A Neighborhood-Based Clustering by Means of the Triangle Inequality and Reference Points

Marzena Kryszkiewicz and Piotr Lasek

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
mkr, p.lasek@ii.pw.edu.pl

Abstract. Grouping data into meaningful clusters is an important task of both artificial intelligence and data mining. An important group of clustering algorithms are density based ones that require calculation of neighborhoods of data points. In this paper, we present the *TI-k-Neighborhood-Index* algorithm that calculates *k*-neighborhoods for all points in a given dataset by means the triangle inequality and offer its variants. We prove experimentally that the *NBC* (Neighboorhood Based Clustering) clustering algorithm supported by our index outperforms *NBC* supported by known spatial indices such as VA-file and R-tree both in the case of low and high dimensional data.

1 Introduction

Grouping data into meaningful clusters is an important task of both artificial intelligence and data mining. An important group of clustering algorithms are density based ones that require calculation of neighborhoods of data points. In [4, 5], we have offered a solution to the problem of calculating a neighborhood within a given radius *Eps* (*Eps-neighborhood*) for each data point, which was based on the triangle inequality. Then, based on this solution, we offered *TI-DBSCAN* and *TI-DBSCAN-REF* as modified versions of the *DBSCAN* (Density Based Clustering with NOISE) algorithm [1] and proved its efficiency experimentally.

In [6], we examined a problem of an efficient calculation of the set of all *k* nearest points (*k-neighborhood*) for each data point. In this task, the value of a neighborhood radius is not initially restricted. Again, the theoretical solution we offered in [6] is based on the triangle inequality. Following this solution, we offered a new *TI-k-Neighborhood-Index* algorithm that calculates *k*-neighborhoods for all points in a given dataset. The usefulness of our method was verified by using it in the *NBC* (Neighborhood Based Clustering) clustering algorithm [9]. We proved experimentally that *NBC* supported by our index outperforms *NBC* supported by known spatial indices such as VA-file [2] and R-tree [3] both in the case of low and high dimensional data.

In this paper, we offer several variants of the *TI-k-Neighborhood-Index* algorithm. In particular, we offer a variant that uses many reference points to extend the applicability of reasoning based on triangle inequality property. We also offer a possibility of estimating a radius of a *k*-neighborhood of a given point based on the

knowledge of a radius of a *k*-neighborhood to which this point belongs. Finally, we consider a special case of determining *k*-neighborhood defined in terms of Minkowski distance and propose respective adaptations of the basic algorithm.

The paper has the following layout. Section 2 recalls the notions of an *Eps*-neighborhood and *k*-neighborhood. In Subsection 3.1, we recall a theoretical basis for calculating an *Eps*-neighborhood for a point within a given radius *Eps*, which we proposed in [4, 5]. In Subsection 3.2, we recall a theoretical basis for calculating a *k*-neighborhood for a point based on the triangle inequality which we proposed in [6]. In Section 4, we present the *TI-k-Neighborhood-Index* algorithm for calculating an index that stores *k*-neighborhoods for all points in a given dataset, which we offered there as well. In Section 5, we offer a modification of the *TI-k-Neighborhood-Index* algorithm that uses many reference points for determining *k*-neighborhoods. In Section 6, we propose a simple method for estimating a radius of a *k*-neighborhood of a point based on the knowledge of a radius of another point's *k*-neighborhood to which the given point belongs. Then, we propose a modification of the *TI-k-Neighborhood-Index* algorithm that uses this property. In Section 7, we propose an optimization of the *TI-k-Neighborhood-Index* algorithm in the case a Minkowski distance metric is applied for neighborhood calculation. Section 8 reports the performance of the *NBC* algorithm depending on a used index including our proposed index. Section 9 concludes the obtained results.

2 Basic Notions

In the sequel, the distance between two points p and q will be denoted by $distance(p, q)$. Please, note that one may use a variety of distance metrics. Depending on an application, one metric may be more suitable than the other. The most popular distance metric is *Euclidean distance*. *Euclidean distance* between points p and q is denoted by $Euclidean(p, q)$ and defined as follows:

$$Euclidean(p, q) = \sqrt{\sum_{i=1..n}(p_i - q_i)^2} .$$

If Euclidean distance is used, a neighborhood of a point has a spherical shape.
Another type of a popular distance metric is *Manhattan distance*. *Manhattan distance* between points p and q is denoted by $Manhattan(p, q)$ and defined as follows:

$$Manhattan(p, q) = \sum_{i=1..n}|p_i - q_i| .$$

When Manhattan distance is used, the shape of a neighborhood of a point is rectangular.

Euclidean and Manhattan distance metrics are special cases of *Minkowski distance*. *Minkowski distance* between points p and q is denoted by $Minkowski(p, q)$ and defined as follows:

$$Minkowski(p, q) = \sqrt[m]{\sum_{i=1..n}|p_i - q_i|^m} .$$

Clearly, for $m = 2$: $Minkowski(p, q) = Euclidean(p, q)$, while for $m = 1$: $Minkowski(p, q) = Manhattan(p, q)$.

For simplicity of the presentation, in our examples we will refer to Euclidean distance without loss of generality. Below, we recall definitions of an *Eps-neighborhood of a point* and *k-neighborhood of a point*.

Eps-neighborhood of a point p (denoted by $N_{Eps}(p)$) is defined as the set of points in dataset D that are different from *p* and distant from *p* by no more than *Eps*; that is,

$$N_{Eps}(p) = \{q \in D \mid q \neq p \wedge distance(p, q) \leq Eps\}.$$

Let *p* be a point in D. The set of all points in D that are different from *p* and closer to *p* than *q* will be denoted by *Closer*(*p*, *q*); that is,

$$Closer(p, q) = \{s \in D \mid s \neq p \wedge distance(s, p) < distance(q, p)\}.$$

Clearly, $Closer(p, p) = \varnothing$.

The *k-neighborhood of a point p* (denoted by *kNB*(*p*)) is defined as the set of all points *q* in D, $q \neq p$, such that the number of points that are different from *p* and closer to *p* than *q* is less than *k*; that is,

$$kNB(p) = \{q \in D \mid q \neq p \wedge |Closer(p, q)| < k\}.$$

Please note that for each point *p*, one may determine a value of parameter *Eps* in such a way that $N_{Eps}(p) = kNB(p)$. In the sequel, the least value of *Eps* such that $N_{Eps}(p) = kNB(p)$ will be called the *radius of kNB*(*p*).

