A *k*-nearest Neighbour Query Processing Strategy Using the mqr-tree

Wendy Osborn(✉)

Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, AB T1K 3M4, Canada
wendy.osborn@uleth.ca

Abstract. This paper presents a *k*-nearest neighbour query processing strategy that utilizes a recently proposed spatial access method, the mqr-tree, to narrow down the search for candidate nearest neighbours. It requires the traversal of only one path of the tree, and in the majority of cases, does not require backtracking all the way to the root. The mqr-tree-based *k*-nearest neighbour strategy is evaluated both individually and compared against the brute-force method for running time. It is shown that the proposed approach performs better in many cases. In addition, the running time is not affected by the number of points being indexed or the number of nearest neighbour being sought.

Keywords: Location-based services · Spatial access methods · k-nearest neighbour query

1 Introduction

A location-based service provides information to a user of a mobile device, such as a smartphone or mobile-enabled tablet – based on their location, interests and the type of query issued by the user [10]. One example of such a query is to locate all restaurants that reside within a certain distance d from the user. Another example of such a query is to locate the nearest k restaurants to the user. These queries are referred to as a region query and a k-nearest neighbour (k-NN) query, respectively.

Several approaches for efficiently finding nearest neighbours to a query point have been proposed [7], with several utilizing various spatial access methods, such as the R-tree or kd-tree [1–6,9,11]. Along with the additional space overhead, some spatial access methods are not balanced and can ultimately become skewed, while others contain overlapping sub-regions, which result in multiple paths needing to be searched. A recently proposed spatial access method, the mqr-tree [8], has been shown to achieve zero overlap in the presence of point data, while also achieving some relative balance (if not perfect balance) between all leaf nodes and the root. In addition, due to its organization of data to maintain the spatial (i.e., topographical) relationships between them, it is an excellent candidate for processing k-NN queries.

L. Barolli et al. (eds.), *Advances in Network-based Information Systems*,
Lecture Notes on Data Engineering and Communications Technologies 7,
DOI 10.1007/978-3-319-65521-5_49

Therefore, I propose a k-nearest neighbour strategy that utilizes the mqr-tree to locate a sub-region of candidate points that will potentially satisfy a k-nearest neighbour query. This strategy only requires the traversal of one path of the tree to satisfy a query. In the majority of cases, it does not require backtracking all the way to the root node. The mqr-tree-based k-NN strategy is evaluated both individually and compared against the brute-force method for running time. It is shown that the proposed approach performs better in many cases. In addition, the running time is not affected by the number of points being indexed by the tree or the number of nearest neighbour being sought.

2 Related Work and Background

In this section, I summarize related work in the area of k-NN query processing. I also summarize the features of the mqr-tree that are required for this work.

2.1 *k*-nearest Neighbour Strategies

Nearest neighbour strategies are classified into structureless and structured. Structureless strategies do not utilize a complex data structure for searching. The most straightforward approach is the brute-force approach. Here, the distances between query point q and all points in set P are calculated before processing P to find the k nearest neighbours. Although the space complexity is low, processing the list can be costly for higher values of k.

Structured strategies utilize some type of data structure to improve performance, at the expense of both space overhead and the cost of constructing the index [1–6,9,11]. Burkhard and Keller [3] proposed three multi-way trees for nearest neighbour searching. Fukunaga [5] utilized recursive decomposition of a search space and a branch-and-bound strategy to create a search data structure. The branch-and-bound strategy systematically accesses and evaluates all candidate data, while eliminating subsets of data as early and often as possible, using the continuously optimized bound(s) derived during the evaluation.

The first nearest neighbour search algorithm for the k-d tree was proposed by Friedman *et al.* [4]. Sproull [11] improved upon it by noting that since Euclidean distance calculations are invariant under rotation, the planes that partition space along a particular dimension in the k-d tree can actually be arbitrary k-dimensional hyperplanes. However, he identified the following limitations, including the additional costs of computing the distance between a point and an arbitrary hyperplane, and of choosing an arbitrary partition hyperplane. Approaches using the R-tree are also proposed, including one from Brinkhoff *et al.* [2], which computes k nearest neighbours via a spatial join operation.

