Finding the Sites with Best Accessibilities to Amenities

Qianlu Lin Chuan Xiao Muhammad Aamir Cheema Wei Wang
{qlin, chuanx, macheema, weiw}@cse.unsw.edu.au

The University of New South Wales
Australia

Abstract. Finding the most accessible locations has a number of applications. For example, a user may want to find an accommodation that is close to different amenities such as schools, supermarkets, and hospitals etc. In this paper, we study the problem of finding the most accessible locations among a set of possible sites. The task is converted to a top-k query that returns k points from a set of sites R with the best accessibilities. Two R-tree based algorithms are proposed to answer the query efficiently. Experimental results show that our proposed algorithms are several times faster than a baseline algorithm on large-scale real datasets under a wide range of parameter settings.

1 Introduction

Optimal location problems have received significant research attention in the past[9, 21]. In this paper, we study a new problem that finds the sites with the best accessibilities to amenities. Consider the example of a person who wants to rent an apartment. He may be interested in finding an apartment such that different amenities are close to it (e.g., it has a restaurant, a bus stop and a super market nearby). The person may specify different types of amenities that are of his interest. The accessibility of an apartment can be defined based on its closest amenity of each type. Furthermore, the person may define a scoring function such that it gives higher priority to certain types of amenities (e.g., a nearby bus stop may be more important than a nearby restaurant). We allow the users to define a monotonic scoring function to set such preferences. Formal definition is given in Section 2.

Similar to the existing work on the other versions of facility location problems [9, 21], our focus is on solving the problem in Euclidean space. Also, we focus on the case where the accessibility of a site depends on the closest amenity of each type. Nevertheless, we remark that the pruning rules presented in this paper can be extended to the case where the accessibility depends on m-closest amenities of each type.

Several approaches have been proposed to solve the all nearest neighbors problem (ANN) [22, 7, 10] and aggregate nearest neighbor problem [15, 14, 16]. However, these techniques only consider one type of amenity and cannot be efficiently

applied to our problem. Nevertheless, we use these techniques to design a baseline algorithm and show that our proposed algorithms perform significantly better than the baseline algorithm.

We propose two efficient R-tree based algorithms. The first algorithm constructs indexes for different types of amenities in separate R-trees, and traverses the R-trees in parallel to progressively output top-k query results. The second algorithm indexes the different types of amenities in a single R-tree and demonstrates better performance in most of the settings. Both algorithms carefully exploit the lower bound of the accessibility scores with several non-trivial optimizations applied.

Another important feature of our algorithms is that we progressively report the best sites in an order of their accessibility scores. Such progressive/incremental answer may be useful in many interactive applications and a user may terminate the algorithm if he is satisfied with first j results (where $j < k$).

Below we summarize our contributions.

- We proposed two algorithms to find top-k accessible sites among a set of possible locations. Unlike traditional algorithms, our algorithms are able to compute the results progressively.
- We developed several non-trivial pruning and optimization techniques that can be integrated into the two proposed algorithms in order to reduce the I/O cost and running time.
- We performed experiments on several real datasets. Our proposed algorithms are shown to outperform the baseline algorithm in all settings.

The rest of the paper is organized as follows: Section 2 gives the problem definition and introduces a baseline algorithm to the problem. We propose two main algorithms in Section 3. Several optimizations to the main algorithms are presented in Section 4. We present and analyze experimental results in Section 5, and survey related work in Section 6. Section 7 concludes the paper.

2 Preliminaries

In this section, we define the problem and introduce a baseline algorithm based on R-trees.

2.1 Problem Definition

We define R and S as two spatial datasets, and each point s in S is assigned a type i. Let $|T|$ be the total number of types. A distance metric $d(r, s)$ measures the Euclidean distance between two points r and s. For a point $r \in R$, let $NN(r, S_i)$ be the nearest point of type i from r. The accessibility cost c_r of r is given by the formula below;

$$c_r = f(d(r, NN(r, S_1)), \cdots, d(r, NN(r, S_{|T|})). \quad (1)$$

where $f()$ is a monotonic scoring function that takes $|T|$ values as parameters and returns a single value[1]. Our goal is to find k points from R, such that their accessibility costs are the smallest among all the points in R.

