Proc. 5th Int. Workshop on Multimedia Data Mining (MDM/KDD '04), Seattle, WA, pp. 28-37, 2004

Efficient Indexing of Complex Objects for Density-based Clustering

Karin Kailing, Hans-Peter Kriegel, Martin Pfeifle, Stefan Schönauer

Institute for Computer Science
University of Munich
Oettingenstr. 67, 80538 Munich, Germany
{kailing,kriegel,pfeifle,schoenauer}@dbs.informatik.uni-muenchen.de

ABSTRACT

Databases are getting more and more important for storing complex objects from scientific, engineering or multimedia applications. Examples for such data are chemical compounds, CAD drawings or XML data. The efficient search for similar objects in such databases is a key feature. However, the general problem of many similarity measures for complex objects is their computational complexity, which makes them unusable for large databases. An area where this complexity problem is a strong handicap is that of density-based clustering where many similarity range queries have to be performed. In this paper, we combine and extend the two techniques of metric index structures and multistep query processing to improve the performance of range query processing. The efficiency of our methods is demonstrated in extensive experiments on real world data including graphs, trees and vector sets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining

General Terms

Performance

Keywords

Density-based Clustering, Metric Indexing, Multistep Query Processing

1. INTRODUCTION

Databases are getting more and more important for storing complex objects from scientific, engineering or multimedia applications. Examples for such data are chemical compounds, CAD drawings, XML data, web sites or color images. The efficient search for similar objects in such databases, for example to classify new objects or to cluster database objects, is a key feature in those application domains. Often a feature transformation is not possible, therefore a simple distance function like the Euclidean distance cannot be used. In this case, the use of more complex distance functions, like the edit distance for graphs or trees is necessary. However, a general problem of all such measures is their computational complexity, which disqualifies their use for large databases.

An area where this complexity problem is a strong handicap is that of clustering, one of the primary data mining tasks. Density-based clustering has proved to be successful for clustering complex objects [10, 8]. Density-based clustering algorithms like DBSCAN [7] or OPTICS [2] are based on range queries for each database object. Each range query requires a lot of exact distance calculations. Thus, these algorithms are only applicable to large collections of complex objects if those range queries are supported efficiently. When working with complex objects the necessary distance calculations are the time-limiting factor. For complex objects distance calculations are significantly more expensive than disk accesses. So the ultimate goal is to save as many distance calculations as possible.

One approach to improve the performance of range queries is to use a filter-refinement architecture. The core idea is to apply a filter criterion to the database objects in order to obtain a small set of candidate answers to a query. The final result is then retrieved from this candidate set through the use of the complex similarity measure. This reduces the number of expensive object distance calculations and speeds up the search process.

Another possibility is the use of a metric index structure. In [5] several efficient access methods for similarity search in metric spaces are presented. In most real world applications a static index structure is not acceptable, so dynamic index structures like the M-tree [6] are applied.

So far both above mentioned concepts, multi-step query processing and metric index structures have only been used separately. We claim that those concepts can beneficially be combined and that through the combination a significant speed-up compared to both separate approaches can be achieved. In this paper, we discuss how the two approaches can be combined and present some other techniques to improve the efficiency of range query processing. Filters can easily be used to speed-up the creation and the traversing of a metric index structure like the M-tree. Additionally, caching can be used to prevent that the same distance calculations are performed more than once. As DBSCAN [7] for example is only interested in getting all objects in the ε- neighborhood of a given query object,