Finally, some approaches can be applied to any hierarchical data structure. Roussopoulos *et al.* [9] proposed a depth-first branch-and-bound k-NN search that can be applied to any tree data structure. For every node visited, its children are placed on a queue in order of distance from the query point. Any nodes that are far from the query point are pruned. This continues until k nearest neighbours have been found. Hjaltason and Samet [6] propose a similar best-first strategy that does not require the number of nearest neighbours to be known in advance.

2.2 mqr-tree

More details on mqr-tree (including validity, insertion, construction, and basic region searching algorithms) can be found in [8].

The mqr-tree [8] is a spatial access method that uses two-dimensional nodes to organize objects in two-dimensional space. This allows the existing spatial relationships to be maintained between objects and the regions of space that contain them. After an object or point is inserted, a validity test is performed to ensure that all spatial relationships are maintained, and any objects or regions that violate the spatial relationship rules are relocated. In addition to traditional region searching and point searches, the features of the mqr-tree also allow it to support k-NN searching. A very nice feature of the mqr-tree that will lend itself nicely to k-NN searching is that zero overlap of regions (on the same level of the tree) occurs when the mqr-tree is used to solely index point data [8].

Figure 1 presents an mqr-tree for the given dataset. A node has the quadrants NW, NE, SW and SE. Each node has a corresponding node MBR, which encompasses all objects, points and regions in the subtrees accessible from the node. All objects and regions containing other objects, are placed in the appropriate quadrant based on their relationship to the centroid of the nodeMBR. For example, in the leaf node containing m1,m2 and p9, we observe that m1 is NW of the centroid for the node MBR that contains it Similarly, m2 is SW and p9 is NE of

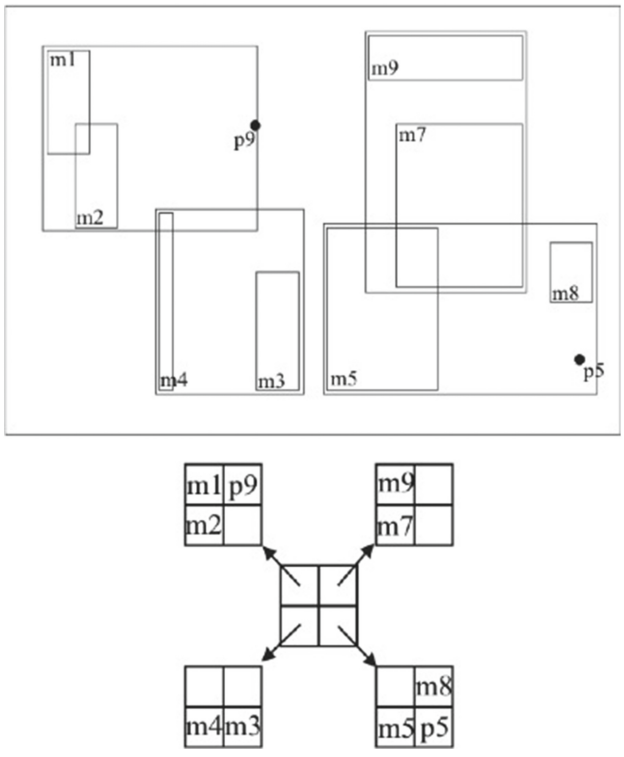


Fig. 1. mqr-tree example (from [8])

the centroid, respectively. Therefore, these objects are placed in the NW, SW, and NE quadrants of the node, respectively. The other leaf nodes, and the root node, are also arranged in this manner.

To use the mqr-tree, it must be constructed if it does not already exist. Constructing the mqr-tree will take anywhere from a few seconds for 10,000 objects, up to almost two minutes for 100,000 points, by using repeated insertion of objects. However, the mqr-tree also support the insertion and deletion of objects, so any updates to the data set can be made in the mqr-tree easily. Further, to use the mqr-tree for k-NN searching, a quick traversal of the tree must be performed to count the number of points that are reachable from every node, if the tree is already built.