2.2 All Nearest Neighbor Algorithm

An immediate solution to the proposed top-k problem is to borrow the techniques from all nearest neighbor queries [10, 22, 7, 12, 2, 8]. For each point r in R, we enumerate its nearest neighbors of each type in S, compute the accessibility cost for r, and then output the k points with the smallest accessibility costs. Existing algorithms for computing all nearest neighbor queries presume the data points are indexed by an R-tree [11] and use various optimizations to reduce the search space.

First, we build an R-tree I_R on the points in R, and for each type of points in S, we build a separate R-tree I_{S_i}. Hence there are $|T| + 1$ R-trees. We then probe these R-trees, starting from their roots.

In order to determine the access order of the nodes, we employ the following two data structures:

Local Priority Queue (LPQ) Each node u in I_R owns exactly one LPQ, a min-heap that maps its entries to the nodes in I_{S_i}. Each entry v in the priority queue has two values `mind` and `maxd`, indicating the minimum distance from u's MBR to v's MBR, and the maximum distance from u's MBR to v's MBR. Figure 1 shows an example of two MBRs and their `mind` and `maxd`. The priority queue orders its entries by increasing `mind` values, while `maxd` values are used for pruning unpromising entries. An LPQ also keeps two values `minmind` and `minmaxd`, representing the smallest of the `mind` values among its entries, and the smallest of the `maxd` values among its entries, respectively.

Global Priority Queue (GPQ) A min-heap maintains all the LPQs that are generated when accessing the R-trees, ordered by increasing `minmind`. `minmaxd` is used for pruning the LPQs that are guaranteed not to produce any nearest neighbors.

LPQ and GPQ allow us to access the nodes with smallest lower bound of distance first, which are the most promising nodes. Additionally, we apply pruning techniques to avoid accessing the nodes that cannot generate nearest neighbors to the points in R. We only allow one LPQ for each node in I_R, so that accessing duplicate nodes in I_R can be avoided.

The all nearest neighbor algorithm iterates through all the types in S. Within each iteration for a type i, it starts with the root of I_R and I_{S_i}, and expands the nodes in a bi-directional fashion [7, 19]. [2] The nearest neighbors are returned

[1] For sake of simplicity, in rest of the paper, we consider that the monotonic scoring function returns the sum of these $|T|$ values. However, we remark that our algorithms can be applied on any monotonic scoring function.

[2] Although there exist alternative ways to expand nodes in R-trees, we choose bi-directional expansion because it is shown to outperform others in extensive experiments [7].

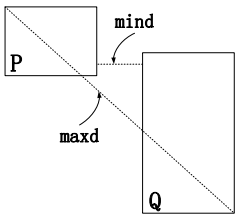


Fig. 1. Illustration of `mind` anx `maxd`

if the point level is reached in both I_R and I_{S_i}. After finish the tree expanding for all the types in S, the k points with smallest accessibility costs are output as final results to the top-k query.

The all nearest neighbor algorithm sequentially processes the points in S according to types. The drawback is that I_R will be traversed $|T|$ times, and we cannot obtain any results until all the types are processed. In addition, the algorithm does not exploit the abundant information provided by the top-k results, and hence the pruning power is very limited. In Section 3, we will show how to traverse the R-trees simultaneously and output top-k results progressively, as well as exploit the inherent information contained in these results.

3 Main Algorithm Frameworks

In this section, we give two basic algorithms to compute the k locations with the smallest accessibility cost.

3.1 Separate-tree Method

The first algorithm is to choose the same indexes as the all nearest neighbor algorithm, but traverse these trees in a parallel fashion. Since different types of points in S are indexed in separate R-trees, we call this approach separate-tree algorithm.

We still choose LPQ and GPQ as the data structures in separate-tree algorithm. However, each entry u in I_R owns $|T|$ LPQs in separate-tree algorithm because we expand the entries in R-trees in a parallel way. We call these LPQs u's LPQ group, still denoted LPQ_u, with which we are able to estimate the lower bound of the accessibility costs for the points indexed in u. In u's LPQ group, we denote $LPQ_u[i]$ the LPQ that maintains entries from I_{S_i}, and calculate the lower bound as

$$LB_c = \sum_{i=1}^{|T|} LPQ_u[i].\texttt{minmind}.$$

A major difference from the all nearest neighbor method is that we arrange the LPQs pushed to GPQ by increasing order of LB_c. A fixed sized min-heap M is

used to keep the top-k results seen so far, and $M[k]$ gives the temporary result with k-th smallest accessibility cost. Before expanding entries to form new LPQ groups, we compare the new LPQ groups' LB_c with $M[k]$'s accessibility cost, and allow only those whose LB_c are smaller than $M[k]$'s accessibility cost. Otherwise they are guaranteed not to produce any results that can beat the current temporary results.