Proposition 2.1. Let $Eps = \max(\{distance(q, p) \mid q \in kNB(p)\})$. Then $kNB(p) = N_{Eps}(p)$ and *Eps* is the radius of *kNB*(*p*).

Proposition 2.2. If $|N_{Eps}(p)| \geq k$, then $N_{Eps}(p) \supseteq kNB(p)$.

3 Using the Triangle Inequality for Efficient Determination of Neighborhoods

3.1 Efficient Determination of Eps-Neighborhoods

Let us start with recalling the triangle inequality property:

Property 3.1.1. (Triangle inequality property). For any three points *p*, *q*, *r*:

$$distance(p, r) \leq distance(p, q) + distance(q, r).$$

Property 3.1.2 presents an equivalent form of Property 3.1.1, which is more suitable for further considerations.

Property 3.1.2. (Triangle inequality property). For any three points *p*, *q*, *r*:

$$distance(p, q) \geq distance(p, r) - distance(q, r).$$

Now, we recall the results related to *Eps*-neighborhood we formulated in [4, 5].

Lemma 3.1.1 [4, 5]. Let D be a set of points. For any two points *p*, *q* in D and any point *r*:

$$distance(p, r) - distance(q, r) > Eps \Rightarrow q \notin N_{Eps}(p) \wedge p \notin N_{Eps}(q).$$

Proof. Let $distance(p, r) - distance(q, r) > Eps$ (*). By Property 3.1.2, $distance(p, q) \geq distance(p, r) - distance(q, r)$ (**). By (*) and (**), $distance(p, q) > Eps$, and $distance(q, p) = distance(p, q)$. Hence, $q \notin N_{Eps}(p)$ and $p \notin N_{Eps}(q)$. □

By Lemma 3.1.1, if we know that the difference of distances from two points p and q to some point r is greater than Eps, we are able to conclude that $q \notin N_{Eps}(p)$ without calculating the actual distance between p and q.

Theorem 3.1.1 [4, 5]. Let r be any point and D be a set of points ordered in a non-decreasing way with respect to their distances to r. Let p be any point in D, q_f be a point following point p in D such that $distance(q_f, r) - distance(p, r) > Eps$, and q_b be a point preceding point p in D such that $distance(p, r) - distance(q_b, r) > Eps$. Then:
a) q_f and all points following q_f in D do not belong to $N_{Eps}(p)$.
b) q_b and all points preceding q_b in D do not belong to $N_{Eps}(p)$.

3.2 Efficient Determination of *k*-Neighborhoods

Now, we will recall a theoretical basis useful for determining k-neighborhood of any point p ($kNB(p)$), which we offered in [6].

Theorem 3.2.1 [6]. Let r be any point and D be a set of points ordered in a non-decreasing way with respect to their distances to r. Let p be any point in D and Eps be a value such that $|N_{Eps}(p)| \geq k$, q_f be a point following point p in D such that $distance(q_f, r) - distance(p, r) > Eps$, and q_b be a point preceding point p in D such that $distance(p, r) - distance(q_b, r) > Eps$. Then:
a) q_f and all points following q_f in D do not belong to $kNB(p)$.
b) q_b and all points preceding q_b in D do not belong to $kNB(p)$.

Proof. Let r be any point and D be a set of points ordered in a non-decreasing way with respect to their distances to r.

a) Let p be any point in D, Eps be a value such that $|N_{Eps}(p)| \geq k$, and q_f be a point following point p in D such that $distance(q_f, r) - distance(p, r) > Eps$. Then by Theorem 3.1.1a, q_f and all points following q_f in D do not belong to $N_{Eps}(p)$ (*) and by Proposition 2.2, $N_{Eps}(p) \supseteq kNB(p)$. Hence, q_f and all points following q_f in D do not belong to $kNB(p)$.
b) The proof is analogous to the proof of Theorem 3.2.1a. □

Example 1 [6]. Let r be a point (0,0). Figure 1 shows a sample set D of two dimensional points. Table 1 illustrates the same set D ordered in a non-decreasing way with respect to the distance of its points to point r. Let us consider a determination of the k-neighboorhood of a point p = F for k = 3. Let us assume that we have calculated the distances between F, and points H, G, and C, respectively, and they are as follows: $distance$(F,H) = 1.64, $distance$(F,G) = 1.25, $distance$(F,C) = 1.77. Let Eps = max($distance$(F,H), $distance$(F,G), $distance$(F,C)); that is, Eps = 1.77. This means, that H,G,C $\in N_{Eps}$(F) and $|N_{Eps}(p)| \geq k$. Now, we note that the first point q_f following point F in D such that distance(q_f, r) – $distance$(F, r) > Eps is point A (distance(A, r) – $distance$(F, r) = 5.8 – 3.2 = 2.6 > Eps), and the first point q_b preceding point F in D such that $distance$(F, r) – $distance$(q_b, r) > Eps is K ($distance$(F, r) – $distance$(K, r) = 3.2 – 0.9 = 2.3 > Eps). By Theorem 3.2.1, points A,

K as well as the points that follow A (here, point B) and precede K in D (here, no point precedes K) do not belong to *kNB*(F). □

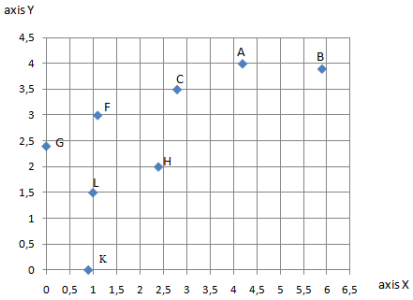

Fig. 1. Set of points D

Table. 1. Ordered set of points D from Fig. 1 with their distances to reference point *r*(0,0)

q∈D	X	Y	distance(q,r)
K	0,9	0,0	0,9
L	1,0	1,5	1,8
G	0,0	2,4	2,4
H	2,4	2,0	3,1
F	**1,1**	**3,0**	**3,2**
C	2,8	3,5	4,5
A	4,2	4,0	5,8
B	5,9	3,9	7,1

In the sequel, a point *r* to which the distances of all points in D have been determined will be called a *reference point*.

