Evaluation of Buffer Queries in Spatial Databases*

Edward P.F. Chan

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
http://emap.uwaterloo.ca

Abstract

A class of commonly asked queries in a spatial database is known as *buffer queries*. An example of such a query is to "find house-power line pairs that are within 50 meters of each other." A buffer query involves two spatial data sets and a distance d. The answer to this query are pairs of objects from the two input sets that are within distance d of each other. Evaluation of buffer queries is a costly operation, even when the numbers of objects in the data sets are relatively small. This paper addresses the problem of how to evaluate this class of queries efficiently. Geometric objects points, lines and regions are used to denote the shape and location of spatial objects. Two objects are within distance d of each other precisely when their minimum distance ($minDist$) is. A fundamental problem with buffer query evaluation is to find an efficient algorithm for solving the $minDist$ problem. Such an algorithm is found and its desirability is demonstrated. Finding a fast $minDist$ algorithm is the first step to evaluate a buffer query efficiently. It is observed that many, and even most, candidates can be determined to be in the answer without resorting to the relatively expensive $minDist$ operation. A candidate is first evaluated with the least expensive technique - called 0-object filtering. If it fails, a more costly operation, called 1-object filtering, is applied. Finally, if both filterings fail, the most expensive $minDist$ algorithm is invoked. To show the effectiveness of these techniques, they are incorporated into the tree join algorithm and tested with real-life as well as synthetic data sets. Extensive experiments show that the proposed algorithm outperforms existing techniques by a wide margin in both the execution time as well as IO accesses. More importantly, the performance gain improves drastically with the increase of distance values.

1 Introduction

In a spatial database system, there are many different types of queries ranging from simple window queries to more complex distance-related queries. An important class of distance-related queries is known as *buffer queries*. Examples of such queries are to "find buildings that are within 50 meters of a highway," or to "find building-river pairs that are within 10 meters of each other." A buffer query involves two spatial data sets and a distance d. The answer to this query are pairs of objects

*An extended abstract of this paper is published in the Seventh International Symposium on Spatial and Temporal Databases (SSTD 2001), Los Angeles, July 2001.

from the two input sets that are within distance d of each other. This paper addresses the problem of how to evaluate this class of queries efficiently.

Geometric objects points, lines and regions are used to denote the shape and location of spatial objects. A fundamental problem with buffer query evaluation is to find an effective algorithm for solving the $minDist$ problem for non-point objects. The brute-force $minDist$ algorithm requires considering all pairs of segments from two geometric objects. A more efficient $minDist$ algorithm, which only requires a sub-sequence of segments from each object to be examined, is derived. The proposed $minDist$ algorithm has the same worst-time complexity as the brute-force. However, experiments with different types of real-life data sets show the proposed algorithm reduces the computation time to a fraction of that when computed with the brute-force. The $minDist$ algorithm could also be used for other distance-related queries such as nearest neighbor [18] or closest pair queries [7].

Finding an effective $minDist$ algorithm is an important first step toward solving the evaluation problem. Buffer queries can be evaluated by modifying existing spatial join algorithms. It is observed that many, and even most, candidates can be determined to be in the answer set with less expensive operations. To reduce the computation time and the number of spatial objects retrieved from the disk, filtering techniques, which we call *0-object* and *1-object* filterings, are employed. In a 0-object filtering, pairs of objects are proven to be in the answer, by looking only at their minimum bounding rectangles ($mbrs$). If it fails, a more expensive 1-object filtering is applied. In a 1-object filtering, an object in a candidate is retrieved and a test is performed to determine if the candidate is in the answer. Experiments are also conducted to investigate properties of these techniques. Only when a candidate fails in both filterings, the most expensive $minDist$ operation is invoked. To show the effectiveness of the filtering techniques, they are incorporated into the well-known tree join algorithm and tested with real-life as well as synthetic data sets. Extensive experiments show that the proposed algorithm outperforms existing techniques by a wide margin in both the execution time as well as IO accesses. More importantly, the performance gain improves drastically with the increase of distance value.

This paper is organized as follows. The next section surveys related work. Section 3 gives some definitions and briefly outlines the experimental environment. In Section 4, a modified tree join algorithm is derived for evaluating a buffer query. In Section 5, we introduce the 0- and 1-object filtering techniques. Section 6 presents an efficient $minDist$ algorithm for line and region objects and shows its desirability. To evaluate the proposed filtering techniques, they are incorporated into the modified tree join algorithm. Extensive experiments are performed with both real-life and synthetic data sets. The experimental results are summarized in Section 7. Finally conclusion and future research direction are given in Section 8.

2 Related Work

Most work on spatial join processing fall into the 3-step framework proposed in [4]. Let us call these steps *MBR-join*, *filtering* and *refinement*. In the first step, commonly with the help of spatial indexes, a set of *candidates* is produced. These candidates are generated based on their *mbrs*. In the filtering step, candidates are examined with some geometric filters. The purpose is to identify as many hits as well as false hits as possible. As a result, candidates are partitioned into three sets: *hits* that fulfil the join predicate, *false hits* which are proven *not* to be in the answer set, and *remaining* or *filtered candidates* which possibly satisfy the join predicate. The filtered candidates are examined in the refinement step by invoking an efficient geometric algorithm to the objects involved. The refinement step is likely the most costly operation as the geometric algorithm is CPU-intensive and both objects are required to be retrieved from the disk.

There exists a variety of algorithms for performing the *MBR*-join [1, 3, 10, 15, 17]. Some work have been done on the filtering step by employing progressive approximation [9, 4], by exploiting symbolic intersection detection [13] and by raster approximation [19]. All the above-mentioned work concentrate on the intersection operator. The exact geometric processing in the refinement step is commonly implemented with efficient plane-sweep algorithms. See for instance [16, 11, 5, 2]. To facilitate the processing in the refinement step, objects are decomposed into smaller pieces [4], by arranging or partitioning the data on disk so as to minimize the chance of a page fault [9], or by reading in as many objects in one set so that duplicate retrieval can be minimized [17].

Other related work that deals only with point objects include work on nearest neighbor queries [18] and closest pairs queries [7]. Algorithms are proposed in these work for finding closest pairs from two point data sets. In addition, cache size and caching scheme are investigated in [7] to see how they affect the performance. As will be shown in Section 5.1, some of the techniques employed in these work are also applicable to buffer query evaluation. Recently, the *distance join* operator is proposed in [14]. The distance join is a general approach for solving distance-related queries by ordering the tuples output according to values produced by a distance function. Theoretically, together with a *minDist* algorithm, it can be used to evaluate a buffer query. However, as generality is their primary concern, they are not addressing the same problem as in this work. For instance, a problem with that algorithm is the efficient implementation of the disk-based priority queue [14, 7]. Even if an efficient disk-based priority queue can be implemented, that approach to buffer query evaluation is very inefficient. As will be seen later, the key to solving buffer query evaluation problem is to *minimize* the number of invocation of *minDist* operations. Additional techniques, like the ones proposed in this work, are required to speed up the evaluation process. To the author's best knowledge, there is *no* work done on buffer query evaluation.

The *minDist* problem between two convex polygons was studied in [6]. Their algorithm is based on the concept of *visibility* and is more complex than our proposed $minDist$ algorithm. As their work is of theoretical interest, no performance evaluation is performed on their algorithm.

3 Notation, Test Data and Environment

Let us first define what are non-point objects in the 2D space. A *chain* of segments or simply a chain, is a finite sequence of segments such that any two adjacent segments share an endpoint and no endpoint belongs to more than two segments. A chain is said to be *simple* if there is no point other than an endpoint that is shared by two or more segments. Informally, a chain is simple if there is no pair of segments crossing over each other and no branching in the chain. A chain is said to be *closed* if the two endpoints of the chain are the same. A *line* is a simple chain of segments while a *region* is an area or the point set enclosed by a simple closed chain. Vertices in a region are arranged in the clockwise direction.

An *mbr* is denoted by $((xmin, ymin), (xmax, ymax))$. An *mbr* m is *expanded* by d units is the *mbr* obtained from m by incrementing the *xmax*, *ymax* and decrementing the *xmin*, *ymin* by d units. Given an *mbr* m, the *NE corner quadrant* of m is the space $\{(x,y) \mid x \geq m.xmax$ and $y \geq m.ymax\}$. *NW*, *SE* and *SW* corner quadrants are defined in a similar manner. Given another disjoint *mbr* n, n is said to be in *X corner quadrant* of m if n is completely contained in the X corner quadrant of m. An *mbr* n is said to be in E quadrant if n is *not* in a corner quadrant of m and $m.xmax \leq n.xmin$. An *mbr* n is in W, N, or E quadrant of m is defined in a similar fashion.

There are at least two definitions of minimum distance ($minDist$) between non-point objects.

Centroid: The $minDist$ between two geometric objects is defined as the Euclidean distance between the centroid of the objects. The centroid is the arithmetic mean of vertices of the objects involved.

Point Set: The $minDist$ between two geometric objects is defined as minimum of $\{dist(p_1, p_2) \mid p_1$ is a point of o_1 and p_2 is a point of $o_2\}$, where *dist* is the Euclidean distance function.

The centroid-based semantic is easy to compute but may not capture the minimum distance correctly. Throughout the discussion, we shall assume the *Point Set* Definition.

There are four sets of real-life geometric data used in the experiments. They are provided by the Faculty of Environmental Studies at the University of Waterloo. The area covered has the size of 60000 * 57000 units. Information on these data sets are summarized in Figure 1. These data sets include both lines and regions and have distinct characteristics. The building data set is relatively small and simple and has the lowest average number of segments. The vegetable data set is the largest, both in terms of its average *mbr* size as well as the average number of vertices per object. An average drainage object has more vertices than that of road but has a smaller *mbr*. Figure 2 summarizes the environment under which the experiments are carried out.