Algorithm 1: SeparateTree (I_R, I_S)

```
1  for each point r in R do
2      c_r ← 0;
3  M ← InitializeTempResults ;  /* Store any k points as initial results */
4  GPQ ← ∅;
5  for i = 1 to |T| do
6      u ← I_R.root; v ← I_{S_i}.root;
7      LPQ_u[i] ← ∅;
8      LPQ_u[i].minmind ← +∞; LPQ_u[i].minmaxd ← +∞;
9      SepTreePushAndUpdate(LPQ_u[i], v);
10 GPQ.push(LPQ_u);
11 while GPQ ≠ ∅ do
12     LPQ_u ← GPQ.pop();
13     SepTreeExpandTrees(LPQ_u, GPQ);
```

Algorithm 1 describes this parallel algorithm. We initialize the min-heap M by choosing any k points as initial temporary results. These points are computed for their all type nearest neighbors and accessibility costs. Like the all nearest neighbor algorithm, the separate tree algorithm starts with the roots of I_R and all I_{S_i}, and then expands the nodes in a bi-directional fashion. The first LPQ group formed is owned by I_R's root, and the root of I_{S_i} is inserted into priority queues (Line 6 – 9). Then this LPQ group is pushed into a GPQ (Line 10). We iteratively select an LPQ group from the GPQ, and expand nodes in both R-trees I_R and I_{S_i}.

Algorithm 2: SepTreePushAndUpdate $(LPQ_u[i], v)$

```
1  if mind(u, v) < LPQ_u[i].minmaxd then
2      LPQ_u[i].push(v);
3      LPQ_u[i].minmind ← min(LPQ_u[i].minmind, mind(u, v));
4      LPQ_u[i].minmaxd ← min(LPQ_u[i].minmaxd, maxd(u, v));
5      LPQ_u.LB_c ← Σ_{i=1}^{|T|} LPQ_u[i].minmind
       ;                    /* update lower bound of accessibility cost */
```

Algorithm 3: SepTreeExpandTrees (LPQ_u, GPQ)

```
1  if u is a point then
2      for i = 1 to |T| do
3          while LPQ_u[i] ≠ ∅ do
4              v ← LPQ_u[i].pop();
5              if v is a point then
6                  c_u ← c_u + d(u, v);
7                  if u's NNs of all types are found and c_u < M[k].cost then
8                      M.add(u, c_u) ;                    /* update temp results */
9                  return
10             else
11                 for each v' ∈ v do
12                     SepTreePushAndUpdate(LPQ_u[i], v');
13     if LPQ_u.LB_c < M[k].cost then GPQ.push(LPQ_u);
14 else
15     for each u' ∈ u do
16         LPQ_u[i]' ← ∅; LPQ_u[i]'.minmind ← +∞; LPQ_u[i]'.minmaxd ← +∞;
17     for i = 1 to |T| do
18         while LPQ_u[i] ≠ ∅ do
19             v ← LPQ_u[i].pop();
20             if v is a point then
21                 for each u' ∈ u do
22                     SepTreePushAndUpdate(LPQ'_u[i], v);
23             else
24                 for each v' ∈ v do
25                     for each u' ∈ u do
26                         SepTreePushAndUpdate(LPQ'_u[i], v');
27     for each u' ∈ u do
28         if LPQ_u.LB_c < M[k].cost then GPQ.push(LPQ_u);
```

The expansion algorithm is shown in Algorithm 3. Given a node u in I_R, the entries in u's LPQ group are popped, according to the order of `mind`. We identify the nearest neighbor, add the distance to the the accessibility cost, and update temporary results when point level is reached in both I_R and I_{S_i} (Line 8). Otherwise, the children of both u and popped entry v paired to form new LPQ groups. Specifically, for each type, we expand u and create a group for each of its children u', and the children of v is then inserted to the LPQ of u'. To avoid accessing the nodes that cannot generate any nearest neighbors for the points in u, we compare the nodes' `mind` to u' with the `minmaxd` of u'. Only if its `mind` is smaller than u''s `minmaxd`, we insert this node to u''s LPQ (Line 1, Algorithm 2). The values of `minmind` and `minmaxd` of the LPQ are updated once an entry is

inserted, and finally we check the LB_c of new formed LPQ group before inserting it into the GPQ (Line 13 and 28), since the temporary results in M can be used to prune unnecessary LPQ groups. In addition, a temporary result is confirmed as a final result if its accessibility cost is smaller than the LB_c of the LPQ group popped from GPQ. The results are progressively output with the execution of the algorithm.

