Eurographics Symposium on Point-Based Graphics (2006)
M. Botsch, B. Chen (Editors)

A Fast k-Neighborhood Algorithm for Large Point-Clouds[†]

Jagan Sankaranarayanan, Hanan Samet and Amitabh Varshney

Department of Computer Science, Center for Automation Research,Institute for Advanced Computer Studies
University of Maryland, College Park, MD - 20742
jagan,hjs,varshney@cs.umd.edu

Abstract

Algorithms that use point-cloud models make heavy use of the neighborhoods of the points. These neighborhoods are used to compute the surface normals for each point, mollification, and noise removal. All of these primitive operations require the seemingly repetitive process of finding the k nearest neighbors of each point. These algorithms are primarily designed to run in main memory. However, rapid advances in scanning technologies have made available point-cloud models that are too large to fit in the main memory of a computer. This calls for more efficient methods of computing the k nearest neighbors of a large collection of points many of which are already in close proximity. A fast k nearest neighbor algorithm is presented that makes use of the locality of successive points whose k nearest neighbors are sought to significantly reduce the time needed to compute the neighborhood needed for the primitive operation as well as enable it to operate in an environment where the data is on disk. Results of experiments demonstrate an order *of magnitude improvement in the* time *to perform the algorithm and* several orders *of magnitude improvement in* work efficiency *when compared with several prominent existing method.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques; I.3.8 [Computer Graphics]: Applications;

1. Introduction

In recent years there has been a marked shift from using triangles to using points as object modeling primitives in computer graphics applications (e.g., [AGPS04, HDD*92, JDD03, LPC*00, PKKG03]). A point-model (often referred to as a *point-cloud*) usually contains millions of points. Improved scanning technologies [LPC*00] have resulted in even larger objects being scanned into point-clouds. Note that a point-cloud is nothing more than a collection of scanned points and may not even contain any topological information. However, most of the topological information can be deduced by applying suitable algorithms on the point-clouds. Some of the fundamental operations performed on a freshly scanned point-cloud include the computation of *surface normals* in order to be able to illuminate the scanned object, applications of *noise-filters* to remove any residual noise from the scanning process, and tools that change the *sampling rate* of the point-model to the desired level. What is common to all three of these operations is that they work by computing the k nearest neighbors for each point in the point-cloud. There are two important distinctions from other applications where the computation of neighbors is required. First of all, neighbors need to be computed for all points in the dataset, although there is a potential scope to optimize this task. Second, no assumption can be made about the size of the dataset. In this paper, we discuss the *k-nearest-neighbor*(*kNN*) algorithm, also known as the all-points k-nearest-neighbor algorithm, which takes a point-cloud dataset R as an input and computes the k nearest neighbors for each point in R.

We start by comparing and contrasting our work from the related work of Clarkson [Cla83] and Vaidya [Vai89]. Clarkson proposes an $O(n \log \delta)$ algorithm for computing the nearest neighbor to each of n points in a dataset S, where δ is the ratio of the diameter of S and the distance between the closest pair of points in S. Clarkson uses a PR-quadtree [Sam06]

[†] The support of the National Science Foundation under Grants CCF-0515241 and EIA-00-91474, Microsoft Research, and the University of Maryland General Research Board is gratefully acknowledged.

Q on the points in S. The running time of his algorithm depends on the depth $d = \delta$ of Q. This dependence on the depth is removed by Vaidya who proposes using a hierarchy of boxes, termed Box tree, to compute the k nearest neighbors to each of the n points in S in $O(kn \log n)$ time. There are two key differences between our algorithm and those of Clarkson and Vaidya. First of all, our algorithm can work on most disk-based (out of core) data structures whether they are based on a regular decomposition of the underlying space such as a PMR quadtree [Sam06] or on object hierarchies such as an R-tree [Gut84]. In contrast to our algorithm, the methods of Clarkson and Vaidya have only been applied to memory-based (i.e., incore) data structures such as the PR quadtree and Box tree, respectively. Consequently, their approaches are limited by the amount of physical memory present in the computer on which they are executed. The second difference is that our algorithm can be easily modified to produce nearest neighbors incrementally, *i.e.*, we are able to provide a variable number of nearest neighbors to each point in S depending on a condition, which is specified at run-time. The incremental behavior has important applications in computer graphics. For example, the number of neighbors used in computing the normal to a point in a point-cloud can be dependent on the curvature of a point.

The development of efficient algorithms for finding the nearest neighbors for a single point or a small collection of points has been an active area of research [HS95, RKV95]. The most prominent neighbor finding algorithms use Depth-First Search (DFS) [RKV95] or Best-First Search (BFS) [HS95] methods to compute neighbors. Both algorithms make use of a search hierarchy which is a spatial data-structure such as an R-tree [Gut84] or a variant of a quadtree or octree (e.g., [Sam06]). The DFS algorithm, also known as branch-and-bound, traverses the elements of the search hierarchy in a predefined order and keeps track of the closest objects so far from the query point. On the other hand, the BFS algorithm traverses the elements of the search hierarchy in an order defined by their distance from the query point. The BFS algorithm that we use, stores both points and *blocks* in a priority queue. It retrieves points in an increasing order of their distance from the query point. This algorithm is *incremental* as the number of nearest neighbors k need not be known in advance. Successive neighbors are obtained as points are removed from the priority queue. A brute force method to perform the *kNN* algorithm would be to compute the distance between every pair of points in the dataset and then to choose the top k results for each point. Alternatively, we also observe that repeated application of a neighbor finding technique [MA97] on each point in the dataset also amounts to performing a *kNN* algorithm. However, like the brute-force method, such an algorithm performs wasteful repeated work as points in proximity share neighbors and ideally it is desirable to avoid recomputing these neighbors.