Data Set	Type	No. of Objects	Ave. no. of Segments	Ave. Mbr Width	Ave. Mbr Height	Text File Size (in Mbytes)
Building	Region	8860	7.3	42	41	2.32
Road	Line	13580	10.3	175	158	5.97
Drainage	Line	15596	26	139	131	14.6
Vegetation	Region	4579	81	299	277	12.8

Figure 1: Test Data Sets Information

Property	Value
Machine	IBM ThinkPad 770Z, mobile PII 366 MHz, 256MB SDRAM, 14.1GB.
O.S.	Window 98 2nd Edition
Java compiler and VM	JBuilder 3.0 with JDK 1.2

Figure 2: Experiment Environment Details

4 Buffer Query Evaluation

In this section, a modified tree join algorithm is presented for evaluating a buffer query. The correctness of this algorithm is based on the fact that two objects are within distance d of each other exactly when their $minDist$ is.

4.1 Framework

Throughout the discussion, variants of R-trees [12] are assumed to be built on the geometric attributes. In our implementation, ordered Hilbert R-trees are used [8] and the main data files contain the geometric objects. An R-tree is said to be *ordered* if the objects in the main data file have the same relative order as their corresponding leaf entries. The spatial query processing framework assumed is the 3-step spatial join processing proposed in [4], as was discussed in Section 2. In this work, the filtering step produces no false hits while in the refinement step, a $minDist$ algorithm presented in Section 6 is applied to the objects involved.

4.2 A Buffer Query Evaluation Algorithm

If R-tree variants have been built on the geometric attributes, a spatial join algorithm can be used to perform *MBR*-join [3]. Since existing spatial join algorithms are designed for the intersection operator, modifications are required so that only pairs whose $minDist$ is (likely) less than or equal to the given d are in the candidate set.

The following is a modified tree join algorithm for evaluating a buffer query, given a distance

d, for two data sets that are represented by two R-tree variants. *Node* is a data type or class denoting a node in an R-tree. Each node contains a number of entries and each entry has an *mbr* and has a *child*: for leaf nodes, the child points to a geometric object in the main data file while for non-leaf nodes, it points to a node in the tree. Let us assume further that for each child, there is a function *retrieve()* that retrieves the object or node pointed at by the child. The algorithm *findMBRCandidatePairs* returns a subset of Cartesian product of entries from the two nodes R and S such that their *mbrs* are likely within distance d. A more detailed discussion on this algorithm is presented in Section 4.3. The function *minDist* accepts two geometric objects and returns the minimum distance between them. An efficient way of evaluating this function is introduced in Section 6.

Algorithm bufferQueryTJ(*Node R***,** *Node S***,** *double d***,** *File resultSet***):** Find elements in the Cartesian product of pointers to objects in the two data sets that are within distance d of each other. R and S are roots of two R-trees variants for two data sets A and B, respectively. The pairs that are in the query answer are stored in a file resultSet.

Input: A file *resultSet*, a distance d, R and S are roots of two R-trees representing the two data sets A and B, respectively.

Output: *resultSet*.

Method:

```
candidates = findMBRCandidatePairs(R, S, d);
for each pair <r, s> in candidates do:
    if (R is a leaf)
        if (S is a leaf)
            if minDist(r.child.retrieve(), s.child.retrieve())≤d
                append <r.child,s.child> to resultSet;
        else /*R is a leaf while S is not. */
            windowQuery(s, r.child, d, resultSet)
    else if (S is a leaf) /*S is a leaf but not R.*/
        windowQuery(r, s.child, d, resultSet)
    else /* both are non-leaf.*/
        bufferQueryTJ(r.child.retrieve(), s.child.retrieve(), d, resultSet)
end /*for*/
```

Algorithm windowQuery(*NodeEntry n***,** *GeometricObjectPtr p***,** *double d***,** *File resultSet***):** Find objects in the subtree n that are within distance d of the object pointed at by p. Store the result in resultSet.

Input: A node entry n, a pointer p to a geometric object, a distance d, and a file *resultSet* storing the result.

Output: *resultSet*.

Method:

```
(1)  let o and r be p.retrieved() and the mbr of o, respectively;
(2)  if (n is a leaf entry)
(3)      if (r and n.mbr is a candidate)
(4)          if (minDist(n.child.retrieve(), o)≤d
(5)              if o∈A append <p,n.child> to resultSet,
(6)              else append <n.child,p> to resultSet;
(7)  else /*n is a nonleaf entry*/
(8)      for each entry k in n.child.retrieve() do
(9)          if (k.mbr intersects r)
(10)             windowQuery(k, p, d, resultSet);
(11)     end /*for*/
```

4.3 MBR-join

In *findMBRCandidatePairs* as well as in statement (3) in *windowQuery*, one needs to determine if a pair of *mbrs* is a candidate. There are at least two ways to test if a pair of *mbrs* are (likely) within distance d of each other:

1. (*Expansion*). Select one *mbr* and expand it by d units. If the expanded *mbr* intersects with the other, the pair is a candidate.

2. (*MBRminDist*). Compute their *minDist*. The *minDist* between *mbrs* can be computed with the *minDist* algorithm in Section 6 or more efficiently, by determining their relative quadrants and compute the distance of the closest pair of points. An outline of the more efficient algorithm *minDist* is given below.

Algorithm minDist(*mbr r*, *mbr s*): Compute the *minDist* between two *mbrs*.

Input: Two *mbrs*.

Output: The *minDist* between the two *mbrs*.

Method:

```
/* quadrant is one of NE, NW, SE, SW, N,E,S and W. */
(1)  if the two mbrs intersect, return 0;
(2)  find quadrant in which s is in relative to r.
(3)  switch (quadrant)
(4)      case NE: /* s is in NE corner quadrant of r. */
(5)          return dist(r.getNE(), s.getSW());
:        :
(12)     case S: /* s is in the South quadrant of r.*/
(13)         return r.ymin- s.ymax;
:        :
(20) end /*switch*/
```

As pairs produced by the $MBRminDist$ method are pairs of the *Expansion* method, but not vice-versa, the $MBRminDist$ method has a smaller candidate set. However, the *Expansion* method has the advantage of fast computation.

To evaluate these strategies, two algorithms are implemented by incorporating *Expansion* and $MBRminDist$ into the *bufferQueryTJ* algorithm:

1. (*Expansion with restricted search space*). This algorithm is outlined below as *findMBRCandidatePairsExpansion*. The algorithm *intersectionTest* is the *SpatialJoin2* algorithm in [3] with the following modification: if a pair of *mbrs* intersect, a tuple of node entries corresponding to the two *mbrs* is added to candidatePairs. Plane-sweep is not included in *intersectionTest* since our experiments show that it is beneficial only for relatively small d.

2. ($MBRminDist$). Same as *findMBRCandidatePairsExpansion* except that whenever a candidate is produced in *intersectionTest*, the *mbrs* are tested to see if their *minDist* is less than or equal to d as well. They are a candidate if they pass the test.

The *findMBRCandidatePairs* in *bufferQueryTJ* is replaced by the algorithms above. The statement (3) of *windowQuery* is also modified accordingly.

Algorithm findMBRCandidatePairsExpansion(*Node R*, *Node S*, *double d***):** Find elements in the Cartesian product of entries in two nodes that are potentially within distance d of each other. Two entries are potentially within distance d if their *mbrs* are. The satisfying pairs are stored in *candidatePairs* and returned to the calling program.

Input: Two R-tree nodes.

Output: *candidatePairs*.

Method:

(1) let m and n be lists of *mbrs* from nodes R and S, respectively;
(2) without loss of generality, let m have fewer entries than n;
(3) for each r in m, expand its *mbr* by d units;
(4) intersectionTest(m,n, *candiatePairs*);
(5) return *candidatePairs*;

The algorithms are evaluated, with various distance values and different combinations of data sets, on the computation time as well as the size of candidate set output. The computation time is the time to compute the *MBR*-join candidate set (i.e., without filtering nor refinement). The experimental result on candidate set size is summarized in Figure 3. The values in the graph are the ratios of the size of candidate set produced by *Expansion* to that generated by $MBRminDist$. The computation time ranges from 18 to 88 seconds. The differences in computation time between

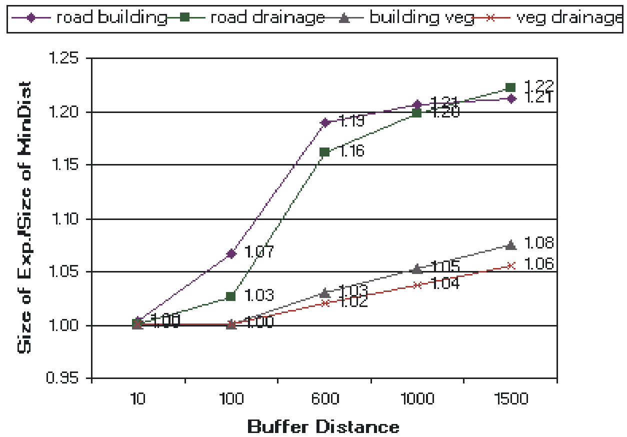


Figure 3: MBR-join Techniques Evaluation

the two algorithms is negligible and thus is not included here. The experiment shows that the $MBRminDist$ algorithm is preferred between the two, independent of distance values and data types. The extra computation is negligible and well-justified with the reduction of the candidates produced. As showed in Section 7, *MBR*-join accounts for a small fraction of the total query evaluation time and thus it is vital to minimize the size of candidate set to improve the performance of query evaluation. For the rest of the discussion, the *findMBRCandidatePairs* is the $MBRminDist$ algorithm above.

5 Filtering Techniques

A problem with *bufferQueryTJ* is that it is very inefficient and that the time to compute the result is long, even for relatively small data sets. The main source of inefficiency is that in step (5) of *bufferQueryTJ* and in step (4) of *windowQuery*, *minDist* is invoked on candidates even if they can easily be determined as hits. To overcome this deficiency, geometric filterings are incorporated in buffer query evaluation. In this section, filtering techniques with different costs are presented to reduce the computation as well as IO time.

5.1 0-object Filtering

Like buffer queries, nearest-neighbor queries [18] and closest-pair queries [7] are distance-related queries. Efficient techniques have been developed for evaluating these classes of queries. Although the above-mentioned work are dealing with points only, some techniques are applicable to non-point data sets. A metric that is useful to buffer query evaluation is the $MinMaxDist$ metric [7].