3.2 One-tree Method

The above separate-tree method adopts the same indexing scheme as the all nearest neighbor algorithm. Here, we consider building indexes for the various types of points in S in one tree. Although the estimation of LB_c will be looser due to multiple types indexed in the nodes of R-trees, we are able to achieve a more efficient node expansion and hence better runtime performance.

We call this method one-tree algorithm. To record the type information, we add an attribute to the nodes of the R-tree built on S. This new attribute maintained in each node of I_S is a *type bitmap* B of length $|T|$ that indicates which types of points are contained in (the descendants of) the node. The bit i is set to 1 if the node or its descendants contain at least one point of type i, or 0 otherwise.

The one-tree algorithm follows the separate-tree algorithm framework, but differs in the generation of LPQs and lower bound estimations of accessibility costs. Now we allow an LPQ to store entries of various types. In order not to miss any real results due to abuse of types, the values of `minmaxd` are broken down into specific types. We use the notation `minmaxd`$[i]$ to capture the smallest of the `maxd` values among its entries that contain points of type i. Before inserting an entry to the LPQ, we check the type bitmap of the entry, and allow only the entries whose `mind` is smaller than `minmaxd`$[i]$ on at least one type i.

Similarly, `minmind`$[i]$ stores the smallest of the `mind` among the entries that contain points of type i. This is to estimate the lower bound of accessibility cost, as given by the following equation:

$$LB_c = \sum_{i=1}^{|T|} LPQ_u.\texttt{minmind}[i].$$

Algorithm 4 captures the pseudo-code of forming new LPQs and updating LB_c in the one-tree algorithm. Before inserting entry v into u's LPQ, we check the types contained in v. If the pair (u, v) can produce final top-k results, there must be at least one type i such that v's `mind` is smaller than LPQ_u's `minmaxd` [i]. We use a boolean variable flag to capture whether such type can be found. If it is set to **true**, we insert v into u's LPQ, and make necessary updates.

4 Optimizations on Existing Algorithms

In this section, we introduce several pruning and optimization techniques that can be integrated into the two basic algorithm frameworks proposed in Section 3.

Algorithm 4: OneTreePushAndUpdate (LPQ_u, v)

```
1 flag ← false;
2 for each type i in B_v do
3     if mind(u, v) < LPQ_u.minmaxd[i] then flag ← true;
4 if flag = true then
5     LPQ_u.push(v);
6     for each type i in B_v do
7         LPQ_u.minmind[i] ← min(LPQ_u.minmind[i], mind(u, v));
8         LPQ_u.minmaxd[i] ← min(LPQ_u.minmaxd[i], maxd(u, v));
9     LPQ_u.LB_c ← Σ_{i=1}^{|T|} LPQ_u.minmind[i];
```

4.1 Break Ties in Priority Queues

In both separate-tree and one-tree algorithms, the entries in LPQs are arranged by increasing order of `mind`. Consider a node u in I_R, and an entry in its LPQ, v. If u's and v's MBRs are overlapped, the value of `mind` will become zero. This often happens for the high-level nodes in R-trees. To alleviate this problem, we break ties by choosing the one with the smaller `maxd` if two entries have the same `mind`.

The same problem may occur in GPQ. We compute the upper bound of the accessibility costs for the points indexed in u:

$$UB_c = \sum_{i=1}^{|T|} LPQ_u.\texttt{minmaxd}[i].$$

The LPQ with the smaller UB_c is chosen to break ties if there exist multiple LPQs that have the equal value of LB_c in GPQ.

4.2 Early Check to Avoid Unnecessary Expansion

In the basic separate-tree and one-tree algorithm, an entry v' is checked for its `mind` before it is inserted to u''s LPQ. The LPQ admits only the entries whose `mind` is smaller than the LPQ's current `minmaxd`.