Some of the work in computing the k nearest neighbors can be reduced by making use of the approximate nearest neighbors [MA97]. In this case, the approximation is achieved by making use of an error-bound ε which restricts the ratio of the distance from the query point q to an approximate neighbor and the distance to the actual neighbor to be within $1+\varepsilon$. When used in the context of a point-cloud algorithm, this method may lead to inaccuracies in the final result. In particular, point-cloud algorithms that determine local surface properties by analyzing the points in the neighborhood may be sensitive to such inaccuracies. For example, such problems can arise in algorithms for computing normals, estimating local curvature, as well as sampling rate and local point-cloud operators such as *noise-filtering* [JDD03, FDCO03], *mollification* and removal of *outliers* [WPH*04]. In general, the correct computation of neighbors is important in two main classes of point-cloud algorithms: algorithms that identify or compute properties that are common to all of the points in the neighborhood, and algorithms that study variations of some of these properties.

An important consideration when dealing with point-models that is often ignored is the size of the point-cloud datasets. The models are scanned at a *high fidelity* in order to create an illusion of a smooth surface. The resultant point-models can be on the order of several millions of points in size. Existing algorithms such as normal computation [MN03] which make use of the suite of algorithms and data structures in the Approximate Nearest Neighbor (ANN) library [MA97] are limited by the amount of physical memory present in the computer on which they are executed. This is because the ANN library makes use of in-core data structures such as the k-d tree [Ben75] and the BBD-tree [AMN*94]. As larger objects are being converted to point-models, there is a need to examine neighborhood finding techniques that work with data that is out of core and and thus out-of-core data structures should be used. Of course, although the drawback of out-of-core methods is the incurrence of I/O costs thereby reducing their attractiveness for real-time processing, the fact that most of the techniques that involve point clouds are performed *offline* mitigates this drawback.

There has been a considerable amount of work on efficient disk-based nearest neighbor finding methods [HS95, RKV95, XLOH04]. Recently, there has also been some work on the *kNN* algorithm [BK04, XLOH04]. In particular, the algorithm by Böhm [BK04], termed *MuX* uses the DFS algorithm to compute the neighborhoods of one block, say b, at a time (i.e., it computes the k nearest neighbors of all points in b before proceeding to compute the k nearest neighbors in other blocks) by maintaining and updating a best set of neighbors for each point in the block as the algorithm progresses. The rationale is that this will minimize disk I/O as the k nearest neighbors of points in the same block are likely to be in the same set of blocks. The GORDER method [XLOH04] takes a slightly different approach in that although it was originally designed for high-dimensional data-points (e.g. similarity retrieval in image processing applications), it can also be applied to low-dimensional datasets. In particular, this algorithm first performs a Principal Component Analysis (PCA) to determine the first few dominant directions in the data space and then all of the objects are projected to this reduced space thereby resulting in drastic reduction in the dimensionality of the point dataset. The resulting blocks are organized using a regular grid, and at this point, a *kNN* algorithm is performed which is really a sequential search of the blocks

Even though both Xia [XLOH04] and Böhm [BK04] methods compute the neighborhood of all points in a block before proceeding to process points in another block, each point in the block keeps track of its k-nearest neighbors encountered thus far. Thus this work is performed independently and in isolation by each point with no reuse of neighbors of one point as neighbors of a point in spatial proximity. Instead, we identify a region in *space* that contains all of the k nearest neighbors of a *collection* of points (the space is termed *locality*). Once the best possible *locality* is built, each point searches only the locality for the correct set of k nearest neighbors. This results in large savings. Also, our method makes no assumption about the size of the dataset or the sampling-rate of the data. Experiments (section 6) show that our algorithm is faster than the GORDER approach and performs substantially fewer distance computations.

The rest of the paper is organized as follows. Section 2 defines the concepts that we use and provides a high level description of our algorithm. Section 3 describes the *locality* building process for *blocks*. Section 4 describes an incremental variant of our *kNN* algorithm, while Section 5 describes a non-incremental variant of our *kNN* algorithm. Section 6 discusses results of experiments, while Section 7 discusses related applications that can benefit from using our algorithm. Finally, concluding remarks are drawn in Section 8.

2. Preliminaries

In this paper we assume that the data consists of points in a multi-dimensional space and that they are represented by a hierarchical spatial data structure. Our algorithm makes use of a disk-based quadtree variant that recursively decomposes the underlying space into blocks until the number of points in a block is less than some *threshold limit (B)* [Sam06]. In fact, any other hierarchical spatial data structure could be used including some that are based on object hierarchies such as the R-tree [Gut84]. The blocks are represented as nodes in a tree access structure which enables point query searching in time proportional to the logarithm of the width of the underlying space. The tree contains two types of nodes: leaf and non-leaf. Each non-leaf node has at most 2^d nonempty *children*, where d corresponds to the *dimension* of the underlying space. A *child* node occupies a region in space that is fully contained in its parent node. Each leaf node contains a pointer to a *bucket* that stores at most B points. The *root* of the tree is a special block that corresponds to the entire underlying space which contains the dataset. While the blocks of the access structure are stored in main-memory, the buckets that contain the points are stored on disk. In our implementation, a count is maintained of the number of points that are contained within the *subtree* of which the corresponding block b is the root and a *minimum bounding box* of the space occupied by the points that b contains.