3 An mqr-tree-Based k-Nearest Neighbour Strategy

In this section I introduce the new k-NN strategy. This strategy utilizes the mqr-tree to quickly locate a set of k or more candidate points to satisfy a k-NN query. A node that will serve as s "starting point" in the mqr-tree is located first, before a candidate set of points that (hopefully) will contain the k nearest neighbours to the query point are fetched. Although as we see, the strategy may need to "backtrack" towards the root in order to find a proper set of k nearest neighbours, the advantage of the mqr-tree of not having overlap of any regions at the same level, means that only one path in the tree needs to be utilized for locating the required nearest neighbours. In addition, results from experiments show that backtracking is more the exception than the rule.

Figure 2 presents the pseudocode for the proposed approach. After obtaining the query point and the number of nearest neighbours being sought (k), the search begins at the root node for a node that will be the starting point for fetching candidate k nearest neighbours. The query point is first evaluated against the root node and its corresponding nodeMBR. One of the following situations will occur:

1. The number of points that are reachable from this node, $npoints$, is $<= k$. If this occurs, then the search for a starting point stops here. If $npoints < k$, then the search goes back to the parent node for the starting point.
2. The query point is within the node's nodeMBR, but in a location not covered by one of the quadrants. the search for a starting point also stops here.
3. The query point will reside in one of the 4 quadrants - NW, NE, SE or SW (in other words, the point resides in the space that is covered by one of these quadrants). The search will continue in the subtree of the encompassing quadrant, and terminate when one of the two above conditions are met.

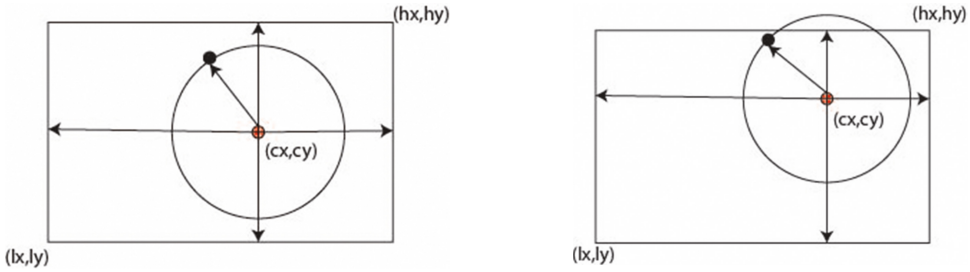
Once the node that will serve as a starting point is found, its nodeMBR is fetched. This is the $superMBR$, or the MBR that will encompass the candidate set of points from which the k nearest neighbours will be found. The mqr-tree will then be traversed from this node to fetch all points in the node's subtree. The fetched points are sorted by increasing distance, before the kth furthest point from the

```
point P = obtain_search_point();
int k = obtain_knn_value();
node X;
node_region SR;
bool found = 0;
point Q[k]; //dynamic, adjusted if need be

//first, locate the first node with the number of points
//equal to k
while(!found && X—>npoints > k)
     if(P in X[NW])
          X = X[NW]—->child;
     else if (P in X[NE])
          X = X[NE]—->child;
     else if (P in X[SE])
          X = X[SE]—->child;
     else if (P in X[SW])
          X = X[SW]—->child;
     else
          found = 1;
     end if
end while

//if not enough points in subtree go back up one level
if(X—>npoints < k)
     X = X—>parent;

//if set of fetched points contains valid k nearest neighbours
//then finished. Otherwise, go back up the tree and repeat
while(1)
     //obtain the points from X, and corresponding superMBR
     traverse_tree(X, Q);
     sort_by_distance(Q);
     SR = obtain_node_region(X)

     if(X—>parent == NULL)
        return Q; //at the root
     else if(dist(P,Q[k]) <= dist(P.cy,SR.hy) &&
        dist(P,Q[k]) <= dist(P.cy,SR.ly) &&
        dist(P,Q[k]) <= dist(P.cx,SR.hx) &&
        dist(P,Q[k]) <= dist(P.lx,SR.lx))
          return Q;  //valid result
     else
          X = X—>parent;
end while
```