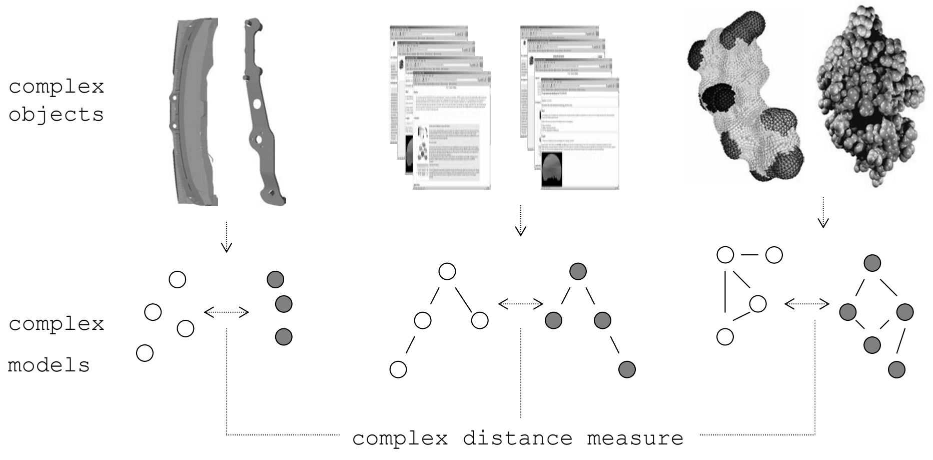

Figure 1: Examples of Complex Objects.

but does not need to know the actual distances, we introduce the concept of ”positive pruning” to save further distance calculations.

The remainder of the paper is organized as follows. In section 2, we present some recent work in the field of indexing and clustering complex objects. Section 3 presents our techniques used to save costly distance calculations while performing range queries. The performance gain of our new techniques is presented in section 4, while section 5.2 concludes the paper and gives some hints at future work.

2. MOTIVATION AND RELATED WORK

In the next subsection, we present three promising and approved modelling approaches and distance measures for complex objects (see figure 1 for an illustration). The evaluation part will show that in all those cases we achieve a performance gain using our new techniques. Afterwards we present some recent approaches for clustering and query processing on complex objects.

As this is an extremely broad field we do not make any claim on completeness, neither for the data types nor for the techniques presented. The main purpose of this section is to motivate the necessity of new techniques which allow efficient similarity range queries on complex objects.

2.1 Data Types of Complex Objects

2.1.1 Sets of Feature Vectors

For CAD applications, suitable similarity models can help to reduce the cost of developing and producing new parts by maximizing the reuse of existing parts. In [10] an effective and flexible similarity model for complex 3D CAD data is introduced, which helps to find and group similar parts. It is not based on the traditional approach of describing one object by a single feature vector but instead an object is mapped onto a set of feature vectors, i.e. an object is described by a vector set. An algorithm and a method for accelerating the processing of similarity queries on vector set data is presented. In the evaluation part we will show that we can significantly improve this approach for range queries.

2.1.2 Tree-Structured Data

In addition to a variety of content-based attributes, complex objects typically carry some kind of internal structure which often forms a hierarchy. Examples of such tree-structured data include chemical compounds, CAD drawings, XML documents or web sites. For similarity search it is important to take into account both the structure and the content features of such objects. A successful approach is to use the edit distance for tree structured data. However, as the computation of this measure is NP-complete, constrained edit distances like the degree-2 edit distance [19] have been introduced. They were successfully applied to trees for web site analysis [18], structural similarity of XML documents [14], shape recognition [15] or chemical substructure search [18]. While yielding good results, they are still computationally complex and, therefore, of limited benefit for searching or clustering in large databases. In [9] a filter and refinement architecture for the degree-2 edit distance is presented to overcome this problem. A set of new filter methods for structural and for content-based information as well as ways to flexibly combine different filter criteria are presented. With experiments on real world data the authors show that this approach is superior to metric index structures. But the experiment on k-nearest-neighbor queries also show that even the most complex filter which combines structural and content feature has to compute the object distances for 10 percent of the database in order to find the nearest neighbor. Again, we will show in the evaluation part, that our new techniques outperform the presented approach for range queries.

2.1.3 Graphs

Attributed graphs are another natural way to model structured data. Most known similarity measures for attributed graphs are either limited to a special type of graph or are computationally extremely complex, i.e. NP-complete. Therefore they are unsuitable for searching or clustering large collections. In [11], the authors present a new similarity measure for attributed graphs, called edge matching distance. They demonstrate, how the edge matching distance can be used for efficient similarity search in attributed graphs. Furthermore, they propose a filter-refinement architecture and an accompanying set of filter methods to reduce the number of necessary distance calculations during similarity search. Their experiments show that the matching distance is a meaningful similarity measure for attributed graphs and that it enables efficient clustering of structured data.

