account the remaining sites s_2, s_3, and s_4. As the site s_2 can reduce more distance compared to the other two sites, it becomes the second *bs* (shown in Figure 4(b)). Finally, 2*bs* are s_1 and s_2 and the total distance is computed as $d(o_4, s_1) + d(o_5, s_1) + d(o_6, s_1) + d(o_1, s_2) + d(o_2, s_2) + d(o_3, s_2) = 20$. However, the computed distance is not minimum and can be further reduced. As we can see in Figure 4(c), if s_2 and s_4 are chosen to be the 2*bs*, the total distance can decrease to 16.

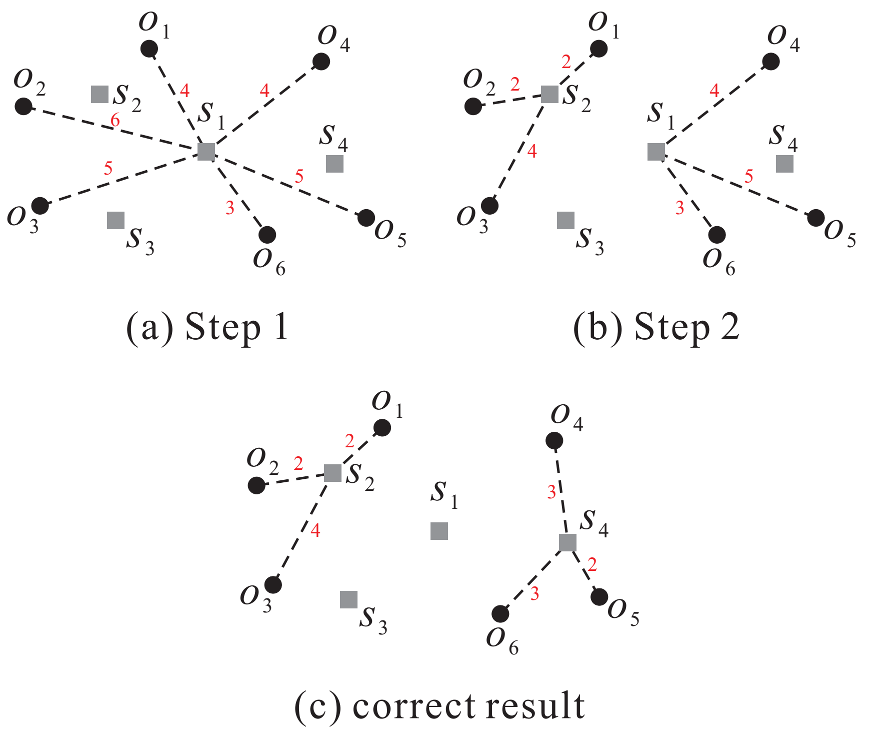

Figure 4. An example of the *MDOLQ*

3. STRAIGHTFORWARD APPROACH

In this section, we first propose a straightforward approach to solve the *KBSQ* problem, and then analyze the processing cost required for this approach. Assume that there are n objects and m sites, and the *K bs* would be chosen from the m sites. The straightforward approach consists of three steps. The first step is to compute the distance $d(o_i, s_j)$ from each object o_i $(1 \leq i \leq n)$ to each site s_j $(1 \leq j \leq m)$. As the K best sites are needed to be retrieved, there are totally $C^m{}_K$ possible combinations and each of the combinations comprises K sites. The second step is to consider all of the combinations. For each combination, the distance from each object to its closest site is determined so as to compute the total distance. In the last step, the combination of K sites having the minimum total distance is chosen to be the query result of *KBSQ*.

Figure 5 illustrates the three steps of the straightforward approach. As shown in Figure 5(a), the distances between objects and sites are computed and stored in a table, in which a tuple represents the distance from an object to all sites. Then, the $C^m{}_K$ combinations of K sites are considered so that $C^m{}_K$ tables are generated (shown in Figure 5(b)). For each table, the minimum attribute value of each tuple (depicted as gray box) refers to the distance between an object and its closest site. As such, the total distance for each combination can be computed by summing up the minimum attribute value of each tuple. Finally, in Figure 5(c) the combination 1 of K sites can be the *K bs* because its total distance is minimum among all combinations.