Since the entry v' is expanded from its parent node v, we can check the `mind` between u' and v, and safely prune v' from u''s LPQ given that the $\texttt{mind}(u', v)$ is larger than or equal to the `minmaxd` of u'. This is because $\texttt{mind}(u', v) \leq \texttt{mind}(u', v')$.

We apply this optimization before expanding v by calculating the `mind` between u' and v, so that the access to v's children can be avoided if the minimum distance between the two MBRs is too large.

4.3 Pre-update Temporary Results

The third major optimization is based on the observation that all the points indexed in a node u may have smaller accessibility cost than the temporary

result M, and the number of points indexed in u exceeds k. We can verify this by comparing UB_c, the upper bound of the accessibility costs for the points indexed in u, and the k-th temporary result's cost. If the upper bound is smaller, the top-k temporary results are to be updated in future expansions. In this case, the k-th temporary result's cost is updated beforehand, although the expansions are not done yet. We assign the value of UB_c to the k-th temporary result's accessibility cost, because the cost of the final k-th result is *at most* as large as UB_c.

5 Experiments

In this section, we report our experimental results and analysis.

5.1 Experiment Setup

The following algorithms are compared in the experiment.

ANN is the all nearest neighbor algorithm described in Section 2.
Sep-Tree is separate-tree top-k search algorithm proposed in Section 3.
One-Tree is one-tree top-k search algorithm proposed in Section 3.

All optimizations described in Section 4 are applied to separate-tree and one-tree unless specified otherwise.

All algorithms are implemented as in-memory algorithms. They are implemented in C++ and performed on a PC with Pentium D 3.00GHz CPU and 2GB RAM. The operating system is Debian 4.1. The algorithms are complied using GCC 4.1.2 with `-O3` flag. We used two publicly available real datasets in

Dataset	$\|S\|$	$\|R\|$
NA	174,956	17,000
SF	175,813	17,000

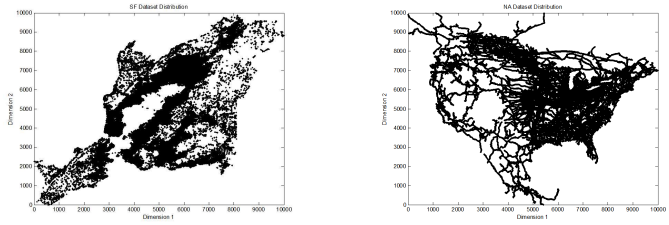
Table 1. Statistics of Datasets

the experiment, San Francisco Road Network (SF) and Road Network of North America (NA). [3] Some important statistics and data distribution are shown in Table 5.1 and Figure 5.1. We split each dataset into $|T|$ types, and the default $|T|$ is set to 20 unless specified otherwise. For each data point in the dataset, a random type is assigned.

To generate the set of possible locations R, we choose 10% of the points from the original dataset, and shift the x and y coordinates by a random number in the range of $[-5, 5]$.

We measure the number of internal nodes expanded and the number of leaf nodes expanded in the R-trees, as well as the processing time. The processing time measured does not include the time for constructing R-tree indexes.

[3] `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`



(a) San Francisco Road Network (SF)

(b) Road Network of North America (NA)

Fig. 2. Data Distribution

5.2 Effect of Optimization

We first study the effect of optimizations, and run the two proposed algorithms on both NA and SF datasets with several optimization techniques applied. We compare four optimizations:

No-Opt The basic separate-tree (one-tree) algorithm with no optimization techniques applied.