4 Building *k*-Neighborhood Index by Using Triangle Inequality with Regard to One Reference Point

If the number of points in D does not exceed *k*, then for each point *p* in D, all remaining points in D belong to *kNB*(*p*). In the sequel, we consider the case when there are more than *k* points in D.

In this section, we present the *TI-k-Neighborhood-Index* algorithm that uses Theorem 3.2.1 to determine *k*-neighborhoods for all points in a given dataset D and store them as a *k*-neighborhood index [6]. The algorithm starts with calculating the distance from each point in D to a reference point *r*, e.g. to the point with all coordinates equal to 0 or with all coordinates equal to the minimum values in their domains. Then the points in D are sorted wrt. their distance to *r*. Next, for each point *p* in D the *TI-k-Neighborhood* function, which calculates *kNB*(*p*), is called. The function identifies first those *k* points *q* following and preceding point *p* in the already ordered set D for which the difference between *distance*(*p*, *r*) and *distance*(*q*, *r*) is minimal. These points are considered as candidate elements of *kNB*(*p*). Then the radius *Eps* is calculated as the maximum of the real distances from these points to *p*. It is guaranteed that real *k* elements of *kNB*(*p*) lie within this radius from point *p*. Next, the remaining points preceding and following point *p* in the ordered set D (starting from points closer to *p* in D) are checked as potential elements of *kNB*(*p*) until the conditions specified in Theorem 3.2.1 are fulfilled. If so, no other points in D are checked as they are guaranteed not to belong to *kNB*(*p*). In order to speed up the algorithm, the value of *Eps* is modified each time a new candidate for an element of *kNB*(*p*) is identified.

```
Algorithm TI-k-Neighborhood-Index(set of points D, k);
/* assert: r denotes a reference point       */
/* assert: There are more than k points in D */
for each point p in set D do
    p.dist = Distance(p,r)
endfor;
sort all points in D non-decreasingly wrt. attribute dist;
for each point p in the ordered set D starting from
    the first point until last point in D do
    insert (position of point p, TI-k-Neighborhood(D, p, k))
        into k-Neighborhood-Index
endfor
```

```
function TI-k-Neighborhood(D, point p, k)
b = p;
f = p;
backwardSearch = PrecedingPoint(D, b);
forwardSearch = FollowingPoint(D, f);
k-Neighborhood = {};
i = 0;
Find-First-k-Candidate-Neighbours-Forward&Backward(D, p, b, f,
    backwardSearch, forwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Backward(D, p, b,
    backwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Forward(D, p, f,
    forwardSearch, k-Neighborhood, k, i);
p.Eps = max({e.dist| e ∈ k-Neighborhood});
Verify-k-Candidate-Neighbours-Backward(D, p, b, backwardSearch,
    k-Neighborhood, k);
Verify-k-Candidate-Neighbours-Forward(D, p, f, forwardSearch,
    k-Neighborhood, k);
return k-Neighborhood
```

```
function PrecedingPoint(D, var point p)
if there is a point in D preceding p then
    p = point immediately preceding p in D;
    backwardSearch = true
else
    backwardSearch = false
endif
return backwardSearch
```

```
function FollowingPoint(D, var point p)
if there is a point in D following p then
    p = point immediately following p in D;
    forwardSearch = true
else
    forwardSearch = false
endif
return forwardSearch
```

```
function Find-First-k-Candidate-Neighbours-Forward&Backward(D,
    var point p, var point b, var point f,
    var backwardSearch, var forwardSearch, var k-Neighborhood,
    k, var i)
while backwardSearch and forwardSearch and (i < k) do
    if p.dist - b.dist < f.dist - p.dist then
        dist = Distance(b, p);
        i = i + 1;
        insert element e = (position of b, dist) into k-Neighborhood
            holding it sorted wrt. e.dist;
        backwardSearch = PrecedingPoint(D, b)
    else
        dist = Distance(f, p);
        i = i + 1;
        insert element e = (position of f, dist) into k-Neighborhood
            holding it sorted wrt. e.dist;
        forwardSearch = FollowingPoint(D, f);
    endif
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Backward(D,var point p,
    var point b, var backwardSearch, var k-Neighborhood, k, var i)
while backwardSearch and (i < k) do
    dist = Distance(b, p);
    i = i + 1;
    insert element e = (position of b, dist) into k-Neighborhood
        holding it sorted wrt. e.dist;
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Forward(D, var point p,
    var point f, var forwardSearch, var k-Neighborhood, k, var i)
while forwardSearch and (i < k) do
    dist = Distance(f, p);
    i = i + 1;
    insert element e = (position of f, dist) into k-Neighborhood
        holding it sorted wrt. e.dist;
    forwardSearch = FollowingPoint(D, f)
endwhile
```

```
function Verify-k-Candidate-Neighbours-Backward(D, var point p,
    var point b, var backwardSearch, var k-Neighborhood, k)
while backwardSearch and ((p.dist − b.dist) ≤ p.Eps) do
    dist = Distance(b, p);
    if dist < p.Eps then
        i = |{e ∈ k-Neighborhood| e.dist = p.Eps}|;
        if |k-Neighborhood| - i ≥ k - 1 then
            delete each element e with e.dist = p.Eps
                from k-Neighborhood;
            insert element e = (position of b, dist) into
                k-Neighborhood holding it sorted wrt. e.dist;
            p.Eps = max({e.dist| e ∈ k-Neighborhood});
            /* Since k-Neighboorhood is sorted wrt. e.dist, only one */
            /* of its elements needs to be read to determine p.Eps.  */
        else
            insert element e = (position of b, dist) into
                k-Neighborhood holding it sorted wrt. e.dist;
```

```
        endif
    elseif dist = p.Eps
        insert element e = (position of b, dist) into
            k-Neighborhood holding it sorted wrt. e.dist
    endif
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