Suppose r and s are two *mbrs* in an *R*-tree. Let $\{r_1, r_2, r_3, r_4\}$ and $\{s_1, s_2, s_3, s_4\}$ be the sets of edges for r and s, respectively. The metric $MinMaxDist$ is defined as follows:

$MinMaxDist(r,s) = min_{i=1,j=1}^{i=4,j=4}\ \{maxDist(r_i,\ s_j)\}$.

$maxDist(r_i,\ s_j$

Lemma 5.1 *Given two node entries r and s and a distance d, if d* $\leq MinMaxDistMBR$*(r.mbr,s.mbr), then there is an object o in the subtree r such that for every object p in s, minDist(o,p)*$\leq$*d.*

[**Proof**]**:** As there is at least a point of an object o is on an edge r_i, it follows that every object p in subtree s are within distance $maxDist(r_i,\ s)$ of o. □

Unlike the metric $MinMaxDist(r,s)$, $MinMaxDistMBR(r,s)$ is asymmetric. The metric $MinMaxDistMBR(r,s)$ is useful when r is a leaf entry and s is a non-leaf entry, and when $d \leq MinMaxDistMBR(r.\text{mbr},s.\text{mbr})$. In this case, all objects in the subtree of s are within distance d of the object pointed at by entry r. This could be used in the algorithm *windowQuery*.

Again let r and s be two *mbrs*. The metric $maxDist(r,s)$ is defined to be the maximum distance of any two points contained in r and s [7]. It is useful when both are *mbrs* of non-leaf nodes. In this case, if $maxDist(r,s) \leq d$, then all pairs of entries in the two subtrees are within distance d of each other.

The metrics $MinMaxDistMBR$ and $maxDist$, when applied in *MBR*-join, are redundant in the sense that $MinMaxDist$ alone produces the same candidate set. Nevertheless, these two metrics could reduce computation time, especially when the buffer distance is large.

The above metrics provide sufficient conditions to determine if objects in a candidate are within distance d without retrieving the actual objects. Let us call a sufficient condition or technique for a candidate to satisfy a buffer distance condition in which exactly x objects are retrieved or accessed an *x-object filtering*. The above 0-object filtering techniques can easily be incorporated into a spatial join algorithm without much cost. As will be shown later, they are very effective, especially when the distance is relatively large.

5.2 1-object Filtering

Given a candidate, one could just retrieve both objects and test for the condition. Alternatively, an object from the pair is retrieved and the vertices are tested against the other *mbr* to see if they satisfy the join predicate. Since exactly one object in a candidate is accessed, this is a 1-object filtering technique. The following is an algorithm for implementing this 1-object filtering. The minimum of the maximum distance ($MinMaxDist$) between a vertex and an *mbr* is computed with a formula in [18].

Algorithm MinMaxDist$_{1-obj}$(GeometricObject o, Mbr r, double d): Given a distance d, a geometric object o and an mbr r, determines an upper bound on distance between the two objects. If the upper bound is less than or equal to d return true else false. The upper bound is obtained by finding the minimum of maximum distances between vertices of o and r.

Input: An object o, an *mbr* r, and a distance d.

Output: True if o and r are guaranteed within distance d of each other and false otherwise.

Method:

```
curMin=+∞;
for each vertex v of o, do the following:
    curMin = min { MinMaxDist(v, r), curMin};
    if (curMin≤d) return true;
end /* for */
return false;
```

The 1-object filtering has the potential of avoiding the retrieval of an object as well as elimination of the relatively expensive *minDist* computation. The cost is the extra computation time which is proportional to the number of vertices of the retrieved object. A fundamental question with this technique is which object in a pair should be retrieved to test against the *mbr* of the other object. Let us call the object in a pair that is accessed or retrieved back the *retrieved* object. In the filtering test, the minimum of $\{MinMaxDist(p_i,r) \mid p_i$ is a vertex of the retrieved object and r is the *mbr* of the other object$\}$ is used as an upper bound on the distance between the two objects. Let $MinMaxDist_{1-obj}(o_1,r_2)$ be the minimum $\{MinMaxDist(p_i,\ r_2) \mid p_i$ is a vertex of $o_1\}$.

Consider now two objects o_1 and o_2 with their *mbrs* r_1 and r_2, respectively. Assume further that r_1 is much smaller than r_2. Then $MinMaxDist_{1-obj}(o_1,\ r_2)$ is likely (but not always) to be greater than $MinMaxDist_{1-obj}(o_2,\ r_1)$, as is illustrated in Figure 4. In this example, it is assumed that the closest vertex is in the middle of a boundary edge of an *mbr*. The $MinMaxDist_{1-obj}(o_1,\ r_2)$ and $MinMaxDist_{1-obj}(o_2,\ r_1)$ are denoted by the solid and dashed lines, respectively.

To investigate how the size of an *mbr* influences the performance of this filtering technique, three strategies are implemented. In the first strategy, both objects are retrieved and two filtering tests are performed; one for each object against the other's *mbr*. The test on a candidate is *successful* if

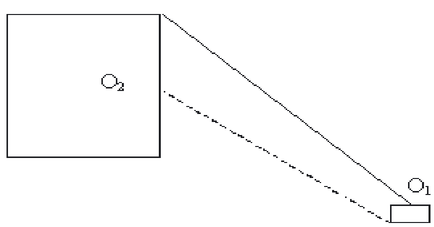
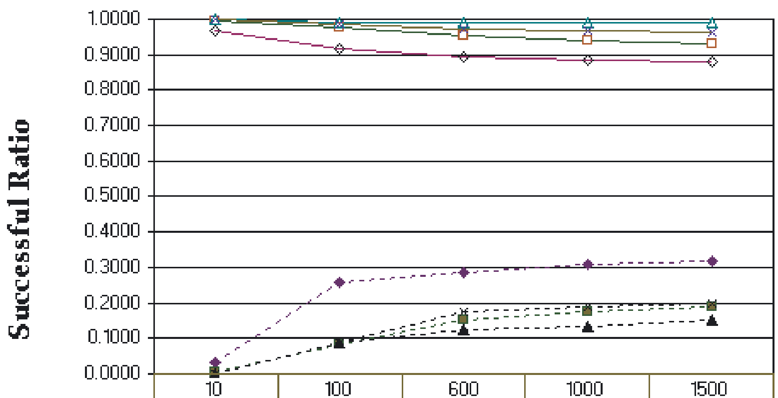


Figure 4: A Small and A Large Mbrs

at least one of the filtering tests produces a value that is less than or equal to the distance value. Let us call this the *perfect* selection. For each distance value, the number of successful tests is collected. Imagine that someone knows which object in a pair should be retrieved *all the times*. Then the number of successful tests in evaluating the buffer query is the same as the number of successful tests of the perfect selection. Thus, the perfect selection represents the strategy that always selects the right object in the pair as the retrieved object. In the second strategy, the larger *mbr* (in term of area) is selected as the retrieved object while the third strategy selects the smaller one. Again a test is *successful* if the filtering test produces a value that is less than or equal to the distance value. The number of successful tests is collected for each strategy in each test. Let us call the second and third strategies the *large* and *small* selection, respectively. The number of successful tests is used to measure the effectiveness of the strategy employed. Clearly the larger the number of successful tests, the better the strategy. The *successful ratio* of a selection (relative to perfect selection) is the ratio of number of successful tests to that of perfect selection. By definition, the successful ratio is less than or equal to 1.

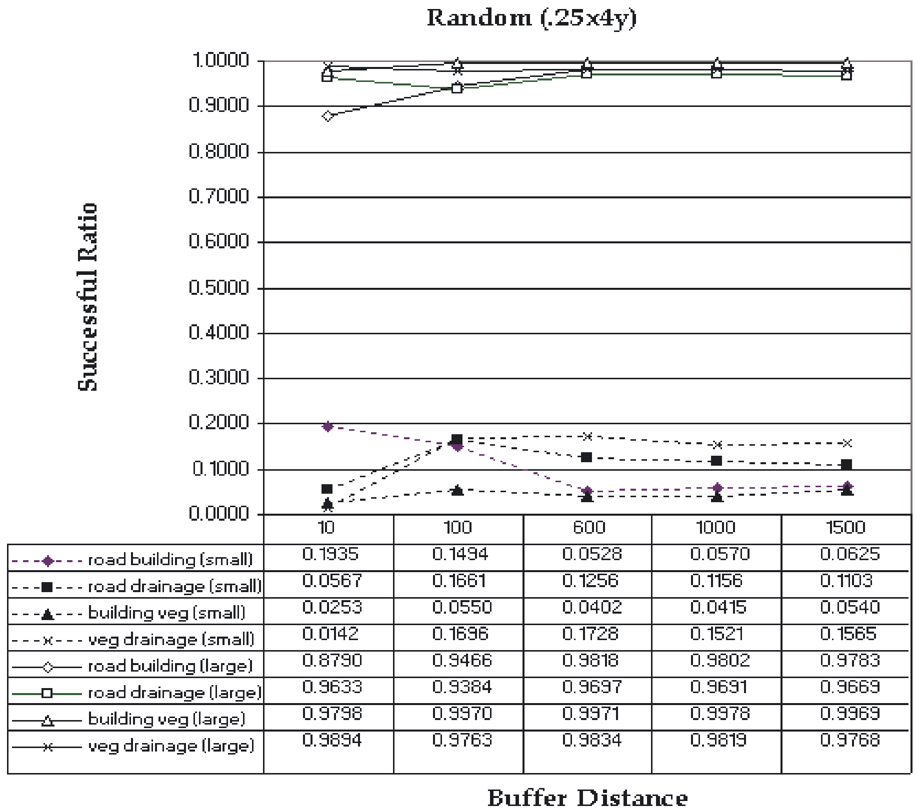


	10	100	600	1000	1500
road building (small)	0.0323	0.2562	0.2856	0.3076	0.3191
road drainage (small)	0.0047	0.0829	0.1514	0.1763	0.1896
building veg (small)	0.0000	0.0865	0.1253	0.1342	0.1499
veg drainage (small)	0.0000	0.0871	0.1757	0.1887	0.2004
road building (large)	0.9677	0.9173	0.8918	0.8833	0.8822
road drainage (large)	0.9969	0.9748	0.9522	0.9396	0.9300
building veg (large)	1.0000	0.9891	0.9904	0.9899	0.9903
veg drainage (large)	1.0000	0.9863	0.9722	0.9672	0.9623