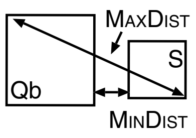


Figure 1: *Example illustrating the values of the* MinDist *and* MaxDist *distance estimates for blocks* Qb *and S.*

We use the Euclidean metric (L_2) for computing distances. Our *kNN* algorithm can be easily modified to accommodate other distance metrics. Our implementation makes extensive use of the two distance estimates MinDist and MaxDist (Figure 1). MinDist (q,e) between two blocks Qb and S refers to the minimum possible distance between any point q in Qb and e in S. When a list of blocks is ordered by their MinDist value with respect to a reference block or a point, the ordering is called a MinDist ordering. MaxDist (q,e) refers to the maximum possible distance between any possible q and e in Qb and S respectively. An ordering based on MaxDist is called a MaxDist ordering. The *kNN* algorithm identifies the k nearest neighbors for each point in the dataset. We refer to the set of k nearest neighbors of a point o as the *neighborhood* of o. While the neighborhood is used in the context of points, *locality* defines a neighborhood of blocks. Intuitively, the locality of a block b is the region in space that contains all the k nearest neighbors of all points in b. We make one other distinction between the concepts of neighborhood and locality. While neighborhoods contain no other point other than the k nearest neighbors, locality is more of an approximation and thus the locality of a block b may contain points that do not belong to the neighborhood of any of the points contained within b.

Our algorithm has the following high-level structure. It first builds the locality for a block and later searches the locality to build neighborhood for each point contained within the block. The pseudo-code presented in Algorithm 1 explains the high level workings of the *kNN* algorithm. Lines 1-2 traverse each block in the dataset R and build an approximate *locality*. Lines 3-4 build a neighborhood using the approximate locality for all points in b.

Algorithm 1
Procedure *kNN*(R,k)
```
(∗ high-level description of kNN algorithm ∗)
1.   for each block $b$ in $R$
2.       do Build locality $l$ for $b$ in $R$
```

```
3.       for each point p in b
4.           do Build neighborhood of p using l and k
5.   return
```

3. Building the Locality of a Block

As the locality defines a region in space, we need a measure that defines the extent of the locality. Given a *query block*, such a measure would implicitly determine if a point or a block belongs to the locality. We specify the extent of a locality by a distance-based measure that we call PRUNEDIST. All points and blocks whose distance from the query block is less than PRUNEDIST belong to the locality. The challenge in building localities is to find a good estimate for PRUNEDIST. Finding the smallest possible value of PRUNEDIST requires that we examine every point which defeats the purpose of our algorithm which is why we resort to estimating it.

We proceed as follows. Assume that the query block(*Qb*) is in the vicinity of other blocks of various sizes. We want to find a set of blocks so that the total number of points that they contain is at least k, while keeping PRUNEDIST as small as possible. We do this by processing the blocks in increasing order of their MAXDIST order from *Qb* and adding them to the locality. In particular, we sum the counts of the number of points in the blocks until the total number of points in the blocks that have been encountered exceeds k and record the current value M of MAXDIST. At this point, we process the remaining blocks according to their MINDIST order from *Qb* and add them to the locality until encountering a block b whose MINDIST value exceeds M. All remaining blocks need not be examined further and are inserted into list PRUNEDLIST. Note that an alternative approach would be to initially process the blocks in MINDIST order, adding them to the locality, and use the MAXDIST value M when the sum of the counts is greater than k to prune every block whose MINDIST is greater than M. This approach does not yield as tight an estimate for PRUNEDIST as can be seen in the example in Figure 2.

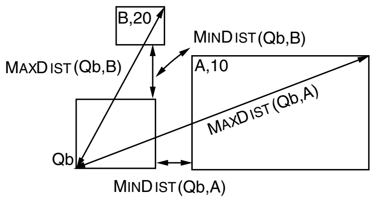


Figure 2: *Query block Qb in the vicinity of two other blocks A and B containing* 10 *and* 20 *points respectively. When* k *is 15, choosing A with a smaller* MINDIST *value does not provide the lowest possible* PRUNEDIST *bound.*

The pseudo-code for the locality building process is given below in Algorithm 2. The initial inputs to the BUILDLOCALITY procedure are the query block(*Qb*), the initial locality *locality*, and k. Using these inputs, the algorithm computes a new locality for *Qb*. Lines 4-8 select blocks in the MAXDIST ordering from *Qb*. The loop terminates when k or more points have been found which is kept track of by detecting when the *allowance* variable is less than or equal to 0 having been initialized to k at the start of the algorithm. Lines 10-14 further add blocks to the locality whose MINDIST to *Qb* is closer than PRUNEDIST. Line 15 returns the new locality and the pruned list of blocks.

The mechanics of the algorithm are illustrated in Figure 3. The figure shows *Qb* in the vicinity of several other blocks. Each block has a label and the number of points it contains. For example, suppose that the *allowance* is initialized to 3. The algorithm selects blocks by the MAXDIST ordering from *Qb* until 3 points are found. Hence, X and Y are selected and PRUNEDIST is now known. The next step is to choose all blocks whose MINDIST from *Qb* is less than PRUNEDIST and thus $B, E, F, I, D, P, R, V, M,$ *and* O are chosen.

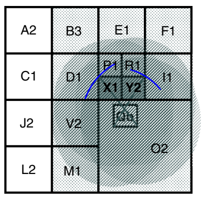


Figure 3: *Illustration of the workings of the* BUILDLOCALITY *algorithm. The labeling scheme assigns each block a label concatenated with the number of points that it contains.* Qb *is the query block. Blocks X and Y are selected based on the value of* MAXDIST, *while blocks B,E,F,I,D,P,R,V,M, and O are also selected as their* MINDIST *value from* Qb $<$ PRUNEDIST.

```
Algorithm 2
Procedure BUILDLOCALITY[Qb,locality,k ]
Input: Qb is the Query point.
Input: locality is a list of blocks; denotes initial locality
Output: newlocality is the pruned locality of Qb
(* Count is the number of points contained in a block *)
1.  (* maxorder stores the locality in increasing order of
    MAXDIST with respect to Qb *)
2.  (* PRUNEDLIST is a list of pruned blocks *)
3.  allowance ← k
4.  while (allowance > 0)
5.      do block_i ← Next(maxorder)
6.      PRUNEDIST ← MAXDIST (Qb,block_i)
7.      allowance ← allowance - Count(block_i)
8.      newlocality.Insert(block_i)
9.  (* minorder is a list; denoting the remaining blocks in
    maxorder in a MINDIST ordering from Qb *)
10. while (Exists(minorder))
11.     do block_i = Next(minorder)
12.     if (MINDIST (Qb,block_i) < PRUNEDIST)
```

```
13.         then newlocality.Insert($block_i$)
14.         else PRUNEDLIST.Insert($block_i$)
15.   return newlocality,PRUNEDLIST
```