```
function Verify-k-Candidate-Neighbours-Forward(D, var point p,
    var point f, var forwardSearch, var k-Neighborhood, k)
while forwardSearch and ((f.dist − p.dist) ≤ p.Eps) do
    dist = Distance(f, p);
    if dist < p.Eps then
        i = |{e ∈ k-Neighborhood| e.dist = p.Eps}|;
        if |k-Neighborhood| - i ≥ k - 1 then
            delete each element e with e.dist = p.Eps
                from k-Neighborhood;
            insert element e = (position of f, dist) into
                k-Neighborhood holding it sorted wrt. e.dist;
            p.Eps = max({e.dist| e ∈ k-Neighborhood});
            /* Since k-Neighboorhood is sorted wrt. e.dist, only one */
            /* of its elements needs to be read to determine p.Eps.  */
        else
            insert element e = (position of f, dist) into
                k-Neighborhood holding it sorted wrt. e.dist;
        endif
    elseif dist = p.Eps
        insert element e = (position of f, dist) into
            k-Neighborhood holding it sorted wrt. e.dist
    endif
    forwardSearch = FollowingPoint(D, f)
endwhile
```

5 Building *k*-Neighborhood Index by Using Triangle Inequality with Regard to Many Reference Points

In this section, we offer a modification of the *TI-k-Neighborhood-Index* algorithm that uses many reference points for estimating the distance among pairs of points instead of one reference point. The modification we propose is an analogue of the proposal of using many reference points for data clustering with the *TI-DBSCAN-REF* algorithm, which we offered in [4, 5]. Additional reference points are used only when the basic reference point according to which the points in D are sorted does not imply that a given point q does not belong to *Eps*-neighborhood of another point p. The estimation of the distance between q and p by means of an additional reference point is based on Lemma 3.1.1. The actual distance between the two points q and p is calculated only when none of the reference points implies that $q \notin N_{Eps}(p)$.

Our proposed many-reference-points version of the *TI-k-Neighborhood-Index* algorithm was obtained by introducing a new function *Is-Candidate-Neighbor-by-Additional-Reference-Points* as well as changes in the *TI-k-Neighborhood-Index* algorithm itself and in the functions: *Verify-k-Candidate-Neighbours-Backward* and *Verify-k-Candidate-Neighbours-Forward*. Beneath, we provide the pseudo-code of

the new function and the functions that differ in one-reference-point and many-reference-points versions of the *TI-k-Neighborhood-Index* algorithm. The differences are highlighted in the code.

In many-reference-points version, all reference points are stored in array *RefPoints*. Example two reference points could be the point with all coordinates equal to the minimum values in their domains and the point with the first coordinate equal to the maximum domain value and the remaining coordinates equal to the minimum values in their domains. In addition, each point *p* in set D is associated with the array *Dists*, storing distances between *p* and reference points stored in *RefPoints*. The first point in *Dists* (that is, *Dists*[1]) plays the same role as the field *dist* in the one-reference-point version of *TI-k-Neighborhood-Index*. Consequently, all points in D will be sorted in a non-decreasing way with respect to their distances *Dists*[1] to the first reference point *RefPoints*[1].

```
Algorithm TI-k-Neighborhood-Index(set of points D, k);
/* assert: There are more than k points in D */
store reference points in array RefPoints;
for each point p in set D do
    for i = 1 to |RefPoints| do
        p.Dists[i] = Distance(p,RefPoints[i]);
    endfor
endfor
sort all points in D non-decreasingly wrt. attribute Dists[1];
for each point p in the ordered set D starting from
    the first point until last point in D do
    insert (position of point p, TI-k-Neighborhood(D, p, k))
        into k-Neighborhood-Index
endfor
```

The many-reference-points version of *Verify-k-Candidate-Neighbours-Backward* function differs from the one-reference-point version only in treating the points in D that were not excluded from the analysis carried out by means of triangle inequality property applied to the first reference point *RefPoints*[1]. Unlike in the one-reference-point version of *Verify-k-Candidate-Neighbours-Backward*, the many-reference-point version calls the *Is-Candidate-Neighbor-by-Additional-Reference-Points* function, which uses the information on the distances from *p* and *q* to at most all additional reference points to check if *q* is an invalid candidate for a neighbor of *p*. As soon as one of the reference points implies that *q* cannot be a neighbor of *p*, this result is returned to the *Verify-k-Candidate-Neighbours-Backward* function and next candidate for a neighbor of *p* is determined. Please, note that in such a case, unlike in the case of one-reference-version, the actual distance between *p* and *q* is not determined at all. Nevertheless, if none of the additional reference points, which were examined by *Is-Candidate-Neighbor-by-Additional-Reference-Points* function, did not imply that *q* could not be a neighbor of *p*, the calculation of the actual distance between *p* and *q* is carried out by the many-reference-points version of *Verify-k-Candidate-Neighbours-Backward*.

```
function Is-Candidate-Neighbor-by-Additional-Reference-Points(var point p,
    var point q, Eps)
candidateNeighbor = true;
i = 2;
while candidateNeighbor and (i ≤ |p.Dists|) do
    if |q.Dists[i] − p.Dists[i]| > Eps then
        candidateNeighbor = false
    else
        i = i + 1
    endif
endwhile
return candidateNeighbor;
```

```
function Verify-k-Candidate-Neighbours-Backward(D, var point p,
    var point b, var backwardSearch, var k-Neighborhood, k)
while backwardSearch and ((p.Dist[1] − b.Dist[1]) ≤ p.Eps) do
    if Is-Candidate-Neighbor-by-Additional-Reference-Points(p, b, p.Eps)
    then
        dist = Distance(b, p);
        if dist < p.Eps then
            i = |{e ∈ k-Neighborhood| e.dist = p.Eps}};
            if |k-Neighborhood| - i ≥ k - 1 then
                delete each element e with e.dist = p.Eps
                    from k-Neighborhood;
                insert element e = (position of b, dist)
                    into k-Neighborhood holding it sorted wrt. e.dist;
                p.Eps = max({e.dist| e ∈ k-Neighborhood});
            else
                insert element e = (position of b, dist)
                    into k-Neighborhood holding it sorted wrt. e.dist;
            endif
        elseif dist = p.Eps
            insert element e = (position of b, dist)
                into k-Neighborhood holding it sorted wrt. e.dist
        endif
    endif
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