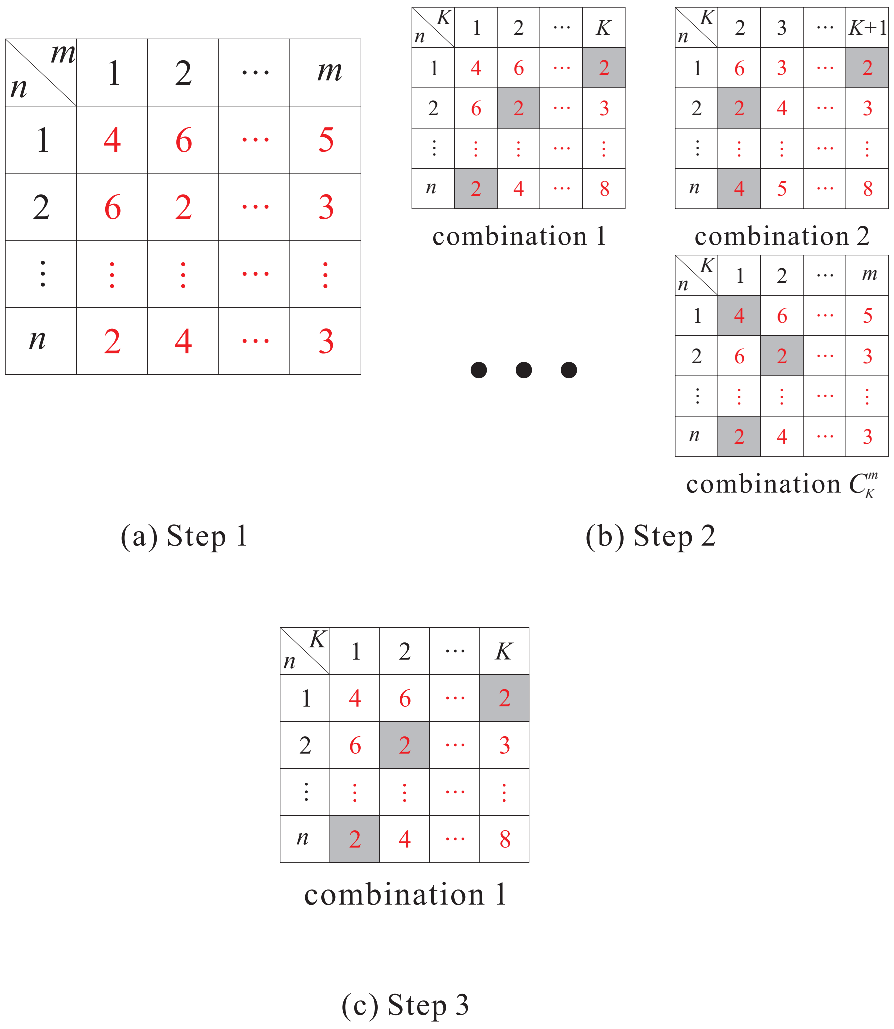

Figure 5. Straightforward approach

Since the straightforward approach includes three steps, we consider the three steps individually to analyze the processing cost. Let m and n be the numbers of sites and objects, respectively. Then, the time complexity of the first step is m*n because the distances between all objects and sites have to be computed. In the second step, C^m_K combinations are considered and thus the complexity is $C^m_K *n*K$. Finally, the combination having the minimum total distance is determined among all combinations so that the complexity of the last step is C^m_K. The processing cost of the straightforward approach is represented as $m*n + C^m{}_k * n * K + C^m{}_k$.

4. KBSQ Algorithm

The above approach is performed without any index support, which is a major weakness in dealing with large datasets. In this section, we propose the *KBSQ* algorithm combined with the existing indexes R*-tree and Voronoi diagram to efficiently process the *KBSQ*.

Recall that, to process the *KBSQ*, we need to find the closest site s for each object o (that is, finding the *RNN* o of site s). As the Voronoi diagram can be used to effectively determine the *RNN* of each site [14], we divide the data space so that each site has its own Voronoi cell. For example, in Figure 6(b), the four sites s_1, s_2, s_3, and s_4 have their corresponding Voronoi cells V_1,

V_2, V_3, and V_4, respectively. Taking the cell V_1 as an example, if object o lies in V_1, then o must be the *RNN* of site s_1. Based on this characteristic, object o needs not be considered in finding the *RNNs* for the other sites. Then, we use the R*-tree, which is a height-balanced indexing structure, to index the objects. In a R*-tree, objects are recursively grouped in a bottom-up manner according to their locations. For instance, in Figure 6(a), eight objects o_1, o_2, ..., o_8 are grouped into four leaf nodes E_4 to E_7 (i.e., the minimum bounding rectangle MBR enclosing the objects). Then, nodes E_4 to E_7 are recursively grouped into nodes E_2 and E_3, that become the entries of the root node E_1.