Fig. 2. k-NN search strategy

query point will be evaluated against the superMBR to determine if the set of k nearest neighbours is valid. A set of k nearest neighbours is valid only if the distance of the kth nearest neighbour from the query point is less than or equal to the distances from the query point to all sides of the superMBR. Any distances that are less than the distance to the kth nearest neighbour may mean that another closer point may reside elsewhere in the tree. Figures 3a and b depict the valid and invalid situations respectively for the 1-nearest neighbour case. As we can see, in Fig. 3a, the distance from the query point to the closest point is less than the distances to all four sides of the superMBR. So we know that this is a valid nearest neighbour. In Fig. 3b, notice that the distance between the query point and the closest point to it is greater than the distance to the north side of the superMBR. This means that there may be a closer nearest neighbour that resides outside of the north side and within another node's nodeMBR, so this candidate is not guaranteed to be the nearest neighbour.

If the set of candidate k nearest neighbours is valid, then these are sent to the user. Otherwise, another set of candidate points will be obtained from the parent of the chosen starting point node. If necessary, this process will be repeated by proceeding up the path one parent node at a time, until either: (1) a set of valid k nearest neighbours is found that passes the validity test, or (2) the root node is reached, which means that the candidate set contains all points in the tree and therefore the closest k of them must be the k nearest neighbours.



(a) Valid (b) Invalid

Fig. 3. Valid and invalid cases

4 Evaluation

This section presents the empirical evaluation of the mqr-tree-based k-NN strategy, including a comparison against the brute-force k-NN approach. I first present the framework and evaluation methodology. Then, I will present the outcome of the evaluation and the resulting discussion of the outcome.

4.1 Methodology

The mqr-tree-based k-NN strategy is implemented in C on a PC running the Centos 7 version of Linux. It was evaluated using several synthetic point sets for both

the data and the queries. These were chosen so that certain characteristics, such as the number of points and distribution, could be controlled. Altogether, 30 points sets were created for the experiments. The first are 10 point sets of uniform distribution. Each contains 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, and 100000 points respectively. Each set of n points is drawn from a two-dimensional square region of size $\sqrt{n^2}$. Next are 10 point sets of exponential (i.e., skewed) distribution. Each also contains the same number of points as the uniform datasets, and are drawn from the same sized regions of space as the uniformly distributed sets. Finally, there are 10 sets of query points. Each contains the square root of the number of points in the data set they are applied to. For example, the query point set corresponding to the 100-point data sets contains 10 query points, while the one corresponding to the 100,000-point data set contains 316 query points. The reason for this is because the query point set in each file proceeds diagonally through the region of space that contains the points.

The following tests are performed by applying the point query sets to their respective point data sets, mentioned above:

- 1-NN, mqr-tree-based strategy. 20 tests, with 10 using the uniform point sets and 10 using the exponential point sets.
- 1-NN, brute-force strategy. 20 tests, with 10 using the uniform point sets and 10 using the exponential point sets.
- k-NN (k = 1 to 10), mqr-tree-based strategy. 20 tests, with 10 using the uniform 100,000-point set and 10 using the exponential 100,000 point set.
- k-NN (k = 1 to 10), brute-force strategy. 20 tests, with 10 using the uniform 100,000-point set and 10 using the exponential 100,000 point set.

For the tests that utilize the mqr-tree-based k-NN strategy, the following performance factors are recorded:

- running time. The running time is recorded for each query. Two average running times are calculated and recorded: the average running time over every query, and the average running time for the queries that run in less than 2 μs. The reason for this will be explained when the results are discussed.
- number of queries that execute in less than 2 μs. Required for the second average running time mentioned above.
- number of pages hits. The number of page hits (i.e., nodes that are checked) for each query is recorded. The average number of page hits is calculated over all queries.
- number of queries that have at least one invalid test. As mentioned in the previous section, it is possible that the initial set of candidate points that is fetched may not contain a valid set of k nearest neighbours, due to other members of the true k nearest neighbours residing outside of the corresponding superMBR. Therefore, the number of queries that have at least one invalid result is recorded.
- number of queries that traverse for points from the root. As mentioned in the previous section, it is possible that the chosen superMBR corresponds to the root node, which means that the entire mqr-tree is traversed. Therefore, we also record the number of queries where this occurs.