The many-reference-points version of the many-reference-points version of *Verify-k-Candidate-Neighbours-Forward* function applies analogous adaptation as in the case of many-reference-points version of *Verify-k-Candidate-Neighbours-Backward*.

```
function Verify-k-Candidate-Neighbours-Forward(D, var point p,
    var point f, var forwardSearch, var k-Neighborhood, k)
while forwardSearch and ((f.Dist[1] - p.Dist[1]) ≤ p.Eps) do
    if Is-Candidate-Neighbor-by-Additional-Reference-Points(p, f, p.Eps)
    then
        dist = Distance(f, p);
        if dist < p.Eps then
            i = |{e ∈ k-Neighborhood| e.dist = p.Eps}};
            if |k-Neighborhood| - i ≥ k - 1 then
                delete each element e with e.dist = p.Eps
                    from k-Neighborhood;
                insert element e = (position of f, dist)
                    into k-Neighborhood holding it sorted wrt. e.dist;
                p.Eps = max({e.dist| e ∈ k-Neighborhood});
            else
                insert element e = (position of f, dist)
                    into k-Neighborhood holding it sorted wrt. e.dist;
            endif
        elseif dist = p.Eps
            insert element e = (position of f, dist)
                into k-Neighborhood holding it sorted wrt. e.dist
        endif
    endif;
    forwardSearch = FollowingPoint(D, f)
endwhile
```

6 Building Index of *k*-Neighborhoods with Additional Estimation of Radiuses

In this section, we start with proposing a simple method of estimating a radius of a *k*-neighborhood of a point, say *q*, based on the knowledge of a radius of a *k*-neighborhood of a point *p* such that $q \in kNB(p)$. Next, we offer the modification of the *TI-k-Neighborhood-Index* algorithm that utilizes the proposed estimation of radiuses of searched *k*-neighborhoods.

Proposition 6.1. Let *p*, *q* be points in D, $q \in kNB(p)$, Eps_p be the radius of $kNB(p)$ Eps_q be the radius of $kNB(q)$, and $Eps = Eps_p + distance(p, q)$. Then $N_{Eps}(q) \supseteq kNB(q)$ and $Eps_q \leq Eps$.

Proof. Let $q \in kNB(p)$, Eps_p be the radius of $kNB(p)$ Eps_q be the radius of $kNB(q)$ and $Eps = Eps_p + distance(p, q)$. Then, the set of all points within Eps_p radius from point *p*; that is, $kNB(p) \cup \{p\}$, lies within $Eps_p + distance(p, q)$ radius from point *q*; that is, is contained in $N_{Eps}(q) \cup \{q\}$. Hence, $N_{Eps}(q) \cup \{q\} \supseteq kNB(p) \cup \{p\}$. Now, since $|kNB(p) \cup \{p\}| \geq k + 1$, then $|N_{Eps}(q) \cup \{q\}| \geq k + 1$. In addition, $q \notin N_{Eps}(q)$, so $|N_{Eps}(q)| \geq k$. Thus, $N_{Eps}(q) \supseteq kNB(q)$, which implies that $Eps_q \leq Eps$. □

As follows from Proposition 6.1, if *p* and *q* are points in a dataset *D* and ε_p is the distance from *p* to its most distant *k* nearest neighbor in D, then all *k* nearest neighbors of *q* in D lie within $\varepsilon_p + dist(p, q)$ distance from *q*.

Figure 2 contains an illustration of Proposition 6.1.

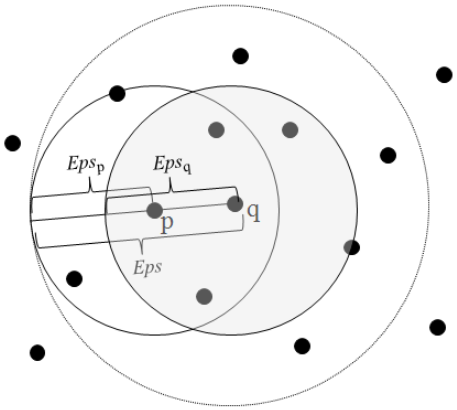

Fig. 2. $N_{Eps}(q) \supseteq kNB(q)$ and $Eps_q \leq Eps$ for $k = 5$, provided $q \in kNB(p)$, Eps_p is the radius of $kNB(p)$, Eps_q is the radius of $kNB(q)$, and $Eps = Eps_p + distance(p, q)$

Beneath, we provide the pseudo-code of these modified functions of the one-reference-point version of the *TI-k-Neighborhood-Index* algorithm that relate to the estimation of radiuses of *k*-neighborhoods as specified in Proposition 6.1. The introduced changes are highlighted in the code. The modified version of the *TI-k-Neighborhood-Index* algorithm contains also a new *EstimateNeighborhoodRadius* function. The function is called after calculating a *k*-neighborhood of each point *p* in D. Based on the radius *p.Eps* of *kNB*(*p*), the estimation *q.Eps* of the radius of each point *q* in *kNB*(*p*) is updated accordingly.