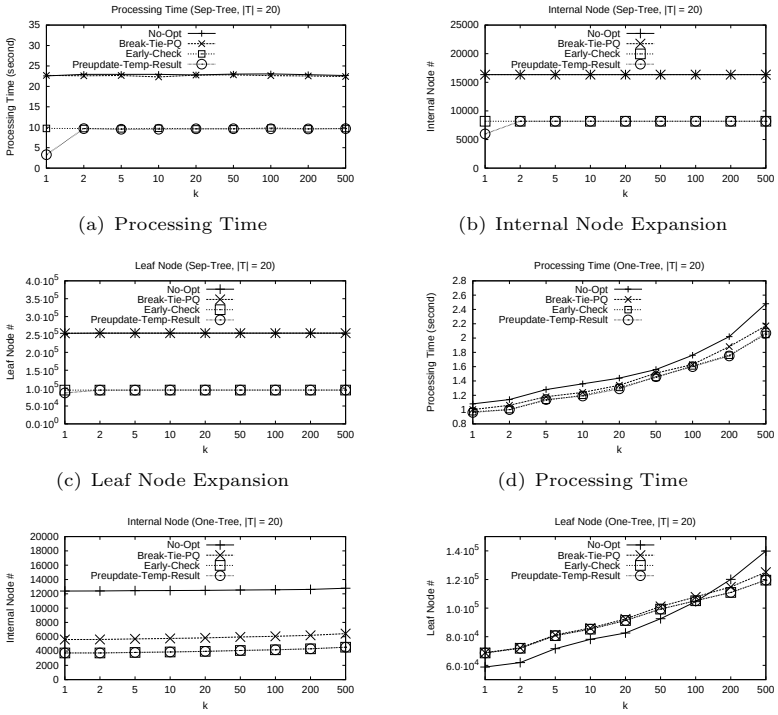
Break-Tie-PQ The above algorithm equipped with the "break ties in priority queues" optimization. In LPQs, we arrange entries by increasing order of `mind`, and then `maxd`. In GPQs, we arrange entries by increasing order of LB_c, and break ties by UB_c.

Early-Check The above algorithm equipped with the "early check" optimization technique to avoid redundant node expansions. Before expanding a node v in I_S, we check the `mind` between the LPQ owner u' and v, and prune v if the value of `mind` is no smaller than the `minmaxd` of u''s LPQ.

Preupdate-Temp-Result The above algorithm equipped with the "preupdate temporary result" technique. If a node u in I_R contains at least k points in R, and its LPQ (group)'s UB_c is smaller than the current k-th temporary result's accessibility cost, we then regard UB_c as the k-th temporary result's accessibility cost so as to improve the algorithm's pruning power.

Figures 5.2(a) – 5.2(c) show the processing time, the number of internal node expansions, and the number of leaf node expansions using separate-tree on NA dataset. The performance on SF dataset displays similar trends and thus is not shown here in the interest of space. It can be observed that Early-Check is the most effective optimization for separate-tree algorithm. It reduces the processing time by 57%. The main reason is that the number of internal nodes is reduced by 50%, and the number of leaf nodes expanded is reduced by 63% after applying Early-Check. The other two optimization techniques exhibit minor improvements to the separate-tree algorithm.

Figures 5.2(d) – 5.2(f) show the processing time, number of internal node and leaf node expansions using one-tree algorithm on NA dataset. Break-Tie-PQ is the most significant optimization technique for one-tree algorithm. It reduces the



(a) Processing Time (b) Internal Node Expansion

(c) Leaf Node Expansion (d) Processing Time

(e) Internal Node Expansion (f) Leaf Node Expansion

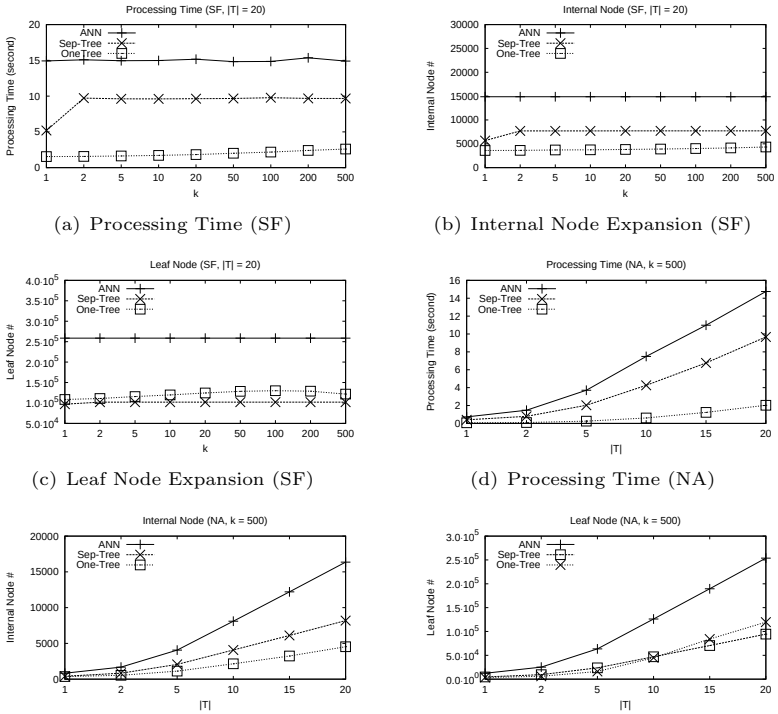
Fig. 3. Effect of Optimization (NA)

processing time by about 10% when k is small and about 20% when k is large. The other two optimizations can further reduce the processing time. The reason why the effect of Break-Tie-PQ is more remarkable on one-tree algorithm than on separate-tree algorithm is that one-tree algorithm constructs indexes by putting the points with various types in one node, and therefore yields looser estimation of lower bound of accessibility cost. There are more LPQs sharing equal values of LB_c, and therefore it is necessary to break them by introducing the upper bound of accessibility cost.