4. Incremental *kNN* Algorithm

We briefly describe the working of an incremental variant of our *kNN* algorithm. This algorithm is useful when variable number of neighbors are required for each point in the dataset. For example, when dealing with certain point-cloud operations, where the number of neighbors required for a point p is a function of its local characteristics (*e.g.*, curvature), the value of k cannot be pre-determined for all the points in the dataset, *i.e.*, few of the points may require more than k neighbors. The incremental *kNN* algorithm as seen in Algorithm 3, can produce as many neighbors as required by the point-cloud operation. This is contrast to the ANN algorithm [MA97], where retrieving the $k+1^{th}$ neighbor of p, would entail recomputing all of the first $k+1$ neighbors to p.

If Qb in Algorithm 2 is a leaf-block, then for each point $p \in Qb$, Algorithm INCKNN(p, *locality* of Qb) is invoked. A priority queue Q in line 3 that retrieves elements by their MINDIST from p, is initialized with the locality of Qb. If an element E retrieved from the top of Q is a BLOCK, it is replaced with its children blocks (line 6–7). Note that the *locality* of Qb is only guaranteed to contain the first k nearest neighbors of any $p \in Qb$, after which the PRUNEDLIST (subsequently, an ancestor) of Qb is added to Q, as shown in lines 9–14. If E is a point, it is reported (line 15) and the control of the program returns back to the user. Additional neighbors of p are retrieved by making subsequent invocations to the algorithm.

Algorithm 3
Procedure INCKNN[p,*locality*]
Input: p is the Query point
Input: *locality* is a list of blocks

```
1.    PB ←PARENT($p$)
2.    PDIST ←PRUNEDIST(PB)
3.    $Q$ ←Priority Queue initialized with locality
4.    while ($Q$ not EMPTY)
5.        do $E$ ←$Q.pop()$
6.        if ($E$ is BLOCK)
7.          then insert children of $E$ in $Q$
8.          else (∗ $E$ is a POINT ∗)
9.               if (DIST($E$,PB) ≥ PDIST)
10.                then insert PRUNEDLIST of PB in $Q$
11.                     if PB is ROOT
12.                       then PDIST ←∞
13.                       else PB ←PARENT(PB)
14.                            PDIST ←PRUNEDIST(PB)
15.              report $E$ as next neighbor (and return)
```

5. Non-Incremental *kNN* Algorithm

In this section, we describe our *kNN* algorithm that computes k neighbors to each point in the dataset. A point a whose k neighbors are being computed, is termed the query point. An ordered set containing the k nearest neighbors to a is collectively termed the *neighborhood* $n(a)$ of a. Although the examples in this section use a two-dimensional setup, the concepts hold true for any arbitrary dimension. Let $n(a) = \{q_1^a, q_2^a, q_3^a ... q_k^a\}$ be the neighborhood of point a, such that q_i^a is the i^{th} nearest neighbor of a, $1 \leq i \leq k$ with q_1^a being the closest. We represent the L_2 distance of a point, $q_i^a \in n(a)$ as $L_2^a(q_i) = \|q_i - a\|$ or d_i^a. Note that, all points in the neighborhood $q_i^a \in n(a)$ are drawn from the locality.

The neighborhood of a succession of query points is obtained as follows. Suppose that the neighborhood of the query point a has been determined by a search process. q_k^a is the farthest point in the neighborhood and all k neighbors are contained within a circle (a hypersphere in higher dimensions) of radius d_k^a centered at a. Let b be the next query point under consideration. As mentioned earlier, the algorithm benefits from choosing b to be close to a. Without loss of generality, assume that b is to the *east* and *north* of a. As both a and b are spatially close to each other, they may share many common neighbors and thus we let b use the neighborhood of a as an initial estimate of b's neighborhood, termed the *approximate neighborhood* of b and denoted by $an(b)$, and then try to improve upon it. At this point, let d_b^k record the distance from b to the farthest point in $an(b)$, the approximate neighborhood of b.

Of course, some of the points in $an(b)$ may not be in the set of k nearest neighbors of b. The fact that we use the L_2 distance metric means that $n(a)$, the neighborhood of a, has a circular shape. Therefore, as shown in Figure 4a, we see that some of the k nearest neighbors of b may lie in the shaded crescent-shaped region formed by taking the set difference of the circle of radius d_b^k centered at b and the circle of radius d_a^k centered at a (i.e., $n(a)$). Thus in order to ensure that we obtain the k nearest neighbors of b, we must also search this crescent-shaped region whose points may displace some of the points in $an(b)$. However, it is not easy to search such a region due to its shape, and thus the *kNN* algorithm would benefit if the shape of the region containing the neighborhood could be altered to enable efficient search while still ensuring that it contain the k nearest neighboring points although it could contain a limited number of additional points.

We achieve such a result by defining a simpler search region which is in the form of a hyper-rectangular bounding box around $n(a)$ which we call the *bounding box* (BBox(a)) of $n(a)$. BBox(a) is a square-shaped region centered at a with width $2 \cdot d_a^k$. Once we have such a search region BBox(a) for a query point a, we obtain a similarly-shaped hyper-rectangular, but not necessarily square, search region BBox$'(b)$ for the next query point b by adding 4 simple hyper-rectangular regions to BBox(a) as shown in Figure 4b. In general for a d dimensional space, 2^d such regions are formed. This process is deceptively simple but may have the unfortunate consequence that its successive application to query points will result in larger and larger bounding boxes. This would defeat the idea of using locality to limit the computational complexity of the *kNN* algorithm. We avoid this repeated growth by following the determination of d_b^k using BBox$'(b)$ with a computation of a smaller BBox(b) with width $2 \cdot d_b^k$.