2.2 Clustering Complex Objects

In recent years, the research community spent a lot of attention to clustering resulting in a large variety of different clustering al-

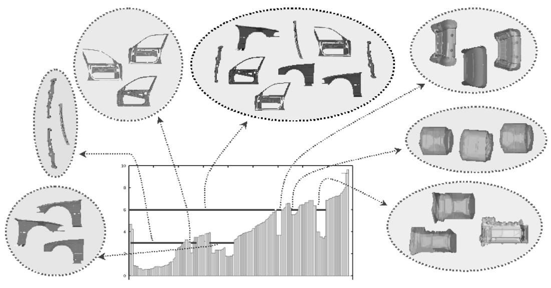
Figure 2: Browsing through cluster hierarchies.

gorithms. However, most of those algorithms were designed for vector data, so there is still a need for research on clustering complex objects.

2.2.1 *Density-Based Clustering of Complex Objects*

In this paper, we focus on the acceleration of density-based clustering algorithms like DBSCAN [7] and OPTICS [2], which are based on ε-range queries. Density-based clustering algorithms provide the following advantages:

1. They can be used for all kinds of metric data spaces and are not confined to vector spaces.
2. They are robust concerning outliers.
3. They have proved to be very efficient and effective in clustering all sorts of data.
4. OPTICS is – in contrast to most other algorithms – relatively insensitive to its two input parameters, ε and *MinPts*. The authors in [2] state that the input parameters just have to be large enough to produce good results.

2.2.2 *Clustering Multi-Represented Objects*

Traditional clustering algorithms are based on one representation space, usually a vector space. However, for complex objects often multiple representations exist for each object. Proteins for example are characterized by an amino acid sequence, a secondary structure and a 3D representation. In [8] an efficient density-based approach to cluster such multi-represented data, taking all available representations into account is presented. The authors propose two different techniques to combine the information of all available representations dependent on the application. The evaluation part shows that this approach is superior to existing techniques. The experiments were done for protein data that is represented by amino-acid sequences and text descriptions as well as for image data, where two different representations based on color histograms and segmentation trees were used.

2.2.3 *Visually Mining through Cluster Hierarchies*

In [4] the authors show how visualizing the hierarchical clustering structure of a database of objects can aid the user in his time consuming task to find similar objects (cf. figure 2). Based on reachability plots produced by the density-based clustering algorithm OPTICS [2], approaches which automatically extract the significant clusters in a hierarchical cluster representation along with suitable cluster representatives are proposed. These techniques can be used as a basis for visual data mining. The effectiveness and efficiency of this approach was shown for CAD objects from a German car manufacturer, and a sample of the Protein databank [3] containing approximately 5000 protein structures.

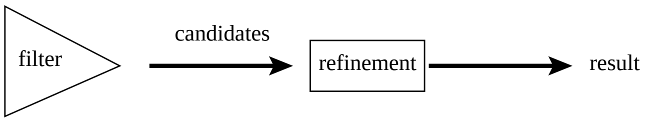

Figure 3: A multistep query processing architecture.

2.3 Query Processing on Complex Objects

2.3.1 *Multi-step Query Processing*

The main goal of a filter-refinement architecture, as depicted in figure 3, is to reduce the number of complex and, therefore, time consuming object distance calculations in the query process. To achieve this goal, query processing is performed in two or more steps. The first step is a filter step which returns a number of candidate objects from the database. For those candidate objects the exact object distance is then determined in the refinement step and the objects fulfilling the query predicate are reported. To reduce the overall search time, the filter step has to fulfill certain constraints. First of all, it is essential, that the filter predicate is considerably easier to evaluate than the exact similarity measure. Second, a substantial part of the database objects must be filtered out. Only if both conditions are satisfied, the performance gain through filtering is greater than the cost for the extra processing step. Additionally, the completeness of the filter step is essential. Completeness in this context means that all database objects satisfying the query condition are included in the candidate set. Available similarity search algorithms guarantee completeness if the distance function in the filter step fulfills the lower-bounding property. For any two objects O_p and O_q, a lower-bounding distance function d_f in the filter step has to return a value that is not greater than the exact object distance d_o of O_p and O_q, i.e. $d_f(O_p, O_q) \leq d_o(O_p, O_q)$. With a lower-bounding distance function it is possible to safely filter out all database objects which have a filter distance greater than the current query range because the exact object distance of those objects cannot be less than the query range. Using a multi-step query architecture requires efficient algorithms which actually make use of the filter step. Agrawal, Faloutsos and Swami proposed such an algorithm for range search [1].