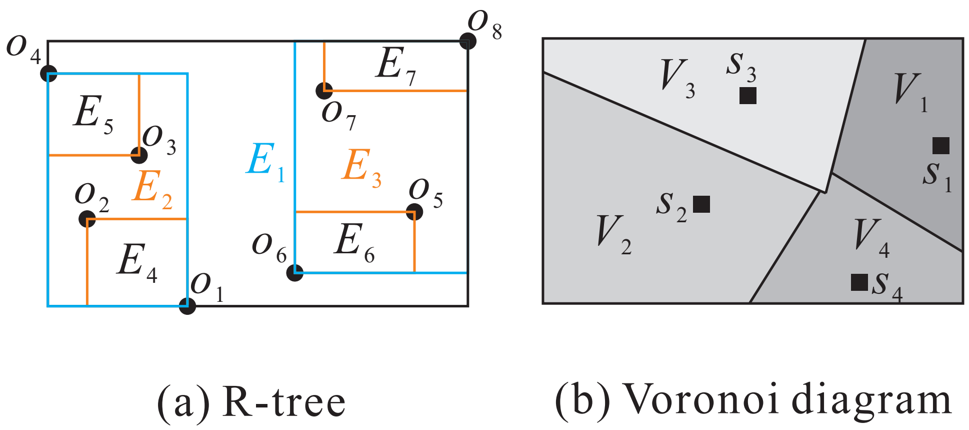

Figure 6. Spatial indexes

Combined with the R*-tree and Voronoi diagram, we design the following pruning criteria to greatly reduce the number of objects considered in query processing.

- **Pruning objects:** given an object o and the K sites s_1, s_2, ..., s_K, if o lies in the Voronoi cell V_i of one site s_i contained in $\{s_1, s_2, ..., s_K\}$, then the distances between object o and the other K-1 sites need not be computed so as to reduce the processing cost.

- **Pruning MBRs:** given a MBR E enclosing a number of objects and the K sites s_1, s_2, ..., s_K, if E is fully contained in the cell V_i of one site s_i contained in $\{s_1, s_2, ..., s_K\}$, then the distances from all objects enclosed in E to the other K-1 sites would not be computed.

To find the K *bs* for the *KBSQ*, we need to consider $C^m{}_K$ combinations of K sites. For each combination of K sites s_1, s_2, ..., s_K with their corresponding Voronoi cells V_1, V_2, ..., V_K, the processing procedure begins with the R*-tree root node and proceeds down the tree. When an internal node E (i.e., MBR E) of the R*-tree is visited, the pruning criterion 2 is utilized to determine which site is the closest site of the objects enclosed in E. If the MBR E is not fully contained in any of the K Voronoi cells, then the child nodes of E need to be further visited. When a leaf node of the R*-tree is checked, the pruning criterion 1 is imposed on the entries (i.e., objects) of this leaf node. After the traversal of the R*-tree, the total distance for the combination of K sites s_1, s_2, ..., s_K can be computed. By taking into account the total combinations, the combination of K sites whose total distance is minimum would be the query result of the *KBSQ*.

Figure 7 continues the previous example in Figure 6 to illustrate the processing procedure, where there are eight objects o_1 to o_8 and four sites s_1 to s_4 in data space. Assume that the combination (s_2, s_3) is considered and the Voronoi cells of sites s_2 and s_3 are shown in Figure 7(a). As the MBR E_2 is not fully contained in the Voronoi cell V_2 of site s_2, the MBRs E_4 and E_5 still need to be visited. When the MBR E_4 is checked, based on the pruning criterion 2 the distances from objects o_1 and o_2 to site s_3 would not be computed because their closest site is s_2. Similarly, the closest site of the objects o_7 and o_8 enclosed in MBR E_7 is determined as site s_3. As for objects o_3 to o_6, their closest sites can be found based on the pruning criterion 1. Having determined the closest site of each object, the total distance for combination (s_2, s_3) is obtained. Consider another combination (s_2, s_4) shown in Figure 7(b). The closest site s_2 of four objects o_1 to o_4 enclosed in MBR E_2 can be found when E_2 is visited. Also, we can compute the total distance for the combination (s_2, s_4) after finding the closest sites for objects o_5 to o_8. By comparing the distances for all combinations, the 2*bs* are retrieved.