5.3 Comparison with All Nearest Neighbor Algorithm

We run the three algorithms with various numbers of returned objects (k) and types of points ($|T|$). Figures 5.3(a) – 5.3(c) show the performance on SF dataset with respect to different k. The general trend is that running time of both separate-tree and one-tree algorithm slightly increases when we move k towards larger values; the running time of ANN algorithm is irrelevant to the change of k because it always computes the nearest neighbors for all the possible locations. The result shows that separate-tree algorithm is 1.5 times as fast as the ANN

algorithm, and the speed-up of one-tree algorithm can be up to 5.7x. The speed-up is mainly due to more efficient internal node and leaf node expansion. As can be seen, separate-tree reduces the the number of internal nodes by 48%, and one-tree reduces the number by additional 23%. In terms of leaf node expansion, separate-tree and one-tree display similar performance, and the reduction can be up to 61%, compared to the ANN algorithm.



(a) Processing Time (SF)

(b) Internal Node Expansion (SF)

(c) Leaf Node Expansion (SF)

(d) Processing Time (NA)

(e) Internal Node Expansion (NA)

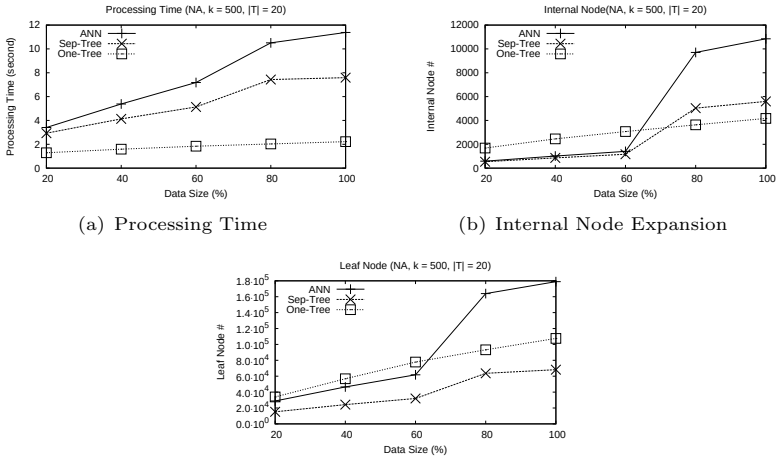
(f) Leaf Node Expansion (NA)

Fig. 4. Comparison with ANN

We study the performance of the algorithms with respect to varying number of types $|T|$, and plot the results on NA dataset in Figures 5.3(d) – 5.3(f). k is set to 500 in this set of experiments. As shown in the figures, the processing time and the node expansion grow linearly while the number of types $|T|$ is increasing. Both separate-tree and one-tree have slower increasing rates than ANN.

5.4 Scalability against Data Sizes

We study how the proposed algorithms perform on different size of datasets. Figure 5.4 shows the performance on NA dataset with respect to different number



(a) Processing Time

(b) Internal Node Expansion

(c) Leaf Node Expansion

Fig. 5. Scalability (NA)

of data points. The number of points in R is fixed at 17,000, and k is set to 500 for this set of experiments. We observe that when data size grows, the processing time of the algorithm grows linearly, When data size is 20% of the original S, one-tree algorithm is about twice as fast as separate-tree algorithm. When data size is 100%, one-tree algorithm is more than three times as fast. Although separate-tree expands three times fewer nodes than one-tree when data size is small, one-tree is still faster under this scenario. This is because one-tree accesses less number of entries in the point level of R-trees, while separate-tree needs to access $|T|$ times for each point to create its LPQ group and compute the accessibility cost.