From the above we see that the computation of BBox(a) of $n(a)$ is an important step in our algorithm and its pseudo-code is given by Algorithm 4. Note that BBox(a) contains all points o_i that satisfy $L_\infty^a(o_i) \le d_a^k$. BBox$(a)$ contains at least k points and all points of $n(a)$. While estimating a bound on number of points in BBox(a) is difficult, at least in two-dimensional space we know that the ratio of the non-overlap space occupied by BBox(a) to $n(a)$ is $\frac{4-\pi}{\pi}$. Consequently, the expected number of points in the non-overlap region is proportionately larger than $n(a)$.

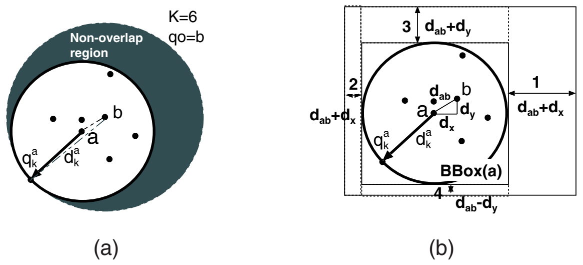


Figure 4: *a) Searching the shaded region for points closer to* b *than q_k^b is sufficient. b) To compute BBox(b) from BBox(a) requires four simple region searches. Compared to the crescent shaped region, these region searches are easy to perform.*

The input to Algorithm 4 is a locality consisting of an initial set of points and a query point a. $n(a)$ is built in lines 3-5, by choosing the first k closest points. This is done by making use of an incremental nearest neighbor finding algorithm such as BFS. Note that at this stage, we could also make use of an approximate version of BFS as pointed out in Section 1. Once the k closest points have been identified, the value of d_k^a is known (line 6). At this point we add the remaining points that are in BBox(a) as they may be needed for the computation of the neighborhood of the next query point b (i.e., $n(b)$). In particular, BBox(a) is constructed by adding points o that satisfy the $L_\infty^a(o) \le d_k^a$ distance criterion (lines 8-10).

Algorithm 4
Procedure BUILDBBOX[*locality*$_a$,a]
Input: *locality*$_a$ list of points in the locality of a
(∗ *locality*$_a$ stores points in order of increasing distance from a ∗)

```
1.  count ← 0
2.  BBox_a ← empty
3.  while (count < k)
4.      do BBox_a.INSERT(NEXTNEARESTNEIGHBOR(locality_a))
5.      count ← count+1
6.  $d_k^a \leftarrow L_2^a(\mathrm{BBox}_a[k])$
7.  (∗ add all points that satisfy the $L_\infty$ criterion ∗)
8.  while ($L_\infty^a(locality_a) \le d_k^a$)
9.      do BBox_a.INSERT(NEXTNEARESTNEIGHBOR(locality_a))
10.     count ← count + 1
11. return
```

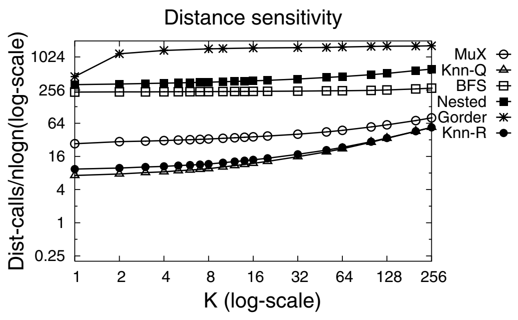
6. Experiments

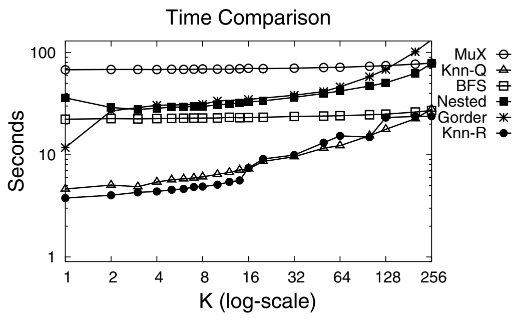
A number of experiments were performed to evaluate the performance of our *kNN* algorithm. The experiments were run on a Quad Intel Xeon server running Linux(2.4.2) operating system with one *Gigabyte* of RAM and SCSI hard disks. The dataset used for evaluation consisted of several commonly used 3D scanned models. The three-dimensional point-cloud models range from 2k to 3200k points. We developed a toolkit in C++ using STL that implements the *kNN* algorithm. The performance of our algorithm was evaluated by varying a number of parameters that are known to influence its performance. We collected statistics about the performance of the algorithm under the experimental environment. For a dataset containing n points, a good benchmark for evaluating a *kNN* algorithm is the *distance sensitivity* [XLOH04] which is defined as follows.

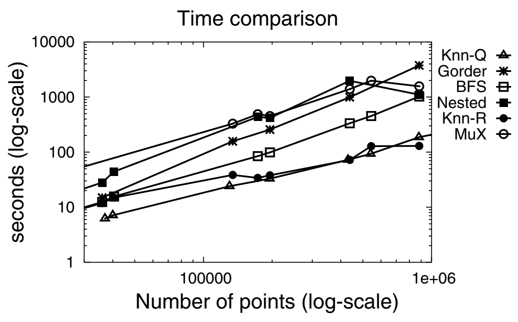
$$\text{distance sensitivity} = \frac{\text{Total distance calculations}}{n \log n}$$

A reasonable algorithm should have a low, and, more importantly, a constant distance sensitivity value.