The many-reference-points version of the *TI-k-Neighborhood-Index* algorithm would require the introduction of analogous changes.

```
Algorithm TI-k-Neighborhood-Index(set of points D, k);
/* assert: r denotes a reference point       */
/* assert: There are more than k points in D */
for each point p in set D do
    p.dist = Distance(p,r);
    p.Eps = ∞;
endfor
sort all points in D non-decreasingly wrt. attribute dist;
for each point p in the ordered set D starting from
    the first point until last point in D do
    insert (position of point p, TI-k-Neighborhood(D, p, k))
        into k-Neighborhood-Index
endfor
```

```
function TI-k-Neighborhood(D, point p, k)
b = p;
f = p;
backwardSearch = PrecedingPoint(D, b);
forwardSearch = FollowingPoint(D, f);
k-Neighborhood = {}
i = 0;
Find-First-k-Candidate-Neighbours-Forward&Backward(D, p, b, f,
    backwardSearch, forwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Backward(D, p, b,
    backwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Forward(D, p, f,
    forwardSearch, k-Neighborhood, k, i);
p.Eps = max({e.dist| e ∈ k-Neighborhood});
Verify-k-Candidate-Neighbours-Backward(D, p, b, backwardSearch,
    k-Neighborhood, k);
Verify-k-Candidate-Neighbours-Forward(D, p, f, forwardSearch,
    k-Neighborhood, k);
EstimateNeighborhoodRadius(D, p, k-Neighborhood);
return k-Neighborhood;
```

```
function Find-First-k-Candidate-Neighbours-Forward&Backward(D,
    var point p, var point b, var point f,
    var backwardSearch, var forwardSearch, var k-Neighborhood,
    k, var i)
while backwardSearch and forwardSearch and (i < k) do
    if p.dist - b.dist < f.dist - p.dist then
        dist = Distance(b, p);
        if dist ≤ p.Eps then
            i = i + 1;
            insert element e = (position of b, dist) into k-Neighborhood
                holding it sorted wrt. e.dist
        endif
        backwardSearch = PrecedingPoint(D, b)
    else
        dist = Distance(f, p);
        if dist ≤ p.Eps then
            i = i + 1;
            insert element e = (position of f, dist) into k-Neighborhood
                holding it sorted wrt. e.dist
        endif
        forwardSearch = FollowingPoint(D, f);
    endif
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Backward(D,var point p,
    var point b, var backwardSearch, var k-Neighborhood, k, var i)
while backwardSearch and (i < k) do
    dist = Distance(b, p);
    if dist ≤ p.Eps then
        i = i + 1;
        insert element e = (position of b, dist) into k-Neighborhood
            holding it sorted wrt. e.dist
    endif
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Forward(D, var point p,
    var point f, var forwardSearch, var k-Neighborhood, k, var i)
while forwardSearch and (i < k) do
    dist = Distance(f, p);
    if dist ≤ p.Eps then
        i = i + 1;
        insert element e = (position of f, dist) into k-Neighborhood
            holding it sorted wrt. e.dist;
    endif
    forwardSearch = FollowingPoint(D, b)
endwhile
```

```
function EstimateNeighborhoodRadius(dataset D, point p,
    k-Neighborhood of point p);
for each element e = (pointId, dist) in k-Neighborhood do
    find point q in D that is referenced by e.pointId;
    q.Eps = min(q.Eps, (p.Eps + e.dist));
endfor
```

7 Building *k*-Neighborhood Index with Regard to Minkowski Distance

In this section, we offer a possibility of speeding up the *TI-k-Neighborhood-Index* algorithm in the case of applying Minkowski distance. In this case, the operation of comparison between $Minkowski(p, q) = \sqrt[m]{\sum_{i=1..n}|p_i - q_i|^m}$ and *Eps* can be substituted by the comparison $Minkowski^m(p, q) = \sum_{i=1..n}|p_i - q_i|^m$ and Eps^m.

Beneath, we provide the pseudo-code of these modified functions of the basic one-reference-point version of the *TI-k-Neighborhood-Index* algorithm that relate to this simple property. The introduced changes are highlighted in the code. Clearly, analogous changes can be introduced in all variants of the algorithm, we have already presented in earlier sections.

```
function TI-k-Neighborhood(D, point p, k)
b = p;
f = p;
backwardSearch = PrecedingPoint(D, b);
forwardSearch = FollowingPoint(D, f);
k-Neighborhood = {}; i = 0;
Find-First-k-Candidate-Neighbours-Forward&Backward(D, p, b, f,
    backwardSearch, forwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Backward(D, p, b,
    backwardSearch, k-Neighborhood, k, i);
Find-First-k-Candidate-Neighbours-Forward(D, p, f,
    forwardSearch, k-Neighborhood, k, i);
p.Eps^m = max({e.dist^m | e ∈ k-Neighborhood});
Verify-k-Candidate-Neighbours-Backward(D, p, b, backwardSearch,
    k-Neighborhood, k);
Verify-k-Candidate-Neighbours-Forward(D, p, f, forwardSearch,
    k-Neighborhood, k);
return k-Neighborhood
```

```
function Find-First-k-Candidate-Neighbours-Forward&Backward(D,
    var point p, var point b, var point f,
    var backwardSearch, var forwardSearch, var k-Neighborhood,
    k, var i)
while backwardSearch and forwardSearch and (i < k) do
    if p.dist - b.dist < f.dist - p.dist then
        dist^m = Minkowski^m(b, p);
        i = i + 1;
        insert element e = (position of b, dist^m) into k-Neighborhood
            holding it sorted wrt. e.dist^m;
        backwardSearch = PrecedingPoint(D, b)
    else
        dist^m = Minkowski^m(f, p);
        i = i + 1;
        insert element e = (position of f, dist^m) into k-Neighborhood
            holding it sorted wrt. e.dist^m;
        forwardSearch = FollowingPoint(D, f);
    endif
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Backward(D,var point p,
  var point b, var backwardSearch, var k-Neighborhood, k, var i)
while backwardSearch and (i < k) do
    dist^m = Minkowski^m(b, p);
    i = i + 1;
    insert element e = (position of b, dist^m) into k-Neighborhood
        holding it sorted wrt. e.dist^m;
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

```
function Find-First-k-Candidate-Neighbours-Forward(D, var point p,
  var point f, var forwardSearch, var k-Neighborhood, k, var i)
while forwardSearch and (i < k) do
    dist^m = Minkowski^m(f, p);
    i = i + 1;
    insert element e = (position of f, dist^m) into k-Neighborhood
        holding it sorted wrt. e.dist^m;
    forwardSearch = FollowingPoint(D, f)
endwhile
```

```
function Verify-k-Candidate-Neighbours-Backward(D, var point p,
    var point b, var backwardSearch, var k-Neighborhood, k)
Eps = (p.Eps^m)^(1/m);
while backwardSearch and ((p.dist - b.dist) ≤ Eps) do
    dist^m = Minkowski^m(b, p);
    if dist^m < p.Eps^m then
        i = |{e ∈ k-Neighborhood| e.dist^m = p.Eps^m}|;
        if |k-Neighborhood| - i ≥ k - 1 then
            delete each element e with e.dist^m = p.Eps^m
                from k-Neighborhood;
            insert element e = (position of b, dist^m) into
                k-Neighborhood holding it sorted wrt. e.dist^m;
            p.Eps^m = max({e.dist^m| e ∈ k-Neighborhood});
            Eps = (p.Eps^m)^(1/m);                          // optional
        else
            insert element e = (position of b, dist^m) into
                k-Neighborhood holding it sorted wrt. e.dist^m;
        endif
    elseif dist^m = p.Eps^m
        insert element e = (position of b, dist^m) into
            k-Neighborhood holding it sorted wrt. e.dist^m
    endif
    backwardSearch = PrecedingPoint(D, b)
endwhile
```