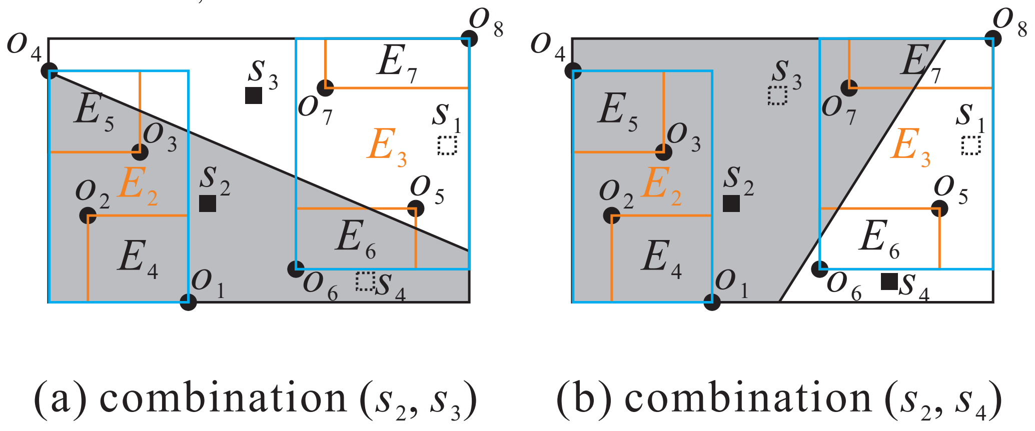

Figure 7. Processing *KBSQ* with indexes

5. Performance Evaluation

We conduct four experiments for the straightforward approach and the proposed *KBSQ* algorithm in this section. The first three experiments are evaluated to study the performance of the proposed methods by measuring the CPU time for processing a *KBSQ*. The last experiment demonstrates the usefulness of the *KBSQ* algorithm by comparing the precision of query result against its competitors.

5.1. Experimental Setting

All experiments are performed on a PC with Intel 2.83 GHz CPU and 4 GB RAM. The algorithms are implemented in Java. One synthetic dataset is used in our simulation. The synthetic dataset consists of 1000 objects whose locations are uniformly spread over a region of 100000 * 100000 meters. In the experimental space, we also generate 30 query datasets, each of which contains 25 sites whose locations are in the same range as those of the objects mentioned above. For each query dataset, we perform a *KBSQ* to find the *K* best sites, where the default value of *K* is set to 5. The performance is measured by the average CPU time in performing workload of the 30 queries. Table 1 summarizes the parameters under investigation, along with their default values and ranges. We compare the proposed *KBSQ* algorithm with the straightforward approach to investigate the performance of processing a *KBSQ*. Also, we compare the precision of the *KBSQ* algorithm against its competitors, including the *RNNQ*, the *GNNQ*, and the *MDOLQ* methods.

Table 1. System parameters.

Parameter	Default	Range
Number of objects (*O*)	1000	500, 1000, 5000, 10000
Number of sites (*S*)	25	20, 25, 30, 35
K	5	1, 5, 10, 20

5.2. Efficiency Of KBSQ Algorithm

In this subsection, we compare the *KBSQ* algorithm with the straightforward approach in terms of the CPU time. Three experiments are conducted to investigate the effects of three important factors on the performance of processing *KBSQ*. These important factors are the number of objects *O*, the number of sites *S*, and the value of *K*.

Figure 8 illustrates the performance of the *KBSQ* algorithm and the straightforward approach as a function of the number of objects (ranging from 500 to 10000). Note that hereafter all figures use a logarithmic scale for the *y*-axis. As we can see from the experimental result, the *KBSQ* algorithm significantly outperforms the straightforward approach in the CPU time, even for a smaller number of objects (e.g., 500). This is mainly because for the straightforward approach the distances of all objects have to be computed which incurs high computation cost. Moreover, the performance gap between the KBSQ algorithm and the straightforward approach increases with the increasing number of objects. The reason is that most distance computations of objects can be avoided by using the *KBSQ* algorithm with the support of R*-tree, but these distance computations are necessary for the straightforward approach.