We evaluated our algorithm by comparing its execution time and its distance sensitivity with that of the GORDER method [XLOH04] and the method of Böhm *et al.* [BK04]. We also included traditional methods like the *nested-join* [UGMW01] and a variant of the *BFS* algorithm that invoked a *BFS* algorithm for each point in the dataset, were used in the comparison. We use both a PMR quadtree and an R-tree variant of the algorithm in the comparative study. Our evaluation was in terms of three-dimensional data as we are primarily interested in databases for computer graphics applications. Our algorithm was tested with a disk page size (B) of 32. Note however, that the values of B and k are chosen independent of each other. We retained 10% of the disk pages in the main memory using a LRU based page replacement policy. For the GORDER algorithm, we used the parameter values that led to its best performance, according to its developers [XLOH04]. In particular, the size of a sub-segment was chosen to be 1000, the number of grids were set to 100, and the size of the data set buffers was chosen to be more than 10% of the data set size. For the *MuX* based method, a page capacity of 100 buckets and a bucket capacity of 1024 objects was adopted. We used a bucket

(a)


(b)


(c)


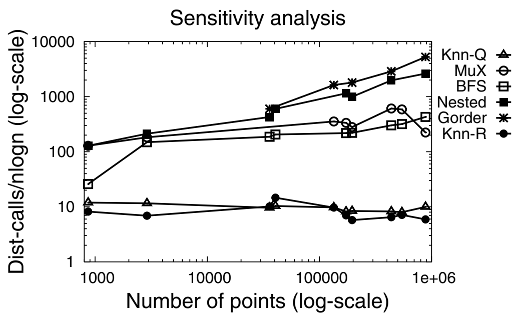
(d)


Figure 5: *The plot shows the performance of our* kNN *algorithm along with the BFS, GORDER, MuX and the Nested-join algorithms. The pseudo name 'kNN-Q' in the plots refer to the quadtree implementation of our algorithm while 'kNN-R' refers to the R-tree implementation of our method. Plots a–b show the performance of the techniques on the* Stanford Bunny *model containing 35947 points for values of* k *ranging between 1 and 256. (a) Records the distance sensitivity, and (b) the time taken to perform the algorithm. Plots c–d record the performance of all the techniques on datasets of various sizes for* $k = 8$. *(c) The time taken to perform the algorithm, and (d) the resultant distance sensitivity.*

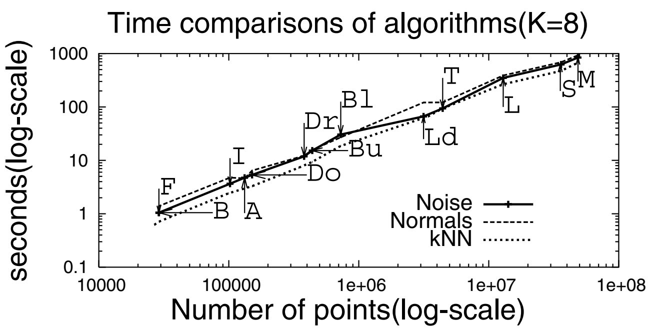
capacity of 1024 for the BFS and nested-join [UGMW01] methods. The results of our experiments were as follows.

1. Our algorithm clearly out-performs all the other methods for all values of k on the Stanford Bunny model as shown in Figure 5a–b. The MuX method performs significantly better than the GORDER method whose performance was comparable to the BFS and the nested-join methods. However, our algorithm leads to at least an order of magnitude improvement in the distance sensitivity over the MuX method for smaller values of k (≤ 32) and at least 50% improvement for larger k (< 256) as seen in Figure 5a. We record atleast 50% improvement in the time to perform the algorithm (Figure 5b).
2. However, as size of the input dataset is increased the performance of the MuX algorithm was comparable to the nested, BFS and the GORDER based methods (Figure 5c). Moreover, our method has an almost constant distance sensitivity even for large datasets. The distance sensitivity of the comparative algorithms are at least an order of magnitude higher for smaller datasets and up to several orders of magnitude higher for the larger datasets in comparison to our method (Figure 5d). We record similar time speedups as seen in Figure 5c.
3. Figure 5c–d show similar performance for the R-tree and the PMR Quadtree variants of our algorithm.

Having established that our algorithm performed better than the GORDER method, we next evaluated the use of our algorithm in a number of applications for different data sets that included several publicly available and a few large synthetically generated point-cloud models. The size of the models ranged from 35k points (*Stanford Bunny* model) to 50 million points (*Syn-50* model). We have developed a few graphical applications that works in conjunction with our *kNN* algorithm. They include computing the surface normals to each point in the point-cloud using a variant of the algorithm by Hoppe *et al.* [HDD*92] and removing noise from the point surface using a variant of the *bilateral filtering* method [JDD03, FDCO03]. Figure 6 shows the time needed to compute *8* neighbors for each point in the point-cloud model. We also provide in Figure 6 the time needed to compute the surface normals and to correct noise when a neighborhood containing 8 points is employed. As we can see, use of our algorithm results in *scalable* performance even as the size of the dataset is increased to a point that it exceeds by several orders of magnitude the amount of available physical memory in the computer. The scalable nature of our approach is also apparent from the almost uniform rate of finding the neighborhoods, i.e., 5900 neighborhoods/second for the Stanford Bunny model and 7779 neighborhoods/second for the Syn-50 point-cloud models.