Buffer Distance

Figure 5: 1-Object Filtering Evaluation Result for Real-Life Data Sets

Four pairs of real-life data sets are examined: road-drainage, road-building, veg-drainage, building-veg. The data sets are selected to reflect different possible combinations of data types. Four pair of synthetic data sets are tested: road_random(.25x4y)-drainage_random(.25x4y), road_random(.25x4y)-building_random(.25x4y), veg_random(.25x4y)-drainage_random(.25x4y), building_random(.25x4y)-veg_random(.25x4y). A *x*_random(.25x4y) data set is generated from the corresponding *x* data set by randomly distributed the objects over the covered area. The resulting data set is a uniform distribution of objects over the map area. Moreover, for each object, the width (x-dimension) is scaled to .25 of the original size while the height (y-dimension) is elongated



	10	100	600	1000	1500
road building (small)	0.1935	0.1494	0.0528	0.0570	0.0625
road drainage (small)	0.0567	0.1661	0.1256	0.1156	0.1103
building veg (small)	0.0253	0.0550	0.0402	0.0415	0.0540
veg drainage (small)	0.0142	0.1696	0.1728	0.1521	0.1565
road building (large)	0.8790	0.9466	0.9818	0.9802	0.9783
road drainage (large)	0.9633	0.9384	0.9697	0.9691	0.9669
building veg (large)	0.9798	0.9970	0.9971	0.9978	0.9969
veg drainage (large)	0.9894	0.9763	0.9834	0.9819	0.9768

Figure 6: 1-Object Filtering Evaluation Result for Synthetic Data Sets

4 times its original size. The resulting objects have the *mbr* the same size (area) as the original objects but with a different shape. The synthetic data sets are used to test if different distribution and shape of objects have any influence on the three strategies.

For each pair and for each strategy, tests are performed with distance values 10, 100, 600, 1000 and 1500. The results are summarized in Figure 5 and Figure 6. From the experiment, the large selection clearly outperforms the small selection, over all data sets and buffer distances. In fact, in many cases, the large selection is close to the perfect selection. Among various combinations, the large strategy is the most effective for building-veg combination. Most objects in vegetation data set have a much larger area than the building objects and thus vegetation data objects are likely be selected as the retrieved objects. Moreover, vegetation data objects are region and most vertices in a vegetation object form a ring that is close to the boundary of *mbr* than with a line. This also helps explain why the veg-drainage has the second best performance in the large selection. The differences between these two combinations are likely due to the larger size and line type of drainage data set. In sum, for both real-life and synthetic data sets tested, and for all buffer distances, the 1-object filtering strategy based on larger *mbr* is very effective. From now on, the large selection is used in the 1-object filtering.

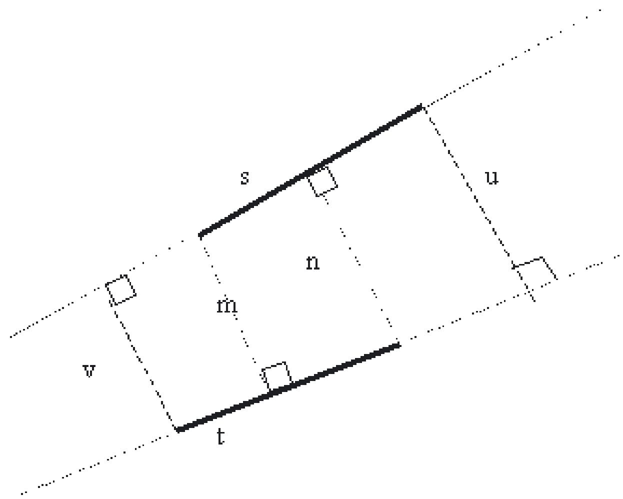


Figure 7: Distance Between Segments

6 Minimum Distance Algorithms

In this section, we investigate the problem of computing the *minDist* between two non-point geometric objects. Clearly if two objects intersect, the *minDist* is zero. From now on, objects are assumed to be disjoint when *minDist* is considered. A plane-sweep algorithm could be invoked to determine if two non-point objects are disjoint [5].

6.1 Minimum Distance Between Points and Segments

Suppose x is a point and s a segment. Let $pp(x, s)$ be the perpendicular line to s that passes through x. To determine the *minDist* of a point w to a segment $s= \{u,v\}$, where u and v are its endpoints, generate a line $pp(w,s)$. If $pp(w, s)$ intersects s at a point q, then the *minDist* between w and s is the distance from w to q. Otherwise the $min(dist(w,u), dist(w,v))$ is the *minDist* of w from s.

Now consider two segments $s= \{s_1,s_2\}$ and $t= \{t_1,t_2\}$. Two endpoints s_i and t_j, one from each segment, is said to be the *closest* if their distance is shortest among all such pairs. Let u be an endpoint of q. Either $pp(u,z)$ intersects z at p, where $z \neq q$ and z and q are the two segments involved, or it does not. In the former case, let us call the segment between s_1 and p an *endpoint perpendicular segment*. In Figure 7, segments n and m are endpoint perpendicular segments and are the only endpoint perpendicular segments between s and t.

Lemma 6.1 *Let s and t be two segments. The minDist(s, t) is the minimum of the distance of closest enpoints and the length of the shortest endpoint perpendicular segment.*

[Proof]: If s and t are parallel, the Lemma follows. Suppose s and t are not parallel. Then the extended lines intersect at some point i with an angle θ. Without loss of generality, all points of s

are on the same side on the extended line with respect to the point i. Similarly for t. If θ is greater than or equal to 90°, then it can be shown easily that the mimimum distance is between the closest endpoints and the Lemma follows. Now asssume θ is less than 90°. Image sweeping a perpendicular line segment m to t from the intersecting point i toward the two line segments s and t until (i) endpoints of m are on the segments s and t, respectively, and (ii) the endpoint on s is an endpoint of s, as is illustrated in Figure 7. If such m exists, the distance is the shortest distance between any pair of points on s and t. First observe that the distance is the shortest between the endpoint of s and any point on t. For any point q of s that is not the endpoint of m, it should be clear that it cannot be an endpoint of the shortest segment. If such m does not exist, repeat the same argument by sweeping a perpendicular line segment n to s. If both m and n do not exist, then the shortest distance is between the closest endpoints (s_i, t_j) of s and t. To prove this claim, consider a perpendicular line v to s with an endpoint anchored at the closest endpoint t_j of t, as shown in Figure 7. Consider now the endpoint of v on the extended line of s moves toward s, the length of the line increases. Thus the shortest distance between t_j and any point of s is the closest endpoint in s. By a similar argument, the shortest distance between s_i and any point of t is the closest endpoint in t. Suppose there is a segment w with a distance shorter than the closest enpoints. Observe that endpoints of w may be moved so that the segment is shortened. If w cannot be shortened further, at least one of its endpoints is one of s_i or t_j, or w is perpendicular to one of s or t. The former case is not possible since we have already shown that the shortest segment involving s_i or t_j is the segment (s_i, t_j). Let us assume w is perpendicular to s. Moves this segment toward s_i and the segment length decreases. A contradiction. It follows that, the condition computes correctly the $minDist$ between two segments. □

6.2 Minimum Distance between Objects

If both objects are lines, the $minDist$ is the minimum of $minDist$ between all pairs of segments from the two objects. If one of them is a region, then the shortest distance between the region object and the non-point object is the $minDist$ between the boundary of the region object and the non-point object. Thus the problem of determining the $minDist$ between non-point objects is reduced to the problem of determining the $minDist$ between two line objects. The above observation gives rise to an algorithm that determines the $minDist$ between two non-point objects.

Algorithm GenMinDist: Given two disjoint sets of segments, compute the $minDist$ between them.

Input: Two disjoint set of segments.

Output: The minimum of $minDist$ between segments from the two sets.

Method:

(1) Let $globalMin$ be set to $+\infty$.

(2) For each segment s of one set, perform steps 3 and 4:

(3) For each segment t of the other set, determine the $minDist$ d between s and t.

(4) $globalMin = min(d, globalMin)$.

(5) return $globalMin$.

The time complexity is $O(n \times m)$, where n and m are the number of segments in each object, respectively. In what follows, a more efficient way of computing the $minDist$ between two simple chains is presented.

6.3 A minDist Algorithm for Simple Chains

The algorithm *GenMinDist* applies to sets of segments that are pairwise disjoint. However, the segments in a set need not be a chain nor is simple. In this subsection, an algorithm is presented for finding $minDist$ between two simple chains.

Consider two disjoint simple chains, the main idea of the algorithm is to identify sub-sequences of chains, which are called *frontiers*, for computing $minDist$ between the two objects. The important property of a frontier of a simple chain is that computing $minDist$ with the frontier is the same as computing $minDist$ with the whole chain.

To simplify the presentation, it is assumed throughout in this subsection that the *mbrs* of two disjoint simple chains are themselves disjoint. The algorithm can be extended to the case where their *mbrs* are overlapping.

Let C_1 and C_2 be two disjoint simple chains. The chain C_1 is said to be in X quadrant (corner quadrant) of C_2 if C_1's *mbr* is in X quadrant (corner quadrant, respectively) of the *mbr* of C_2. A vertex in a chain c is said to be a *touching* vertex if it is a point on a boundary of the *mbr* of c. The frontier for a simple chain is bounded by two touching vertices. To illustrate how a frontier is found, we first consider C_1 and C_2 are simple closed chains.

6.3.1 $minDist$ For Simple Closed Chains

In this subsection, we show how to compute $minDist$ for simple closed chains. We then extend the idea to simple chains in the following subsection.