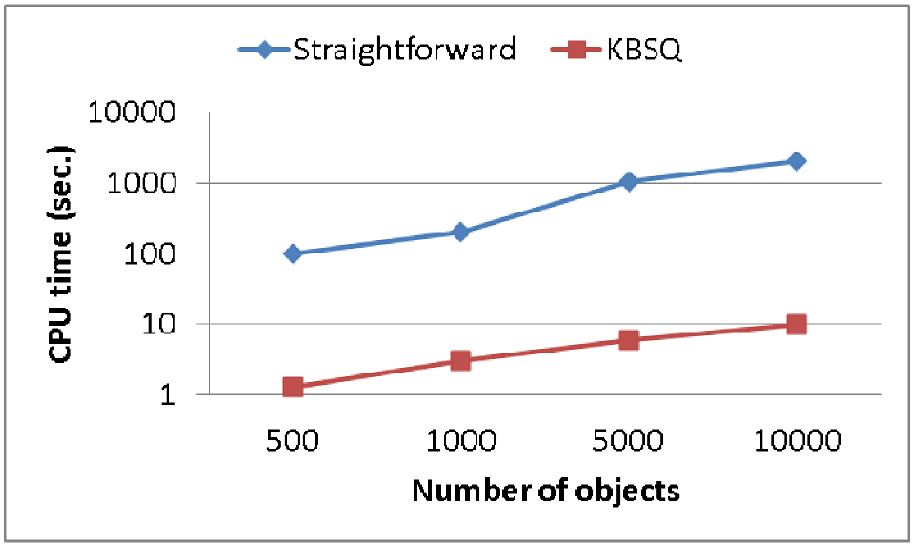

Figure 8. Effect of number of objects.

Figure 9 demonstrates the effect of various numbers of sites (i.e., varying *S* from 20 to 35) on the performance of the *KBSQ* algorithm and the straightforward approach. When the number of sites increases, the CPU overhead for both algorithms grows. The reason is that as the number of sites becomes greater, the number of combinations to be considered increases so that more distance computations between objects and sites are required. The experimental result shows that the *KBSQ* algorithm outperforms its competitor significantly in all cases, which confirms again that applying the *KBSQ* algorithm with R*-tree and Voronoi diagram can greatly improve the performance of processing a KBSQ.

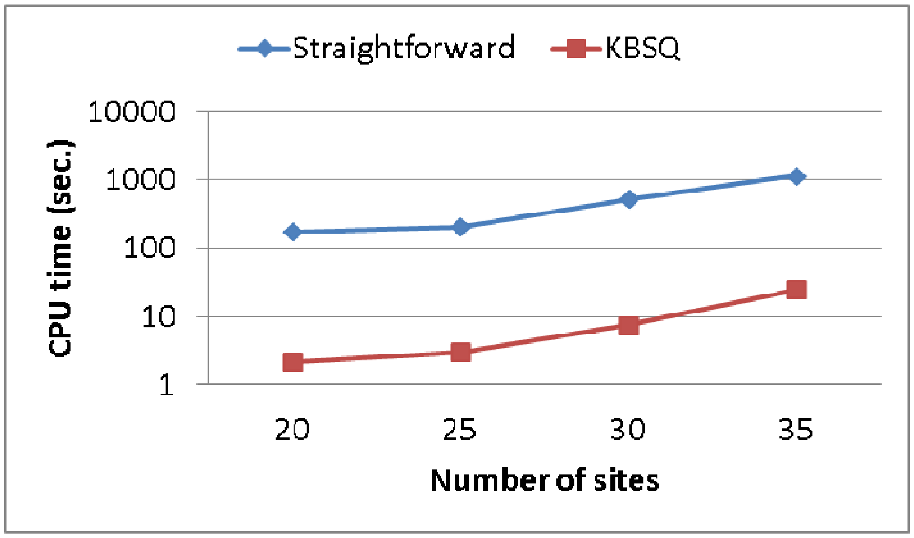

Figure 9. Effect of number of sites

Finally in this subsection, we study how the value of *K* affects the performance of the *KBSQ* algorithm and the straightforward approach, by varying *K* from 1 to 20. Similar to the previous experimental results, the *KBSQ* algorithm achieves significantly better performance than the straightforward approach (as shown in Figure 10). The *KBSQ* algorithm outperforms the straightforward approach by a factor of 70 to 240 in terms of the CPU cost. In addition, an interesting observation from Figure 10 is that a smaller *K* (e.g., 1) or a larger *K* (e.g., 20) results in a lower CPU time for the *KBSQ* algorithm and the straightforward approach. This is because for a smaller (or larger) value of *K*, less number of combinations needs to be considered in processing a *KBSQ* so that the required CPU time can be reduced.