```
function Verify-k-Candidate-Neighbours-Forward(D, var point p,
    var point f, var forwardSearch, var k-Neighborhood, k)
Eps = (p.Eps^m)^(1/m);
while forwardSearch and ((f.dist - p.dist) ≤ Eps) do
    dist^m = Minkowski^m(b, p);
    if dist^m < p.Eps^m then
        i = |{e ∈ k-Neighborhood| e.dist^m = p.Eps^m}|;
        if |k-Neighborhood| - i ≥ k - 1 then
            delete each element e with e.dist^m = p.Eps^m
                from k-Neighborhood;
            insert element e = (position of f, dist^m) into
                k-Neighborhood holding it sorted wrt. e.dist^m;
            p.Eps^m = max({e.dist^m| e ∈ k-Neighborhood});
            Eps = (p.Eps^m)^(1/m);                          // optional
        else
            insert element e = (position of f, dist^m) into
                k-Neighborhood holding it sorted wrt. e.dist^m;
        endif
    elseif dist^m = p.Eps^m
        insert element e = (position of f, dist^m) into
            k-Neighborhood holding it sorted wrt. e.dist^m
    endif
    forwardSearch = FollowingPoint(D, f)
endwhile
```

8. Experimental Results

In this section we report the clustering, index building as well as preparation runtimes of different implementations of the *NBC* algorithm using *RTree*, *VA-File* and *TI-k-Neighboorhood-Index*.

Table 2. Datasets used in experiments and run times (in milliseconds) of NBC-RTree, NBC-*VAFile* and *TI-NBC-Sqr*. Notation: dim. – number of dimensions, card. – number of points, "-" – results not available within at least 12 hours or due to memory limitations, ind. – time of index building, clust. – time of clustering, prep. – time of initial sorting along with times of computing distances to the reference points. "ind." times include "prep." times.

No	dataset	dim.	card.	NBC-RTree		NBC-VAFILE		TI-NBC-Sqr		
				k=10		k=10, b=7		k=10		
				ind.	clust.	ind.	clust.	ind.	prep.	clust.
1	sequoia_2000_d2_1252	2	1252	557	1512	135	1343	70	9	4
2	sequoia_2000_d2_2503	2	2503	421	2523	33	5643	147	18	8
3	sequoia_2000_d2_3910	2	3910	629	4114	46	16460	267	38	9
4	sequoia_2000_d2_5213	2	5213	882	5290	61	30613	385	61	9
5	sequoia_2000_d2_6256	2	6256	1107	6598	74	49382	529	91	11
6	sequoia_2000_d2_62556	2	62556	-	-	-	-	27504	13744	69
7	manual_d2_2658	2	2658	426	3417	38	7547	146	24	9
8	manual_d2_14453	2	14453	2622	18617	-	-	1730	587	25
9	random_d3_50000	3	50000	10091	128204	-	-	53458	6455	85
10	random_d10_10000	10	10000	-	-	-	-	41414	218	20
11	random_d10_20000	10	20000	-	-	-	-	186060	878	50
12	random_d10_50000	10	50000	-	-	-	-	1299620	8522	173
13	random_d20_500	20	500	465	2217	-	-	267	7	2
14	random_d40_500	40	500	1175	3778	-	-	510	9	2
15	random_d200_500	200	500	-	-	-	-	2066	12	1
16	random_d200_1000	200	1000	-	-	-	-	9362	18	3
17	birch_d2_100000	2	100000	-	-	-	-	114197	89277	137
18	random_d100_1000	100	1000	-	-	-	-	4122	13	3
19	random_d50_10000	50	10000	-	-	-	-	215934	230	20
20	random_d100_10000	100	10000	-	-	-	-	478481	258	21
21	random_d50_20000	50	20000	-	-	-	-	893450	922	40
22	random_d100_20000	100	20000	-	-	-	-	1902731	1052	35
23	random_d5_100000	5	100000	-	-	-	-	1155215	67218	307
24	kddcup_98_d56_56000	56	56000	-	-	-	-	1200696	73949	164

In case of NBC implementations using the triangle inequality property we have introduced the following naming conventions: The *TI* prefix denotes that the NBC implementation employs the triangle inequality property. The suffixes such as Sqr, Mink, Ref, etc., represent different features of *TI-k-Neighboorhood-Index* described in

previous sections. And so: the *Mink* version operates on Minkowski distance without calculating distance roots, *Sqr* means that the implementations operates on squares of distances between points (equivalent to *Mink* with m = 2), *2Ref* – uses two reference points, *NRef*– uses as many reference points as is the dimensions count.

In the experiments, we used a number of datasets (and/or their subsamples) of different cardinality and dimensionality. In particular, we used widely known datasets such as: birch [8], SEQUOIA 2000 [7], and kddcup 98 [10] as well as datasets generated automatically (random) or manually. In several cases, it was impossible to cluster data based on R-tree or VA-file indices within 12 or more hours or due to memory limitations.

Table 3. Datasets used in experiments and run times (in milliseconds) of TI-NBC-Sqr and TI-NBC-Mink. Notation: dim. – number of dimensions, card. – number of points.

No	dataset	dim.	card.	TI-NBC-Sqr (the same as TI-NBC-Mink with m =2)			TI-NBC-Mink		
				k=10			k=10, m=1		
				ind.	prep.	clust.	ind.	prep.	clust.
1	sequoia_2000_d2_1252	2	1252	70	9	4	29	9	4
2	sequoia_2000_d2_2503	2	2503	147	18	8	39	19	7
3	sequoia_2000_d2_3910	2	3910	267	38	9	69	38	8
4	sequoia_2000_d2_5213	2	5213	385	61	9	96	61	9
5	sequoia_2000_d2_6256	2	6256	529	91	11	132	90	15
6	sequoia_2000_d2_62556	2	62556	27504	13744	69	21067	20635	75
7	manual_d2_2658	2	2658	146	24	9	52	17	8
8	manual_d2_14453	2	14453	1730	587	25	527	232	28
9	random_d3_50000	3	50000	53458	6455	85	8570	6396	140
10	random_d10_10000	10	10000	41414	218	20	333	228	13
11	random_d10_20000	10	20000	186060	878	50	1189	885	30
12	random_d10_50000	10	50000	1299620	8522	173	10136	8466	102
13	random_d20_500	20	500	267	7	2	21	7	2
14	random_d40_500	40	500	510	9	2	26	8	1
15	random_d200_500	200	500	2066	12	1	64	12	2