The X *frontier* of a simple closed chain C_1 is defined by two touching vertices which are located as follows, where X is one of the four corners of an *mbr*: At the corner X of the *mbr*, there are two incident edges. The edges can be ordered with respect to the center of the *mbr* in clockwise direction: assign increasing numbers to edges with the restriction that the numbers of two incident edges at the corner are consecutive. The smaller is the *begin* while the larger is the *end* edge. Starting at the corner X, search along the begin edge to locate the first touching vertex. The

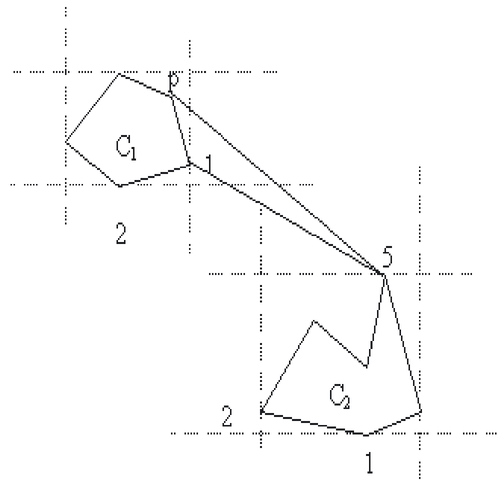


Figure 8: An Example

vertex found is the *begin* vertex of the frontier. Likewise the *end* vertex is found by searching along the end edge, starting at corner X, for the first touching vertex. The two touching vertices guarantee to exist as each edge must have at least one point from C_1. The sub-chain from begin to end vertices is the X frontier of C_1. Note that the sub-chain from the end vertex to the begin vertex is *different* from the X frontier of C_1. The portion of begin (end) edge that is between the corner X and begin (end) vertex is said to be *covered* by the frontier. In Figure 8, C_1 is in NW corner quadrant of C_2. Or equivalently, C_2 is in the SE corner quadrant of C_1. The SE frontier of C_1 is the sub-chain from vertex 1 to vertex 2 while the NW frontier of C_2 is the sub-chain from vertex 2 to vertex 5. It can be shown that the $minDist$ between these frontiers is the $minDist$ between the two objects. Observe that the point p on C_1 is not on the SE frontier of C_1 and its distance from any point q in C_2 is longer than that from the begin vertex 1 to q. This leads to the following.

Lemma 6.2 *Suppose C_2 is at the X corner quadrant of C_1. Let q be a point of C_2 and p a point on the begin (end) edge of X corner of C_1 that is not covered by the X frontier of C_1. Then dist(q, p)>dist(q, u), where u is the begin (end, resp.) vertex of the X frontier of C_1.*

[**Proof**]**:** Since the begin (end) vertex and p are on the same edge of an *mbr*, one of x- or y-value are the same. Without loss of generality, let their x-values be the same. By the assmuption that p is not in the covered portion and due to the relative position of C_1 and C_2, the difference of y-value between q and p must be greater than the y-value difference between that of q and u. Thus the Lemma follows. □

Corollary 6.3 *Suppose C_1 is in X corner quadrant of C_2 and C_2 is in Y corner quadrant of C_1.*

Then the minDist between the Y frontier of C_1 and X frontier of C_2 is the minDist between C_1 and C_2.

[**Proof**]**:** Suppose at least one of two closest points from C_1 and C_2 is not on the corresponding frontier. By assumption on the relative position of C_1 and C_2, if the straight line joining the closest points intersects a boundary edge of an *mbr*, it must be a begin or an end edge. By assumption on the closest points, the straight line joining them passes through a point on the boundary that is not covered by the corresponding frontier. By Lemma 6.2, this pair cannot be the closest. A contradiction. □

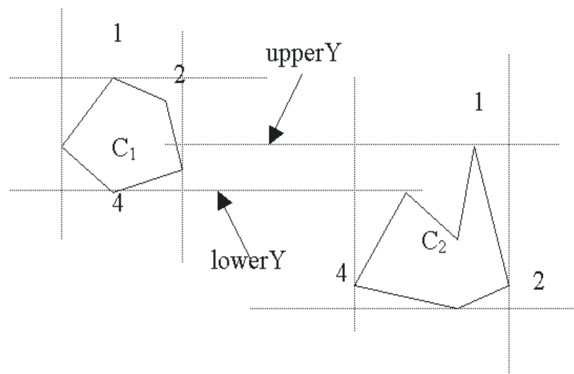


Figure 9: An Example

Suppose C_1 is in the *W* quadrant of C_2, as shown in Figure 9. Or equivalently, C_2 is at the *E* quadrant of C_1. Define *upperY* as $min(C_1.ymax,\ C_2.ymax)$ and *lowerY* as to $max(C_1.ymin,\ C_2.ymin)$. The *upperY* and *lowerY* denote the overlapping range along the Y-axis for the two *mbrs*. The *E* frontier of C_1 and the *W* of C_2 are determined as follows.

The *E* (*W*) frontier for a simple closed chain C is identified as follows:

Search the *E* (*W*) edge of C for the touching vertex with y-value *just* greater than or equal to *upperY*. If there is no such touching vertex on *E* (*W*) edge, search *N* edge westward (eastward), starting from the *NE* (*NW*) corner, for the first touching vertex. The touching vertex found is the begin (end) vertex for the *E* (*W*) frontier. Search the *E* (*W*) edge of C for the touching vertex with y-value *just* less than or equal to *lowerY*. If there is no such touching vertex on *E* (*W*) edge, search *S* edge westward (eastward), starting from the *SE* (*SW*) corner, for the first touching vertex. The touching vertex found is the end (begin) vertex for the *E* (*W*) frontier. In Figure 9, the *E* frontier of C_1 and *W* frontier of C_2 are vertices 1 to 4 and vertices 4 to 1, respectively.

Lemma 6.4 *Suppose C_1 is on the W quadrant of C_2. Or equivalently, C_2 is on the E quadrant of C_1. Then the minDist of E frontier of C_1 and the W frontier of C_2 is the minDist between C_1 and C_2.*

[Proof]: We want to show that for any pair of points from C_1 and C_2, they are points on their corresponding frontiers, if their distance is the shortest. We prove this by considering all possible cases of *upperY* and *lowerY*. It is sufficient to consider cases in which at least one of the points is on an edge of the *mbr* that is not covered by a frontier.

Case 1: $C_1.ymax = upperY$ and $C_1.ymin = lowerY$. The begin and end vertices of C_1's E frontier are on N and S edges, respectively. Without loss of generality, let p be a point on C_1's N edge that is not covered by the E frontier. The argument for p on the S edge not covered by E frontier is similar. We do not need to consider the case that p is on the W edge. Let q be a point in C_2 and let u be the C_1's begin vertex of E frontier on the N edge. By definition of begin vertex of E frontier, p and u have the same y-value but the x-value of u is greater than that of p. By assumption, the x-value of q is greater than that of u and thus the difference in x-value between p and q is greater than that of between u and q. This implies $dist(p, q) > dist(u, q)$. Thus if p is involved in the closest pair, it must be on the frontier. By a similar argument, it can be easily shown that q must be on the corresponding frontier if it is in the closest pair.

Case 2: $C_1.ymax = upperY$ and $C_1.ymin \neq lowerY$. The begin vertex of E frontier is on the N edge while the end vertex is either on the E or S edge. Let p be a point on C_1's N edge that is not covered by the E frontier. By a similar argument in Case 1, p cannot be involved in the pair the distance of which is the shortest. Suppose p is a point on the E or S that is not covered by the frontier. Let q be a point in C_2. By assumption on *lowerY*, the y-value of q is greater than or equal to *lowerY*. Let u be the end vertex of the E frontier of C_1. By definition of end vertex of E frontier, one of x- or y-values of u is greater than while the other equal to that of p. Since the x- and y-values of u are less than or equal to that of q, $dist(p, q) > dist(u, q)$. Thus if p is in the closest pair, it must be on the corresponding frontiers. By a similar argument, it can be easily shown that q must be on the corresponding frontier if it is in the closest pair.

As all other cases are analogous to a case above, we have shown that the frontiers are sufficient to determine *minDist* between C_1 and C_2. $\square$

Suppose now two simples closed chains are in north-south position. Define *upperX* as $min(C_1.xmax, C_2.xmax)$ and *lowerX* as to $max(C_1.xmin, C_2.xmin)$. The N (S) frontier for a simple closed chain C is identified as follows:

Search the N (S) edge of C for the touching vertex with x-value *just* greater than or equal to *upperX*. If there is no such touching vertex on N (S) edge, search E edge southward (northward), starting from the *NE* (*SE*) corner, for the first touching vertex. The touching vertex found is the end (begin) vertex for the N (S) frontier. Search the N (S) edge of C for the touching vertex with x-value *just* less than or equal to *lowerX*. If there is no such touching vertex on N (S) edge, search W edge southward (northward), starting from the *NW* (*SW*) corner, for the first touching vertex. The touching vertex found is the begin (end) vertex for the N (S) frontier.

The proof that the frontiers identified are sufficient to determine $minDist$ is similar to Lemma 6.4.

6.3.2 $minDist$ For Simple Chains

In the above discussion, the objects involved are simple closed chains. If the objects are simple non-closed chains, then some modifications are required. For simple non-closed chains, it is assumed that vertices are number consecutively. However, they are not required to be arranged in clockwise direction. Consider the line object l in Figure 10.