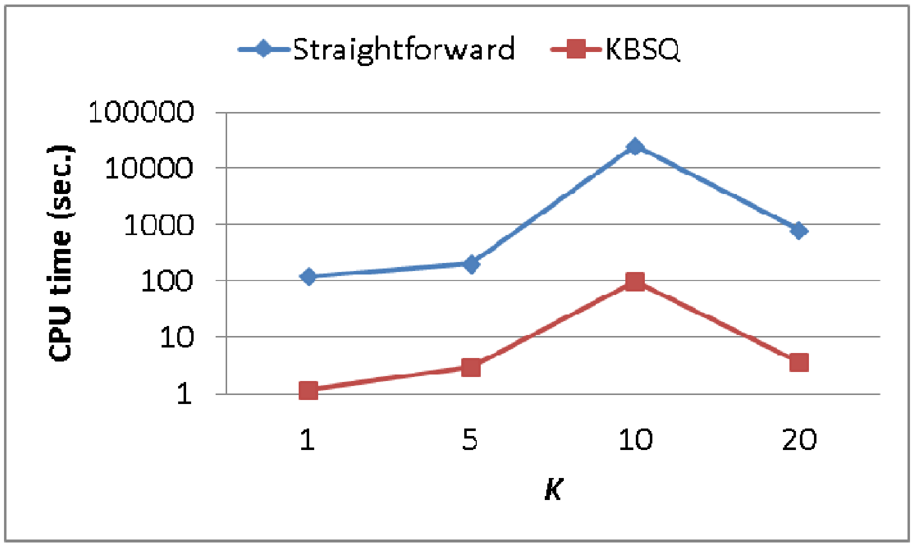

Figure 10. Effect of *K*

5.3. Precision Of KBSQ Algorithm

The following experiment demonstrates the precision of the *KBSQ* algorithm and its competitors (including the *RNNQ*, the *GNNQ*, the *MDOLQ* methods) under various values of *K*, where the precision is represented as follows:

$$precision = \frac{\#(bs_{result} \cap bs_{real})}{\#bs_{real}} \times 100\%$$

In the above equation, bs_{result} refers to the set of *K* best sites retrieved by executing the *KBSQ* algorithm, the *RNNQ* method, the *GNNQ* method, or the *MDOLQ* method. As for bs_{real}, it is the set of the real *K* best sites. In Figure 11, we vary *K* form 1 to 20 to investigate the precision of the *KBSQ* algorithm, the *RNNQ* method, the *GNNQ* method, and the *MDOLQ* method. As we can see, as the real *K* best sites can be precisely determined by executing the *KBSQ* algorithm, the precision of the *KBSQ* algorithm is always equal to 100% under different values of *K*. However, if the *RNNQ*, the *GNNQ*, and the *MDOLQ* methods are adopted to answer a *KBSQ*, some of the real *K* best sites are missed. As shown in the experimental result, the precision for the *MDOLQ* method can only reach 60% to 85%. Even worse, the precision for the *RNNQ* and the *GNNQ* methods is below 60% for a smaller value of *K*, which means that most of the retrieved best sites are incorrect.