5.5 Index Sizes

Dataset	ANN	separate-tree	one-tree	Original Dataset
NA	11.36MB	11.36MB	12.80MB	4.35MB
SF	11.34MB	11.34MB	12.69MB	4.33MB

Table 2. Index Size

Table 5.5 shows the size of index on the two datasets using different algorithms, and the size of the original datasets. separate-tree algorithm uses the same amount of disk memory as ANN algorithm, which is 2.6 times as large as the original datasets. one-tree uses about 13% more disk space to store the R-trees because it needs to keep an additional bitmap in each node to indicate what types are assigned to the points indexed by this node and its descendants.

Summary. Considering the runtime performance and the space usage of the three algorithms, we find that one-tree algorithm achieves the best runtime performance while occupying similar amount of space with the other two. We recommend users to select the highest accessible locations using one-tree algorithm.

6 Related Work

Facility location problem, also known as location analysis, is to find optimal placement of facilities respective to cost to a given set S. The problem we propose in this paper is one of the variations.

Another variation is Minsum facility location [9]. It is to seek a location that minimizes the sum of distances from a set of points to the selected location. [9] proposed three tree-based algorithm to solve it. Among these three proposed algorithms, Virtual OL-tree is the most efficient algorithm. Virtual OL-tree is an extension of k-d-B tree [17]. In [21], an variant of the Minsum facility location problem was studied. This paper proposed a partition-based algorithm. It recursively partitions candidate cell of the query region to smaller cells.

Other related studies include [4, 3, 20]. In [4, 3], the optimization problem is defined as: Given two sets S and P, find the point s in S that satisfies: 1) number of bichromatic reverse nearest neighbours (BRNN) of s in P is maximum; 2) the maximum distance of the BRNN of s is minimum; 3) the minimum distance of the BRNN of s is maximum. [20] solves the problem to find a region Q such that when placing s in Q, its BRNN size is maximum.

The problem we propose is related to k nearest neighbor (NN) query as well. kNN query has been extensively studied by spatial database community and many spatial indexes were proposed to solve this problem [18, 13, 5]. Among these indexes, R-tree [11] and its variations [1] are most popular ones. Two variations of NN query, group nearest neighbor queries and all nearest neighbor, have been recently studied. [6] provided a detailed survey of work related to these two types of queries.

7 Conclusion

In this paper, we study the problem finding k best locations that are close to various types of facilities. We focus on Euclidean space and measure the accessibility using the sum of distances to nearest neighbors. Two algorithms are proposed to efficiently find the top-k answers, with several non-trivial optimizations applied to reduce the number of node expansion and improve runtime performance. The separate-tree algorithm creates indexes for different types of points in separate R-trees, while the one-tree algorithm indexes all the points in a single R-tree. The experiment results show that both proposed algorithms outperform the baseline algorithm with a speed-up up to 5.7 times.

Acknowledgement. The authors would like to thank the anonymous reviewers for their insightful comments. Wei Wang is supported by ARC Discovery Projects DP0987273 and DP0881779.

References

1. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.
2. C. Böhm and F. Krebs. The -nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.
3. S. Cabello, J. M. Díaz-Báñez, S. Langerman, C. Seara, and I. Ventura. Reverse facility location problems. In *CCCG*, pages 68–71, 2005.
4. S. Cabello, J. M. Díaz-Báñez, S. Langerman, C. Seara, and I. Ventura. Facility location problems in the plane based on reverse nearest neighbor queries. *European Journal of Operational Research*, 202(1):99–106, 2010.
5. S. Chaudhuri and L. Gravano. Evaluating top- selection queries. In *VLDB*, pages 397–410, 1999.
6. M. Cheema. Circulartrip and arctrip: Effective grid access methods for continuous spatial queries.
7. Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *ICDE*, pages 1056–1065, 2007.
8. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, 49(1):67–104, 2004.
9. Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.
10. T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma. Optimizing all-nearest-neighbor queries with trigonometric pruning. In *SSDBM*, pages 501–518, 2010.
11. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.
12. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248, 1998.
13. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.
14. H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *GIS*, pages 192–199, 2005.
15. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, pages 634–645, 2005.
16. D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, pages 301–312, 2004.
17. J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD Conference*, pages 10–18, 1981.
18. T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, pages 154–165, 1998.
19. H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD Conference*, pages 343–354, 2000.
20. R. C.-W. Wong, M. T. Özsu, P. S. Yu, A. W.-C. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.
21. D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, pages 643–654, 2006.
22. J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, pages 297–306, 2004.