7. Related Applications

The most obvious application of the *kNN* algorithm is in the construction of *kNN* graphs. *kNN* graphs are use-

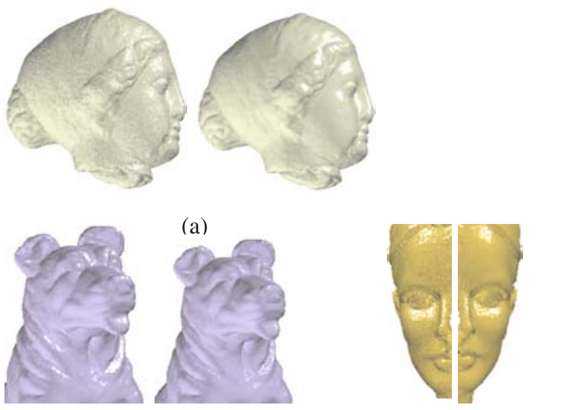


(a)

Model Name	Size (millions)	*kNN*	Surface Normals	Noise Removal
Bunny(Bu)	0.037	6.22	9.0	9.64
Femme(F)	0.04	7.13	10.5	13.9
Igea(I)	0.13	24.05	36.6	47.52
Dog(Do)	0.195	32.9	53.4	64.45
Dragon(Dr)	0.43	72.62	118.9	122.2
Buddha(Bu)	0.54	93.04	152.3	157.25
Blade(Bl)	0.88	185.92	304.2	270.0
Dragon(Ld)	3.9	663.84	900	1209.8
Thai(T)	5	940.04	1240	1215.7
Lucy(L)	14	2657.9	3504	3877.78
Syn-38(S)	37.5	4741.79	-	-
Syn-50(M)	50	6427.5	-	-

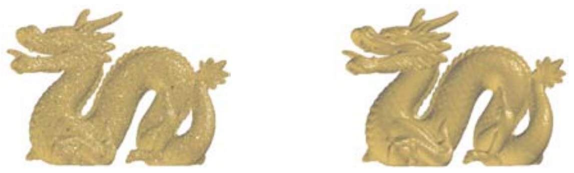
(b)

Figure 6: *(a) Execution time of the* kNN *algorithm for different point models, and (b) the time to execute a number of operations (i.e., normal computation and noise removal) which make use of the* kNN *algorithm. All results are for* k = 8.



(a) (b) (c)

Figure 7: *Three noisy models which were de-noised using filtering and mollification techniques. In the pairs of figures shown for each of the models, the figure on the left is the noisy model, while the figure on the right is the corrected point-model. The (a)* Igea *and (b)* dog *models were denoised with the filtering method, while the (c)* femme *model was denoised using the mollification technique.*



(a) (b)

Figure 8: *(a) A noisy mesh-model of a dragon, and (b) the corresponding model whose surface normals were recomputed using our* kNN *algorithm. The algorithm took about 118 seconds and used 8 neighbors.*

ful when repeated nearest neighbor queries need to be performed on a dataset. Our *kNN* algorithm may also be used in *point reaction-diffusion* [Tur91] algorithms, in order to create most naturally occurring patterns in nature. We have applied our algorithm to the *bilateral mesh filtering* algorithms in [JDD03, FDCO03], the results are shown in Figure 7.

Weyrich *et. al.* [WPH*04] have identified useful point-cloud operations that make use of the moving least squares (MLS) [ABCO*01] technique of Alexa *et al*. Of these operations, we believe that MLS *point-relaxation*, MLS *smoothing*, and MLS based *upscaling* would benefit from using our *kNN* algorithm.

Tools that perform *upscaling* [GH97, PGK02] and down-scaling [ABCO*01] of point-clouds can use our *kNN* algorithm to generate datasets at various *levels of detail (LOD)* [LRC*03]. Using the *quadratic error simplification* techniques in [GH97, PGK02], we we have generated point-models at different levels of details, as shown in Figure 9.

8. Concluding Remarks

We have presented a new *kNN* algorithm that yields an improvement of several orders of magnitude for distance sensitivity and at least one order of magnitude improvement in execution time over an existing method known as GORDER method designed for dealing with large volumes of data that are disk-resident. We have applied our method to point-clouds of varying size including some as large as 50 million points with good performance. A number of applications of the algorithm were presented. Although our focus was on the computation of correct *k* neighbors, our methods can also be applied to work with approximate *k* neighbors

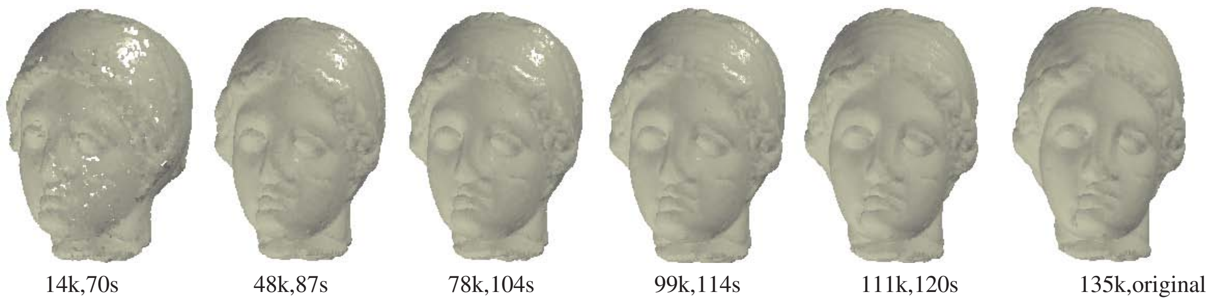


Figure 9: *Sizes and execution times for the result of applying a variant of the The* Igea *point-model of a simplification algorithm [GH97] using the* kNN *algorithm to the* Igea *point-model of size 135k.*

by simply stopping the search for the k nearest neighbors when k neighbors of the query point within ε of the true distance of the k^{th} neighbor have been found. We also plan to explore the applicability of some of the concepts discussed in this paper to high-dimensional datasets using techniques in [DIIM04, GPB05, XLOH04].

Acknowledgments The authors would like to thank the anonymous reviewers for their useful comments and suggestions which helped improve the quality of this paper immensely. Special thanks are due to Chenyi Xia for providing us with the GORDER source code. The meshes used in the paper are the creations of the following people or institutions. The *Bunny, Buddha, Thai, Lucy* and the *Dragon* models belong to the Stanford 3D Scanning Repository. The *Igea* model is from Cyberware Inc. The *Turbine blade* model was obtained from the CD-ROM of the VTK Book. The *Dog* model belongs to MRL-NYU. The *Femme* model is thanks to Jean-Yves Bouguet.