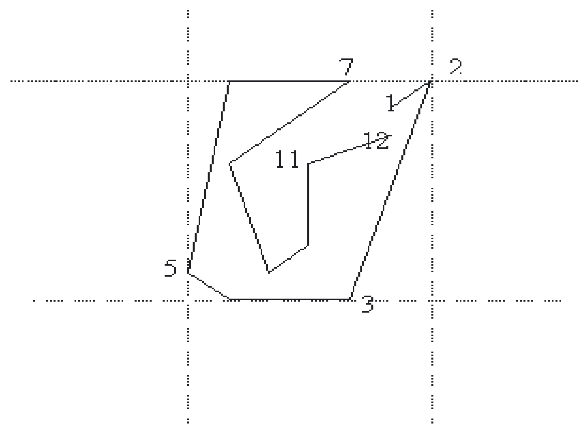


Figure 10: A Line

Suppose another object o is in the N quadrant of l with $o.xmin<l.xmin$ and $o.xmax>l.xmax$. With the algorithm in Section 6.3.1, the begin and end vertices identified are vertices 2 and 5, respectively. Consider the continuous sub-line between the begin and end vertices. Let us call this the *initial* N frontier of l. For simple non-closed chains, it is not important the vertices identified are in clockwise direction since there is only one continuous sub-line between the begin and end vertices. The sub-line from vertex 1 to vertex 2 and the sub-line from vertex 5 to vertex 12 are called *dangling lines*. In this case, one more task needs to be performed in identifying the frontier. For each dangling sub-line, test to see if it is in-between the initial frontier identified and the other object. It is included as part of the frontier exactly when it is in-between the other object and the sub-line vertex 2 to vertex 5. By assumption that the chain is simple, either all points of a dangling line is in-between the frontier and the other object or no point is. In the example above, both dangling lines are included in the frontier and thus the whole line is the N frontier for the line object l. The correctness follows from the proof in the previous subsection and the fact that a chain is simple. On the other hand, if the other object o is in the S quadrant of the above line object l with $o.xmin<l.xmin$ and $o.xmax>l.xmax$. The S frontier for l is the sub-line from vertices 2 to 5.

So far we consider objects whose $mbrs$ do not overlap. However, the algorithm can be extended in a straight-forward manner to disjoints objects whose $mbrs$ overlap. This has been incorporated into our implementation. From now on, we call this the *MinDist* algorithm.

6.4 Performance Evaluation of GenMinDist and MinDist algorithms

To evaluate their effectiveness, both *MinDist* and *GenMinDist* algorithms are implemented and performance evaluation is performed on them with the data sets presented in Section 3. In both approaches, the most important operation in computing the *minDist* between two objects is determining the *minDist* between a pair of line segments. Let us call such an operation a *segment calculation*. Thus in evaluating the performance of the two algorithms, we compare the number of segment calculations as well as the total computation time required.

	GenMinDist		**MinDist**		**Comparison**	
	Time (sec)	Segment Calculation	Time (sec)	Segment Calculation	(A)/(C)	(B)/(D)
	(A)	(B)	(C)	(D)		
Road Building	83	649353	14	84443	592.86%	768.98%
Road Drainage	240	2661756	63	669185	380.95%	397.76%
Building Vegetation	493	5073169	48	341532	1027.08%	1485.42%
Vegetation Drainage	1507	20432654	211	2573033	714.22%	794.11%
Road Vegetation	625	8181409	102	1141582	612.75%	716.67%
Building Drainage	204	1626991	31	206270	658.06%	788.77%

Table 1: GenMinDist and MinDist Comparison

A test involves two distinct data sets. A set of randomly selected pairs of objects from the two data sets of size *sampleSize* is generated first and used as input to the algorithms. All these objects are main memory resident. The computation time measures only the time required for *minDist* computation on these main memory objects. To avoid any unforeseeable anomaly, this process repeats *noOfSamples* times. The average is used in the result of a test. In the performance evaluation, *noOfSamples* and *sampleSize* are set to 10 and 10000, respectively. Table 1 shows the result of segment calculation and total computation time comparison. There are three sub-tables: one for *GenMinDist*, one for *MinDist* algorithm, and the last is the comparison of *GenMinDist* to that of *MinDist*. *Time* and *Segment Calculation* are the computation time (in sec.) and the number of segment calculations in the *average* of each test. The values in the last sub-table represent the ratio of the values for *GenMinDist* to that of *MinDist*.

Independent of the combinations, the *MinDist* algorithm has a far better performance than the *GenMinDist*. The *MinDist* requires about 1/4 of time in the worst case and about 1/10 in the best case when compared to *GenMinDist*. *MinDist* performs best when both data sets are regions while it performs less impressive when both are lines.

7 Performance Evaluation of Buffer Query With Filtering

In the previous sections, filtering techniques are proposed and a more efficient *minDist* algorithm is presented. In Section 7.1, the filtering techniques are incorporated into *bufferQueryTJ*. It is then evaluated in Section 7.2. For the rest of this paper, the more efficient *MinDist* algorithm is used whenever *minDist* is invoked.

7.1 A Modified Buffer Query Evaluation Algorithm

The 0- and 1-object filterings can be incorporated into the **bufferQueryTJ** easily. Let us call the modified algorithm **bufferQueryPrune**. The modified algorithm is obtained from **bufferQueryTJ** by replacing statements (5) and (6) with the following statements.

```
(5)     if MinMaxDist(r.mbr, s.mbr)≤d /* 0-obj filtering*/
(6)        append <r.child,s.child> to resultSet;
(7)     else /*perform 1-obj filtering */
(8)        if (r.mbr.area()≥s.mbr.area())
(9)           largeObj = r.child.retrieve(); small =s;
(10)       else largeObj =s.child.retrieve(); small =r;
(11)       if (MinMaxDist_{1-obj}(largeObj,small.mbr, d))
(12)          append <r.child, s.child> to resultSet;
(13)       else /* refinement: need to retrieve the small object.*/
(14)          smallObj=small.child.retrieve();
(15)          if minDist(largeObj, smallObj)≤d
(16)          append <r.child,s.child> to resultSet;
```

A similar change is also made to statements (4) to (6) in algorithm **windowQuery**. In addition, 0-object filtering techniques *MinMaxDistMBR* and *maxDist* in Section 5.1 are applied to non-leaf entries as well. If these tests are successful, all leaf entries are retrieved and included in the resultSet.

7.2 Performance Evaluation

7.2.1 Environment

To evaluate the performance of the proposed algorithm, a caching scheme is implemented for swapping in and out geometric objects from a main data file. As the data files and geometric objects are of various sizes, the size of a cache is specified as a percentage of the file size and objects are swapped in and out of the main memory. The replacement scheme used is the well-known *LRU* replacement scheme. In each session, statistics such as execution time and the number of objects swapped in are generated to evaluate the performance of the algorithm.

There are four pairs of real-life data sets: road-building, road-drainage, building-veg, veg-drainage. And there are four pairs of synthetic data sets: road_random-building_random, road_random-drainage_random, building_random-veg_random, veg_random-drainage_random. A x_random data set is generated from the corresponding x data set by randomly distributed the objects over the covered area. The resulting data set is a uniform distribution of objects over the map area. The buffer distances are set at 10, 100, 600 and 1500 and they represent very small to very large buffer distances relative to the data sets. The cache sizes are 1%, 20% and 100% of a file size. The 100% cache size is used since we want to determine how many times, on average, an object in other cache sizes are retrieved.

7.2.2 Evaluation of Filtering Techniques

In *BufferQueryPrune*, a candidate is first evaluated with the least expensive technique - 0-object filtering. If it fails, a more costly operation, 1-object filtering, is applied. Finally, if both filterings fail, the most expensive $minDist$ is invoked. To investigate the performance of filtering techniques, three algorithms are implemented and tested. The first is the *BufferQueryTJ* which is without 0- and 1-object filterings. The second is the *BufferQueryTJ* but incorporating the 0-object filtering. The third is *BufferQueryPrune* which incorporates both 0- and 1-object filtering techniques.

Figure 11 summarizes and compares the execution time of these algorithms. The execution time is the *average* execution time for the three different cache sizes. The real-life and synthetic data sets are denoted by dashed and solid lines, respectively. Although there are differences among various combinations, the following are some important general observations. Firstly, except in some combinations with buffer distance equal to 10, *BufferQueryPrune* outperforms the other two algorithms; and in fact, in many cases, by a wide margin. This implies the overhead of the filtering techniques in *BufferQueryPrune* is not costly and is well-justified. Secondly, compared to *BufferQueryTJ*, the performance of *BufferQueryPrune* improves significantly with the increase in buffer distance. Thirdly, the 0- and 1-object filtering techniques have incremental contribution to the buffer query evaluation. Fourthly, relative to *BufferQueryTJ*, *BufferQueryPrune* has a slower rate of increase in execution time as distance increases. This is primarily due to the decrease in both execution time and the data need to be read from the disk. Lastly, the majority of buffer query evaluation time is on filtering and evaluation. Only small fraction of the time is on *MBR*-join. For the real-life data sets, the total execution time for a buffer query with algorithm *bufferQueryPrune* ranges from about 150 seconds to 3500 seconds. Recall that in Section 4.3, the time for computing the candidate set in a query evaluation in a test ranges from 18 to 88 seconds.

In a buffer query evaluation, an object may be read or swapped-in more than once. Let us call this *duplicate swap-in*. Next let us look at how the filtering techniques affect duplicate swap-in as cache size changes. First observe that by setting the cache size to 100%, there is no duplicate swap-