Table 1. 1-NN on uniform data

#points	#queries	avg#phits	#invalid	#root	avgtime	#under2μs	avgtime2μs
100	10	24.60	4	2	0.51	10	0.51
500	22	62.18	9	2	0.56	22	0.56
1000	31	55.84	7	1	0.58	30	0.52
5000	70	193.59	34	2	1.99	65	0.52
10000	100	199.63	38	2	3.37	96	0.56
50000	223	277.70	81	1	22.49	216	0.57
100000	316	531.31	136	1	65.19	299	0.53

For the tests that utilize the brute force approach, only the individual running time for each test is recorded, along with the average running time over all point queries. The other performance factors recorded for the mqr-tree-based approach are not applicable for the brute-force approach.

4.2 Results

The results for the mqr-tree-based k-NN approach are presented first, followed by the comparison of the running time versus the brute-force approach. For all tables presented here, k is the number of nearest neighbours being sought, #points is the number of points in the mqr-tree, #queries is the number of queries executed on the mqr-tree, avg#phits is the average number of nodes visited for a k-NN search, #invalid is the number of queries that had one or more invalid fetches of k nearest neighbours while searching the tree, #root is the number of queries that traversed for the candidate set of k nearest neighbours from the root, #under2μs is the number of queries that finished in under 2 μs, avgtime is the average execution time per query over all of the queries executed (i.e. #queries), and avgtime2μs is the average time per query over those queries that executed in under 2 μs (i.e. #under2μ). Also, due to limited space, we only include a subset of results for each table.

Tables 1 and 2 present the results of the 1-NN tests across all exponential and uniform point sets, respectively. We observe for the uniform point sets that for a significantly high percentage of queries (90% or greater), that a significantly low average running time is observed. In addition, the number of tree traversals that occur from the root are very low, and the number of invalid candidate 1 nearest neighbours that are fetched is less than 50% in most cases, and not more than 54% (for the queries on 100,000 points). The higher average run time and average number of page hits is a result of the few queries that produce unacceptable run times and number of page hits, which significantly skew the results. But in most cases, the performance here is very good.

For the exponential cases, we observe a slightly higher percentage of invalid candidate 1-nearest neighbours and traversals from the root, and a slightly lower percentage of queries that execute in under 2 μs. In addition, both the average

Table 2. 1-NN on exponential data

#points	#queries	avg#phits	#invalid	#root	avgtime	#under2μs	avgtime2μs
100	10	72.20	1	7	0.61	10	0.61
500	22	254.36	12	8	0.77	22	0.77
1000	31	279.52	16	7	1.01	28	0.82
5000	70	993.74	39	13	8.08	53	0.51
10000	100	1061.04	50	8	15.18	87	0.51
50000	223	3114.56	115	7	215.46	210	0.54
100000	316	5761.48	171	7	806.13	285	0.55

Table 3. K-NN on uniform data

k	avg#phits	#invalid	#root	avgtime	#under2μs	avgtime2μs
1	531.30	136	1	65.18	299	0.53
2	872.87	187	2	99.13	290	0.54
3	1389.28	211	4	240.01	286	0.55
4	1498.94	217	4	244.24	280	0.56
5	1656.18	223	4	252.71	275	0.57
6	1705.44	227	4	252.52	269	0.57
7	1716.00	226	4	252.38	268	0.58
8	1778.54	222	4	257.02	267	0.58
9	2292.91	226	6	343.13	261	0.58
10	2556.74	235	7	401.92	259	0.59

number of page hits and the average overall run time are unacceptably high. However, the same situation occurs here that a small number of queries take an unacceptable time to execute, and therefore skews the overall results.

Next, Tables 3 and 4 present the results of the mqr-tree-based k-NN tests on both the 100,000-point uniform and exponential point sets, respectively. We observe for the uniform point set that as the number of nearest neighbours increases, the number of queries that execute under 2 μs decreases slightly, but at its lowest, is still 81% of the overall number of queries. The average running time of the queries that run under 2 μs is still quite low. The remaining performance factors show an increase as the value of k increases, and in the case of the average overall running time, the number of page hits, and the number of invalid candidates k nearest neighbours are quite high. The first two, again, are the result of the few queries that ultimately take significantly more running time and skew the results. The increase in the latter is due to the increase in k, since more points must reside in the superMBR in order to be considered valid. With respect to the exponential point set, although the values are approximately equal or higher than those found for the uniform point set, the same trends are found for increasing k.