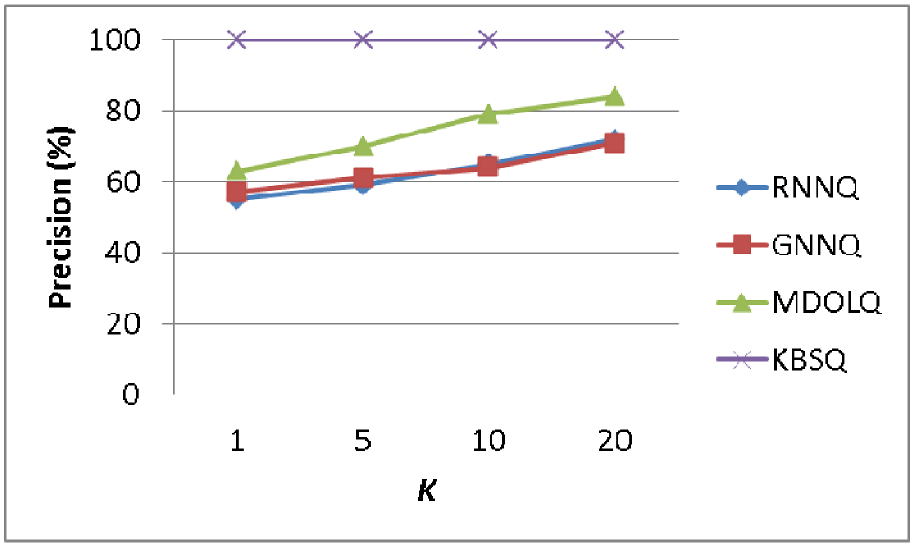

Figure 11. Precision for different *K*

6. CONCLUSIONS

In this paper, we focused on processing the *K* Best Site Query (*KBSQ*) which is a novel and important type of spatial queries. We highlighted the limitations of the previous approaches for the queries similar to the *KBSQ*, including the *RNNQ*, the *GNNQ*, and the *MDOLQ*. To solve the *KBSQ* problem, we first proposed a straightforward approach and then analyzed its processing cost. In order to improve the performance of processing the *KBSQ*, we further proposed a *KBSQ* algorithm combined with R*-tree and Voronoi diagram to greatly reduce the CPU and I/O costs. Comprehensive set of experiments demonstrated the efficiency and the precision of the proposed approaches.

Our next step is to discuss the space requirement of the proposed methods and design a novel index structure for answering the *KBSQ*. Then, we will focus on processing the *KBSQ* for moving objects with fixed or uncertain velocity. More complicated issues will be introduced because of the movement of objects. Finally, we would like to extend the proposed approach to process the *KBSQ* in road network.

ACKNOWLEDGEMENTS

This work was supported by National Science Council of Taiwan (R.O.C.) under Grants NSC 101-2119-M-244 -001 and NSC 102-2119-M-244 -001.

REFERENCES

[1] Benetis, R.; Jensen, C.S.; Karciauskas, G.; Saltenis, S. Nearest neighbor and reverse nearest neighbor queries for moving objects. *VLDB Journal* 2006, *15*, 229-249.

[2] Hakkoymaz, V. A specification model for temporal and spatial relations of segments in multimedia presentations. *Journal of Digital Information Management* 2010, *8*, 136-146.

[3] Huang, Y.-K.; Chen, C.-C.; Lee, C. Continuous k-nearest neighbor query for moving objects with uncertain velocity. *GeoInformatica* 2009, *13*, 1-25.

[4] Huang, Y.-K.; Liao, S.-J.; Lee C. Evaluating continuous k-nearest neighbor query on moving objects with uncertainty. *Information Systems* 2009, *34*, 415-437.

[5] Mokbel, M.F.; Xiong, X.; Aref, W.G. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In Proceedings of the ACM SIGMOD 2004.

[6] Pagel, B.-U.; Six, H.-W.; Winter, M. Window query-optimal clustering of spatial objects. In Proceedings of the ACM SIGMOD 1995.

[7] Papadias, D.; Tao, Y.; Mouratidis, K.; Hui, C.K. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.* 2005, *30*, 529-576.

[8] Huang, Y.-K.; Lin, L.-F. Evaluating k-best site query on spatial objects. In Proceedings of the NDT 2011.

[9] Guttman, A. R-trees: A dynamic index structure for spatial searching. In Proceedings of the ACM SIGMOD 1984.

[10] Samet, H. The design and analysis of spatial data structures. Addison-Wesley, Reading 1990.

[11] Korn, F.; Muthukrishnan, S. Influence sets based on reverse nearest neighbor queries. In Proceedings of the ACM SIGMOD 2010.

[12] Papadias, D.; Shen, Q.; Tao, Y.; Mouratidis, K. Group nearest neighbor queries. In Proceedings of the ICDE 2004.

[13] Zhang, D.; Du, Y.; Xia, T.; Tao, Y. Progressive computation of the min-dist optimal-location query. In Proceedings of the VLDB 2006.

[14] Zhang, J .; Zhu, M.; Papadias, D.; Tao, Y.; Lee, D.L. Location-based spatial queries. In Proceedings of the ACM SIGMOD 2003.