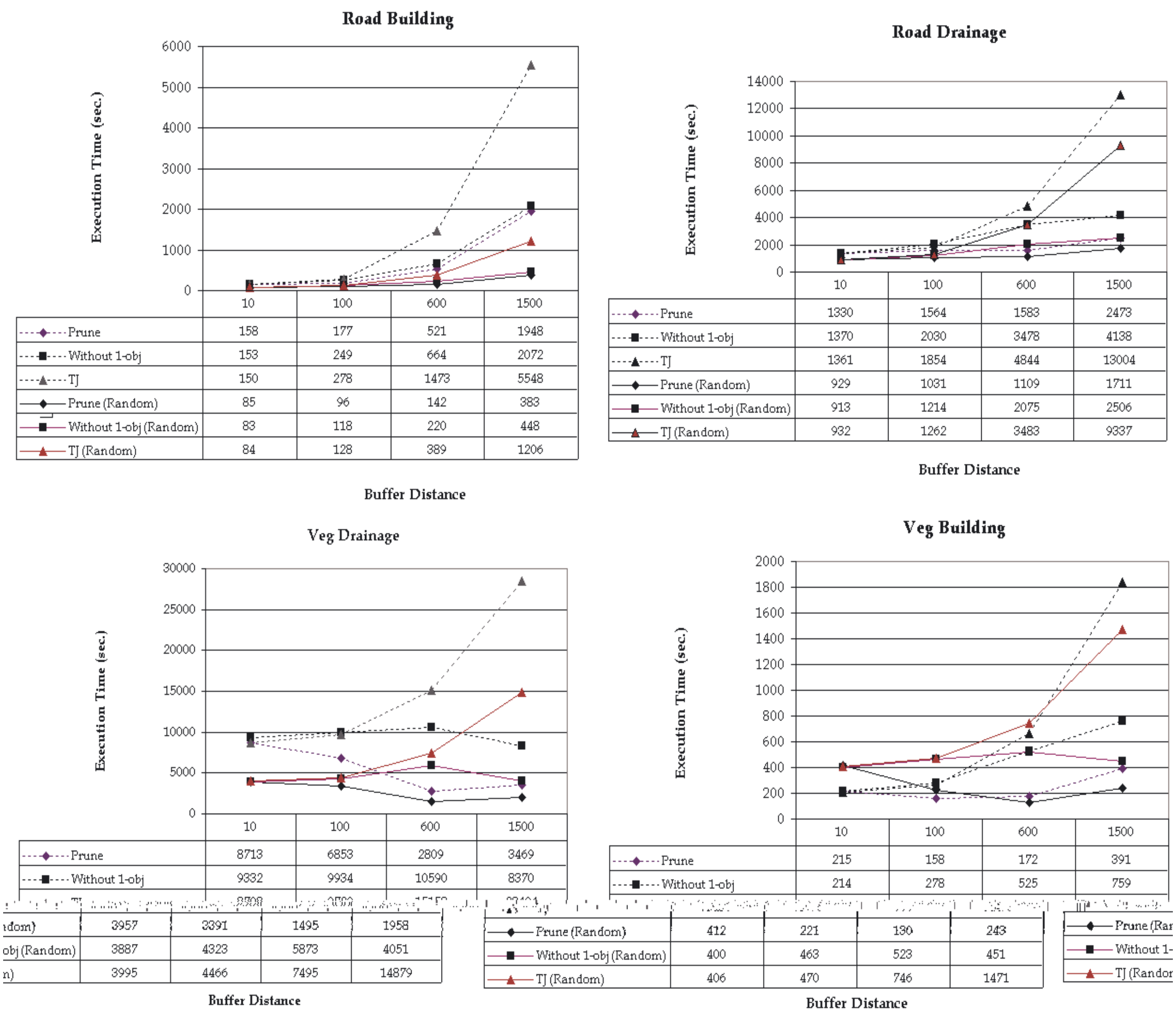


Figure 11: Execution Time

in during the evaluation. Let us define *duplicate swap-in ratio* as the ratio of number of objects swapped-in during a query evaluation to that when no duplicate swap-in; that is, when the cache size is set to 100%. Informally, when the duplicate swap-in ratio is x, it means that, on average, every object involves in the query evaluation is swapped-in x times. Clearly for cache size of 100%, the duplicate swap-in ratio is 1. The smaller the ratio, the better the performance. Tables 2 and 3 summarize important data on logical data accesses. In the columns of 1% and 20%, the values are the duplicate swap-in *ratios* when the cache size is set to the corresponding fraction. For the columns of 100%, they record the *number* of objects swapped-in during the evaluation process.

centrate on Table 2. From Table 2, duplicate swap-in increases as buffer distance increases. This is expected as distance d increases, more and more objects from the same data set are within distance d of an object in the other data set. This results in duplicate swap-in in the evaluation process. Buffer query evaluation is costly, especially when the distance is large. Likewise, the duplicate swap-in decreases with larger cache sizes. From Table 2, *BufferQueryPrune* has the lowest duplicate swap-in across all data sets, cache sizes and buffer distances. Relative to *BufferQueryTJ*, the rate of increase in duplicate swap-in is relatively flat for *BufferQueryPrune*. This shows the effectiveness of the filtering techniques. From columns 100%, it also requires the fewest number of objects in query evaluation. This implies that *BufferQueryPrune* requires the least number of data accesses. Relative to *BufferQueryTJ*, the improvement becomes significant when the distance is greater than 100.

	Prune			Without 1-Object			TJ		
	1%	20%	100%	1%	20%	100%	1%	20%	100%
Road Building 10	1.91	1.26	10554	1.91	1.26	10565	1.91	1.26	10565
Road Drainage 10	2.03	1.21	13641	2.04	1.21	14154	2.04	1.21	14154
Veg Building 10	1.49	1.00	2270	1.49	1.00	2273	1.49	1.00	2273
Veg Drainage 10	2.19	1.31	12526	2.22	1.32	12852	2.22	1.32	12852
Road Building 100	1.79	1.58	12680	2.18	1.89	14508	2.74	2.35	15646
Road Drainage 100	1.60	1.39	14316	1.66	1.46	17962	1.76	1.55	18134
Veg Building 100	1.17	1.01	3336	1.31	1.13	4431	1.36	1.18	4731
Veg Drainage 100	1.63	1.45	10197	1.78	1.61	15431	1.84	1.66	15682
Road Building 600	3.46	2.72	16332	4.35	3.39	18753	20.17	15.09	22084
Road Drainage 600	2.48	2.02	19097	2.92	2.38	25311	7.30	5.58	27629
Veg Building 600	1.52	1.38	6838	2.07	1.87	9536	4.32	3.83	12436
Veg Drainage 600	2.05	1.78	13064	2.75	2.37	18326	5.82	4.36	19858
Road Building 1500	5.81	4.13	19245	7.44	5.54	20703	92.01	67.01	22440
Road Drainage 1500	3.75	3.04	24255	4.69	3.73	28050	32.42	24.18	29161
Veg Building 1500	2.17	1.94	9872	3.22	2.82	11958	18.27	15.84	13436
Veg Drainage 1500	2.95	2.48	16566	4.26	3.53	19492	21.08	16.86	20175

Table 2: Duplicate Swap-in Summary for Real-Life Data Sets

In sum, the performance of *bufferQueryPrune* is superior when compared to *bufferQueryTJ*. Our experiments show that it is preferred independent of the data sets, buffer distances and cache sizes. The improvement in performance by *bufferQueryPrune* is significant, especially with large buffer distances. The filtering strategies employed determine if a candidate pair is in the answer or not with a minimum cost. It invokes the expensive operation *minDist* only if it is absolutely necessary. As a result, it minimizes both the CPU as well as IO. In the next section, we will analyse

	Prune			Without 1-Object			TJ		
	1%	20%	100%	1%	20%	100%	1%	20%	100%
Road Building 10	1.60	1.06	5189	1.60	1.06	5201	1.60	1.06	5201
Road Drainage 10	1.63	1.13	12822	1.62	1.13	12886	1.62	1.13	12887
Veg Building 10	1.42	1.02	3722	1.42	1.02	3729	1.42	1.02	3729
Veg Drainage 10	1.58	1.05	7740	1.58	1.05	7849	1.58	1.05	7849
Road Building 100	1.20	1.07	6883	1.26	1.13	7935	1.27	1.16	9781
Road Drainage 100	1.44	1.23	15047	1.44	1.24	18026	1.45	1.26	19678
Veg Building 100	1.09	1.02	3791	1.16	1.07	5614	1.17	1.08	5924
Veg Drainage 100	1.25	1.10	8098	1.28	1.13	10804	1.16	1.13	11108
Road Building 600	1.38	1.23	11222	1.76	1.53	13580	3.63	4.40	22240
Road Drainage 600	1.92	1.57	20752	2.25	1.81	25551	4.95	3.61	29158
Veg Building 600	1.27	1.13	6253	1.56	1.38	9564	2.56	2.17	13021
Veg Drainage 600	1.59	1.30	11126	1.85	1.52	16148	3.03	2.36	19678
Road Building 1500	1.87	1.60	15614	2.45	2.05	17401	15.63	12.29	22440
Road Drainage 1500	2.85	2.21	25483	3.53	2.69	28179	21.30	14.52	29176
Veg Building 1500	1.67	1.39	8885	2.22	1.86	11628	9.66	7.74	13439
Veg Drainage 1500	2.23	1.73	14840	2.77	2.14	18604	11.33	8.01	20175

Table 3: Duplicate Swap-in Summary for Synthetic Data Sets

the contribution of 0- and 1-object filtering techniques.

7.2.3 The Contribution of 0- and 1-object Filterings

In this section, we shall concentrate on *bufferQueryPrune*. Let us first look at, for each filtering technique, how it contributes to the answer set. Figures 12 and 13 show the fraction of the candidate set that a technique contributes to the answer for various combinations of data sets with different distance values. As the two figures have very similar pattern, let us concentrate on Figure 12. When the buffer distance is very small (i.e., distance = 10), 0-object filtering is ineffective while 1-object filtering has some contribution. From Figure 11, the difference in execution time, however, is negligible. This is due to the low cost of the 0-object filtering and the small number of candidates. However, as buffer distance increases, the effectiveness of 0- and 1-object filterings become more and more predominant, and thus as the percentage of the size of candidate size, fewer need to be evaluated with $minDist$. This also contributes to reduction in the number of duplicate swap-in which in term implies fewer IO operations. For instance, for large buffer distance (i.e., distance = 1500), less than 10% of candidates need to be evaluated with $minDist$ functions. This demonstrates the effectiveness of the filtering techniques proposed as distance increases. For relatively large distances (i.e., distance = 600, 1500), the combined techniques work equally well