References

[ABCO*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. In *VIS'01:Proceedings of the conference on Visualization* (San Diego, CA, Oct. 2001), IEEE Computer Society, pp. 21–28. 8

[AGPS04] ANDERSSON M., GIESEN J., PAULY M., SPECKMANN B.: Bounds on the k-neighborhood for locally uniformly sampled surfaces. In *Proceedings of the Eurographics Symposium on Point-Based Graphics* (Zurich, Switzerland, June 2004), pp. 167–171. 1

[AMN*94] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A.: An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, VA, Jan. 1994), pp. 573–582. (journal version: [AMN*98]). 2

[AMN*98] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A. Y.: An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM 45*, 6 (Nov. 1998), 891–923. 9

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517. 2

[BK04] BÖHM C., KREBS F.: The *k*-nearest neighbor join: Turbo charging the KDD process. In *Knowledge and Information Systems (KAIS)* (London, UK, Nov. 2004), vol. 6, no. 6, Springer-Verlag. 2, 3, 6

[Cla83] CLARKSON K. L.: Fast algorithm for the all nearest neighbors problem. In *Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science* (Tucson, AZ, Nov. 1983), pp. 226–232. 1

[DIIM04] DATAR M., IMMORLICA N., INDYK P., MIRROKNI V. S.: Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry* (New York, NY, USA, June 2004), ACM Press, pp. 253–262. 9

[FDCO03] FLEISHMAN S., DRORI I., COHEN-OR D.: Bilateral mesh denoising. In *Proceedings of the SIGGRAPH'03 Conference* (San Diego, CA, July 2003), vol. 22(3), ACM Press, pp. 950–953. 2, 7, 8

[GH97] GARLAND M., HECKBERT P.: Surface simplification using quadratic error metrics. In *Proceedings of the SIGGRAPH'97 Conference* (Los Angeles, CA, Aug. 1997), ACM Press, pp. 209–216. 8, 9, 11

[GPB05] GOLDSTEIN J., PLATT J. C., BURGES C. J. C.: Redundant bit vectors for quickly searching high-dimensional regions. In *The Sheffield Machine Learning Workshop* (Sheffield, UK, Sept. 2005), vol. 3635 of Springer-Verlag Lecture Notes in Computer Science, pp. 137–158. 9

[Gut84] GUTTMAN A.: R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD'84 Conference* (Boston, MA, June 1984), ACM Press, pp. 47–57. 2, 3

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *Proceedings of the SIGGRAPH'92 Conference* (Chicago, IL, July 1992), ACM Press, pp. 71–78. 1, 7

[HS95] HJALTASON G. R., SAMET H.: Ranking in spatial databases. In *Advances in Spatial Databases — 4th International Symposium, SSD'95* (Portland, ME, Aug. 1995), Egenhofer M. J., Herring J. R., (Eds.), Series: Lecture Notes in Computer Science, Vol. 951, Springer, pp. 83–95. 2

[JDD03] JONES T. R., DURAND F., DESBRUN M.: Non-iterative, feature-preserving mesh smoothing. In *Proceedings of the SIGGRAPH'03 Conference* (San Diego, CA, July 2003), vol. 22(3), ACM Press, pp. 943–949. 1, 2, 7, 8

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital Michelangelo project: 3D scanning of large statues. In *Proceedings of the SIGGRAPH'00 Conference* (New Orleans, LA, July 2000), ACM Press, pp. 131–144. 1

[LRC*03] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann, San Francisco, CA, 2003. 8

[MA97] MOUNT D. M., ARYA S.: ANN: A library for approximate nearest neighbor searching. In *Proceedings of the 2nd Annual Center for Geometric Computing Workshop on Computational Geometry*. electronic edition, Durham, NC, Oct. 1997. 2, 5

[MN03] MITRA N. J., NGUYEN A.: Estimating surface normals in noisy point cloud data. In *Proceedings of the 19th ACM Symposium on Computational Geometry* (San Diego, CA, June 2003), ACM, pp. 322–328. 2

[PGK02] PAULY M., GROSS M., KOBBELT L. P.: Efficient simplification of point-sampled surfaces. In *VIS'02:Proceedings of the conference on Visualization* (Boston, MA, Oct. 2002), IEEE Computer Society, pp. 163–170. 8

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM Trans. Graph. 22*, 3 (2003), 641–650. 1

[RKV95] ROUSSOPOULOS N., KELLEY S., VINCENT F.: Nearest neighbor queries. In *Proceedings of the ACM SIGMOD'95 Conference* (San Jose, CA, May 1995), ACM Press, pp. 71–79. 2

[Sam06] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, July 2006. 1, 2, 3

[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the SIGGRAPH'91 Conference* (Las Vegas, NV, July 1991), ACM Press, pp. 289–298. 8

[UGMW01] ULLMAN J. D., GARCIA-MOLINA H., WIDOM J.: *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, 2001. 6, 7

[Vai89] VAIDYA P. M.: An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry 4*, 2 (1989), 101–115. 1

[WPH*04] WEYRICH T., PAULY M., HEINZLE S., KEISER R., SCANDELLA S.: Post-processing of scanned 3d surface data. In *Proceedings of Eurographics Symposium on Point-Based Graphics 2004* (Aire-La-Ville, Switzerland, June 2004), Eurographics Association, pp. 85–94. 2, 8

[XLOH04] XIA C., LU J., OOI B. C., HU J.: GORDER: an efficient method for KNN join processing. In *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases* (Toronto, Canada, Sept. 2004), Morgan Kaufmann, pp. 756–767. 2, 3, 6, 9