Table 4. K-NN on exponential data

k	avg#phits	#invalid	#root	avgtime	#under2μs	avgtime2μs
1	5761.48	171	7	805.44	287	0.56
2	6205.02	191	8	870.96	283	0.56
3	6205.13	193	8	871.26	282	0.56
4	6439.28	226	9	906.36	282	0.57
5	6439.62	228	9	906.29	284	0.58
6	7529.20	230	11	1071.49	278	0.59
7	7529.28	218	11	1067.21	279	0.59
8	9240.96	226	14	1306.46	269	0.59
9	9244.39	227	14	1307.95	271	0.60
10	11438.16	230	20	1624.50	263	0.63

Finally, Figs. 4 and 5 present the results of the comparison between the average run times of the mqr-tree-based k-NN strategy and the brute force approach, for the 1-NN and k-NN tests, respectively. For both charts, we use the mqr-tree-based running time of under 2 μs, since this reflects the majority of the queries that were executed in our evaluations.

With respect to the 1-NN results, it is noted that the mqr-tree-based strategy achieves fairly constant running time, regardless of the number of points in the point set. For smaller point set size – in particular, up to 2000 points – we see no advantage for the mqr-tree-based k-NN strategy over the brute force approach. In fact, brute force performs slightly better than the proposed approach in many of these cases. However, once the point set reaches 5000, then savings in running time over the brute-force approach start to appear and increase

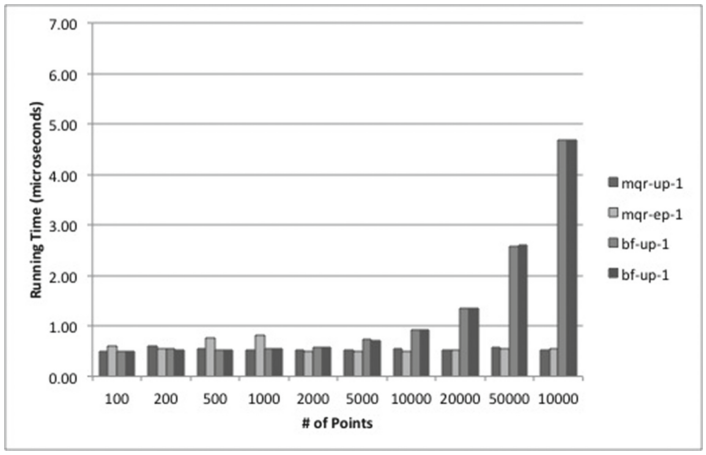


Fig. 4. 1-NN Comparison versus brute force search

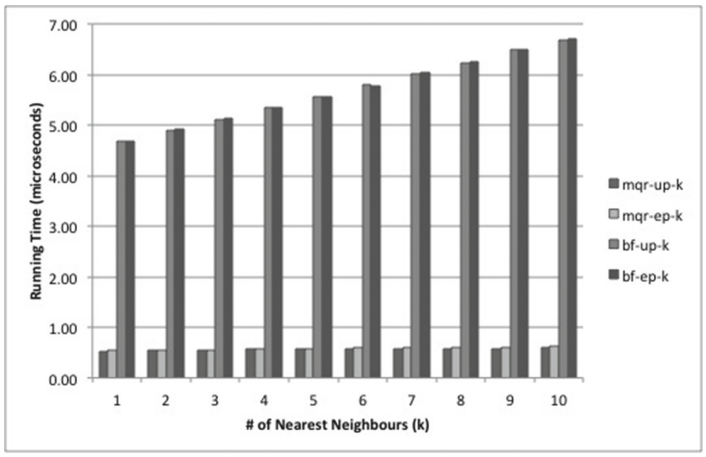


Fig. 5. k-NN Comparison versus brute force search

significantly as the point set size increases. With respect to the k-NN results, we observe that the mqr-tree-based strategy significantly outperforms the brute force approach for all values of k. It is also noted here that the running time of the mqr-tree-based approach is fairly constant, regardless of the number of nearest neighbours being sought.