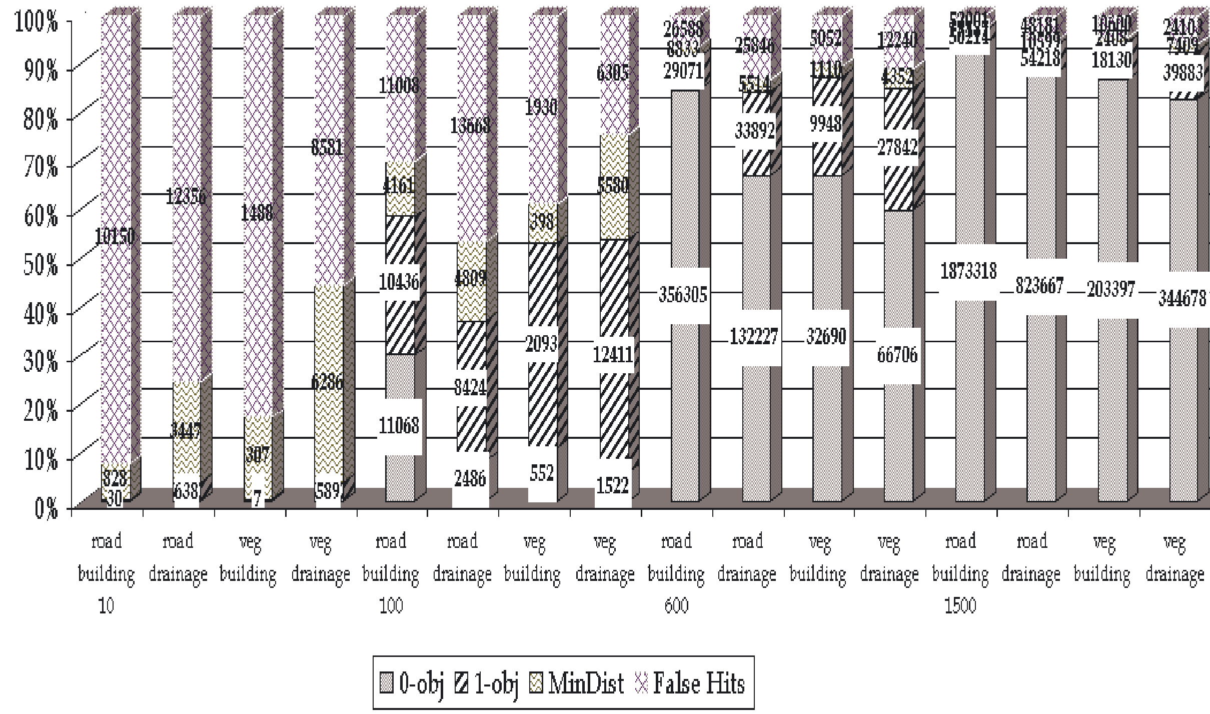


Figure 12: Contributing Ratios of Various Techniques for Real-Life Data Sets

for all data sets tested. This can be explained by the fact that distance value is much larger than the average size of the object's *mbrs*. For smaller distance values (i.e., distance =100), the 0-object filtering performs better for road-building since the average object's *mbrs* are smaller while 1-object filtering work better for veg-building and veg-drainage combinations because of the much larger size of vegetation objects.

8 Conclusion

We investigated the problem of how to evaluate buffer queries efficiently. A buffer query involves two data sets and a buffer distance. A fundamental problem in buffer query evaluation is to determine if two geometric objects are within a given distance d of each other. We derived an efficient algorithm *MinDist* for solving this problem. We showed that, with real-life data, the proposed *MinDist* algorithm outperforms the brute-force approach by a wide margin. The performance of this algorithm is particular impressive for large region data sets.

Together with a *MinDist* algorithm, existing spatial query evaluation algorithms can be modified easily to evaluate a buffer query. However, the cost of evaluating a buffer query increases drastically when the distance is relatively large. We observed that many or even most candidates produced in *MBR*-join need not be evaluated with the relatively expensive *MinDist*. We proposed an algorithm

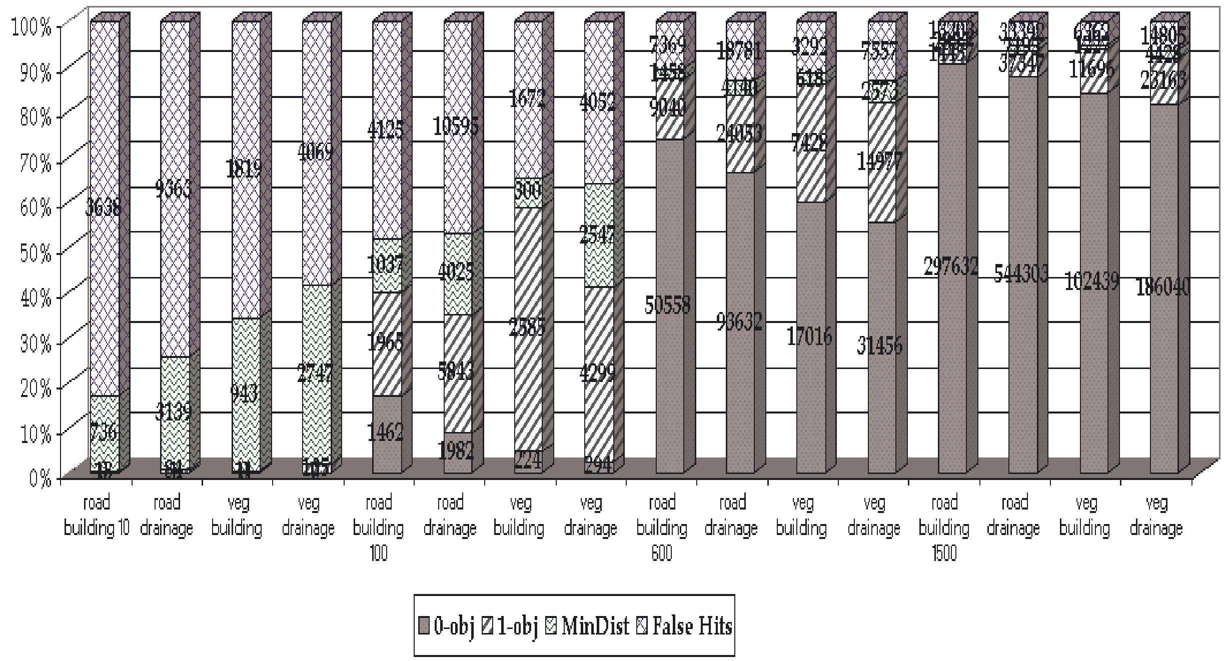


Figure 13: Contributing Ratios of Various Techniques for Synthetic Data Sets

that employs 0- and 1-object filterings to reduce the computation as well as IO accesses. In this algorithm, a candidate is first evaluated with the least expensive technique - 0-object filtering. If it fails, a more costly operation, 1-object filtering, is applied. Finally, if both filterings fail, the most expensive *MinDist* is invoked. We showed with real-life as well as synthetic data sets that the proposed algorithm is very effective: both the execution time and IO accesses are reduced significantly as buffer distance increases. It works well across all cache sizes, buffer distances and data sets.

Duplicate swap-in is unavoidable if buffer distances are not restricted to a small range. An issue for future investigation is to develop techniques that reduce duplicate swap-in further in a buffer query evaluation.

Acknowledgement

The author wishes to thank Faculty of Environmental Studies at the University of Waterloo for providing the test data sets. Financial assistance from the Natural Sciences and Engineering Research Council of Canada and from Bell University Laboratories is gratefully acknowledged.

References

[1] Arge, L., Procopiuc, S., Ramaswamy, S., Suel, T. and Vitter, J. "Scalable Sweeping-based Spatial Join," *Proceedings of the 24th International Symposium on Very Large Databases*, pp. 570-581, Morgan Kaufmann, 1998.

[2] Becker, L., Giesen, A., Hinrichs, K. and Vahrenhold, J. "Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets," *Proceedings of the Sixth International Symposium on Advances in Spatial Databases*, pp. 270-285, Hong Kong, China, 1999.

[3] Brinkhoff, T., Kriegel, H-P and Seeger B., "Efficient Processing of Spatial Joins Using R-Trees," *Proceedings of ACM SIGMOD*, Washington, D.C., 1993, pp.237-246.

[4] Brinkhoff, T., Kriegel, H-P, Schneider, R. and Seeger B., "Multi-Step Processing of Spatial Joins," *Proceedings of ACM SIGMOD*, Washington, D.C., 1994, pp.197-208.

[5] Chan, E.P.F. and Ng. J.N.H., "A General and Efficient Implementation of Geometric Operators and Predicates," *Proceedings of the Fifth International Symposium on Advances in Spatial Databases*, pp. 69-93, Berlin, Germany, 1997.

[6] Chin, F. and Wang. C.A., "Optimal Algorithms for the Intersection and the Minimum Distance Problems Between Planar Polygons," *IEEE Transactions on Computer*, pp. 1203-1207, 1983.

[7] Corral, A., Manolopoulos, Y., Theodoridis, Y. and Vassilakopoulos, M. "Closest Pair Queries in Spatial Databases," *Proceedings of ACM SIGMOD*, Dollars, TX, 2000, pp.189-200.

[8] Kamel, I. and Faloutsos, C., "Hilbert R-Tree: An Improved R-Tree Using Fractals," *Proceedings of 20th VLDB*, 1994, pp. 500-509.

[9] Kriegel, H., Brinkhoff, T. and Schneider, R., "An Efficient Map Overlay Algorithm Based on Spatial Access Methods and Computational Geometry" *Proceedings of the International Workshop on DBMS's for Geographic Applications*, Capri, May 12-17, 1991, pp. 194-211.

[10] Guting, R.H. and Schilling, W., "A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem," *Information Sciences 42*, 1987, pp.95-112.

[11] Guting, R.H., Ridder, T. and Schneider, M., "Implementation of ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types," *Proceedings of Advances in Spatial Databases - Fourth International Symposium, SSD '95*, pp. 216-239, Portland, Maine, August 1995. Lecture Notes in Computer Science 951. Springer-Verlag.

[12] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1984, pp. 47-57.

[13] Huang, Y-W, Jones, M. and Rundensteiner, E., "Improving Spatial Intersect Using Symbolic Intersect Detection," *Proceedings of the Fifth International Symposium on Advances in Spatial Databases*, pp. 165-177, Berlin, Germa[illegible]

[18] Roussopoulos, N., Kelley, S. and Vincent, F., "Nearest Neighbor Queries," *Proceedings of ACM SIGMOD*, San Jose, CA, 1995, pp.71-79.

[19] Zimbrao, G and de Souza, J.M., "A Raster Approximation for the Processing of Spatial Joins," *Proceedings of 24th VLDB*, 1998, pp. 558-569.