16	random_d200_1000	200	1000	9362	18	3	114	19	4
17	birch_d2_100000	2	100000	114197	89277	137	60283	59794	122
18	random_d100_1000	100	1000	4122	13	3	67	14	3
19	random_d50_10000	50	10000	215934	230	20	513	229	16
20	random_d100_10000	100	10000	478481	258	21	725	256	15
21	random_d50_20000	50	20000	893450	922	40	1485	924	36
22	random_d100_20000	100	20000	1902731	1052	35	2112	1038	20
23	random_d5_100000	5	100000	1155215	67218	307	74596	65534	933
24	kddcup_98_d56_56000	56	56000	1200696	73949	164	82724	71163	225

The experiments reported it Tables 2-4 were carried out for a typical value of k = 10. The VA-file depends on a parameter b that specifies the number of bits used to represent an attribute. The reported experiments were carried out for b = 7.

Table 4. Run times (in milliseconds) of *TI-NBC-Sqr-Est*, *TI-NBC-Sqr-Est-2Ref* and *TI-NBC-Sqr-Est-NRef*. Notation: dim. – number of dimensions, card. – number of points. Experiments were performed for the same datasets as in Table 2 and Table 3.

No	dim.	card.	TI-NBC-Sqr-Est			TI-NBC-Sqr-Est-2Ref			TI-NBC-Sqr-Est-NRef		
			k=10			k=10			k=10		
			ind.	prep.	clust.	ind.	prep.	clust.	ind.	prep.	clust.
1	2	1252	78	0	0	67	9	5	69	17	4
2	2	2503	140	15	0	131	19	8	121	22	8
3	2	3910	249	46	16	225	36	9	223	48	9
4	2	5213	343	62	16	339	62	10	327	71	10
5	2	6256	452	78	16	417	88	11	417	98	13
6	2	62556	32900	21465	78	24942	13428	77	22777	13569	79
7	2	2658	140	16	16	133	24	9	133	28	9
8	2	14453	1497	577	31	1477	560	28	1450	586	26
9	3	50000	41480	6006	78	56256	6505	84	59948	6947	86
10	10	10000	37985	218	15	41194	223	21	43280	313	23
11	10	20000	147046	858	47	228915	844	47	220709	1215	48
12	10	50000	1252267	9233	169	1512300	7769	171	1379011	6224	172
13	20	500	249	15	0	262	8	2	306	29	2
14	40	500	437	15	0	498	13	2	577	64	2
15	200	500	2340	0	0	2248	20	2	2754	1022	1
16	200	1000	8939	31	0	8623	33	3	10543	1872	3
17	2	100000	98452	79950	140	100155	80291	136	89247	74006	141
18	100	1000	4461	15	0	4321	22	3	5273	577	0
19	50	10000	238131	234	16	241865	250	20	302531	1544	32
20	100	10000	425243	250	16	462031	323	19	576343	5616	20
21	50	20000	1011166	968	45	1033965	951	41	1261466	3648	39
22	100	20000	1943006	1054	34	1804895	1191	31	2282391	10604	47
23	5	100000	1089107	69648	314	1603800	67938	312	1326846	67579	312
24	56	56000	1045500	72411	454	1172928	68359	156	1166976	63383	156

When it was possible, *TI-NBC* outperformed both *R-tree-NBC* (up to 2 orders of magnitude) and *VA-file-NBC* (up to 1 order of magnitude). In addition, *TI-NBC* turned out capable to cluster high dimensional data (up to a few hundreds dimensions). Moreover, when it comes to times of index building, the version of *TI-NBC* using the Manhattan distance (TI-NBC-Mink, m=1), turned out to be about ten times more efficient than the version using the Euclidean distance (*TI-NBC-Sqr*).

We have also tested how the number of reference points affects the efficiency of the TI-NBC algorithm (Table 4). It turned out, that runtimes all compared versions (*TI-*

NBC-Sqr-Est, *TI-NBC-Sqr-Est-2Ref*, *TI-NBC-Sqr-Est-NRef*) perform similarly, however the version with two reference points, in some cases, is slightly better than the others.

9. Conclusions

In the paper, we first have recalled a methodology and the *TI-k-Neighborhood-Index* for determining *k*-neighborhood of a given point based on the triangle inequality property applied with regard to one reference point. Then we have offered several variants of this algorithm. In particular, we have offered a variant that uses many reference points to extend the applicability of reasoning based on triangle inequality property. We also have offered a variant that uses estimation of a radius of a *k*-neighborhood of a given point based on the knowledge of radiuses of already calculated *k*-neighborhoods to which this point belongs. Finally, we considered a special case of determining *k*-neighborhood defined in terms of Minkowski distance and proposed respective an optimization.

References

[1] Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Database with Noise. In: Proc. of KDD'96 (1996) 226-231

[2] Blott, S., Weber, R., A Simple Vector Approximation File for Similarity Search in High-dimensional Vector Spaces, Technical Report 19, ESPRIT project HERMES, no. 9141, technical report number 19, March (1997)

[3] Guttman, A.: R-Trees: A Dynamic Index Structure For Spatial Searching. In: Proc. of ACM SIGMOD, Boston (1984) 47-57

[4] Kryszkiewicz, M., Lasek, P.: TI-DBSCAN: Clustering with DBSCAN by means of the triangle inequality. ICS Research Report 3/2010, Warsaw, April 2010

[5] Kryszkiewicz M., Lasek P.: TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. RSCTC 2010: 60-69

[6] Kryszkiewicz M., Lasek P.: A Neighborhood-Based Clustering by Means of the Triangle Inequality. IDEAL 2010: 284-291

[7] Stonebraker, M., Frew, J., Gardels, K., Meredith, J.: The SEQUOIA 2000 Storage Benchmark. In: Proc. of ACM SIGMOD, Washington (1993), 2-11

[8] Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: A New Data Clustering Algorithm and its Applications, Data Mining and Knowledge Discovery, 1 (2) (1997) 141-182

[9] Zhou, S., Zhao, Y., Guan, J., Huang, J.Z.: A Neighborhood-Based Clustering Algorithm. In: Proc. of PAKDD (2005) 361-371

[10] http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html