4.3 Discussion

The mqr-tree-based k-NN strategy achieves significantly low running times in many cases. In addition, the running time in these cases are fairly constant, regardless of the number of points in the point set, or the number of nearest neighbours being sought. When compared to the brute force approach, the proposed strategy outperforms brute force in larger point sets. It is expected that this will continue to be the case as the number of points increases.

However, there is the situation of the "rogue" searches - the ones that take unacceptably longer than necessary. I would like to explain the situations that caused a small number of queries to produce unacceptably high running time and page fetches, which ultimately cause some overall results to look worse than they should have been. This can be cause by two scenarios. The first is that the query is in "unindexed space". This is the situation where a query resides within a nodeMBR (ultimately chosen as a superMBR), in a location that was not indexed by any of the subtrees of the corresponding node. Although this situation can occur at any point in the mqr-tree, it is most noticeable when it happens in the root. This means that the entire tree must be traversed for the candidate k nearest neighbours. This is especially unacceptable when k is low. The second is that the query is located near "border". This is the situation where the query resides very close to the border of the space being indexed. This would equate to the query being near the end of the nodeMBR of the root. Further, the query resides further down the mqr-tree, but continually fails validity tests,

and moves back up the tree until it reaches the root. This would result in many invalid candidate k nearest neighbour sets being fetched, which ultimately would cause the running time and the number of page hits to be unacceptably high.

5 Conclusion

This paper presents a k-nearest neighbour query processing strategy that utilizes a recently-proposed spatial access method, the mqr-tree, to narrow down the search for candidate nearest neighbours. It requires the traversal of only one path of the tree, and in the majority of cases, does not require backtracking all the way to the root. The mqr-tree-based k-nearest neighbour strategy is evaluated both individually and compared against the brute-force method for running time. It is shown that the proposed approach performs better in many cases. In addition, the running time is not affected by the number of points being indexed by the tree or the number of nearest neighbour being sought.

Some future research directions include: (1) further empirical evaluation versus other strategies – in particular, those that use the R-tree, since it is known to contain overlap of subregions; (2) improvements to the proposed strategy in order to eliminate the cases of root processing, and (3) proposal of a more bottom-up strategy for k-nearest neighbour query processing, which would eliminate the requirement of specifying k in advance.

References

1. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM **45**(6), 891–923 (1998)
2. Brinkhoff, T., Kriegel, H.-P., Seeger, B.: Efficient processing of spatial joins using r-trees. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, pp. 237–246. ACM, New York (1993)
3. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. Commun. ACM **16**(4), 230–236 (1973)
4. Friedman, J.H., Baskett, F., Shustek, L.J.: An algorithm for finding nearest neighbors. IEEE Trans. Comput. **24**(10), 1000–1006 (1975)
5. Fukunage, K., Narendra, P.M.: A branch and bound algorithm for computing k-nearest neighbors. IEEE Trans. Comput. **24**(7), 750–753 (1975)
6. Hjaltason, G.R., Samet, H.: Ranking in spatial databases. In: Proceedings of the 4th International Symposium on Advances in Spatial Databases, SSD 1995, pp. 83–95. Springer-Verlag (1995)
7. Ilarri, S., Mena, E., Illarramendi, A.: Location-dependent query processing: where we are and where we are heading. ACM Comput. Surv. **42**(3), 1–73 (2010)
8. Moreau, M., Osborn, W.: mqr-tree: a two-dimensional spatial access method. J. Comput. Sci. Eng. **15**, 1–12 (2012)
9. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. SIGMOD Rec. **24**(2), 71–79 (1995)
10. Schiller, J.H., Voisard, A. (eds.): Location-Based Services. Morgan Kaufmann, San Francisco (2004)
11. Sproull, R.F.: Refinements to nearest-neighbor searching in k-dimensional trees. Algorithmica **6**(4), 579–589 (1991)