2.3.2 *Metric Index Structures*

In some applications, objects cannot be mapped into feature vectors. However, there still exists some notion of similarity between objects, which can be expressed as a metric distance between the objects, i.e. the objects are embedded in a metric space. Several index structures for pure metric spaces have been proposed in the literature (see [5] for an overview). A well-known dynamic index structure for metric spaces is the M-tree [6]. The M-tree, which is explained in detail in section 3.1, aims at providing good I/O-performance as well as reducing the number of distance computations.

3. EFFICIENT RANGE-QUERIES ON COMPLEX OBJECTS

So far, the concepts of multi-step query processing and metric index structures have only been used separately. We claim that these concepts can beneficially be combined and that, through the combination, a significant speed-up compared to both separate approaches can be achieved. In the following, we will demonstrate the ideas for range queries with the M-tree as index structure and arbitrary filters fulfilling the lower-bounding criterion. It has to be noted that the techniques can also be applied to similar metric index structures like the Slim-tree [17].

This section is organized as follows. After introducing the necessary concepts for similarity range queries using the M-tree, we present the concept of ”positive pruning” in section 3.2. In section 3.3, we combine the two worlds of direct metric index structures and multi-step query processing based on filtering. Furthermore, we show in this section that filters cannot only be used for improving the query response time of an M-tree, but also for efficiently creating an instance of an M-tree. In section 3.4, we show how caching can be applied to accelerate the processing of similarity range queries.

3.1 Similarity Range Queries using the M-tree

The M-tree (*metric tree*) [6] is a balanced, paged and dynamic index structure that partitions data objects not by means of their absolute positions in the multi-dimensional feature space, but on the basis of their relative distances in this feature space. The only prerequisite is that the distance function between the indexed objects is metric. Thus, the M-tree's domain of applicability is quite general, and all sorts of complex data objects can be organized with this index structure.

The maximum size of all nodes of the M-tree is fixed. All database objects O_d or references to them are stored in the leaf nodes of an M-tree, along with their feature values and the distance $d(O_d, P(O_d))$ to their parent object $P(O_d)$. Inner nodes contain so-called *routing objects*, which correspond to database objects to whom a *routing role* was assigned by a promoting algorithm that is executed whenever a node has to be split. Additional to the object description and the distance to the parent object, routing objects O_r also store their *covering radius* $r(O_r)$ and a pointer $ptr(T(O_r))$ to the root node of their sub-tree, the so-called *covering tree* of O_r. For all objects O_d in this covering tree, the condition holds that the distance $d(O_r, O_d)$ is smaller or equal to the covering radius $r(O_r)$. This property induces a hierarchical structure of an M-tree, with the covering radius of a parent object always being greater or equal than all covering radii of their children and the root object of an M-tree storing the maximum of all covering radii.

Range queries are specified by a query object O_q and a range value ε by which the answer set is defined to contain all the objects O_d from the database that have a distance to the query object O_q of less than or equal to ε:

DEFINITION 1 (SIMILARITY RANGE QUERY). *Let $\mathcal{O}$ be a domain of objects and $DB \subseteq \mathcal{O}$ be a database. For a query object $O_q \in \mathcal{O}$ and a query range $\varepsilon \in \mathbb{R}_0^+$, the similarity range query $simRange : \mathcal{O} \times \mathbb{R}_0^+ \mapsto 2^{DB}$ returns the set*

$$simRange(O_q, \varepsilon) = \{O_d \in DB | dist(O_d, O_q) \leq \varepsilon\}.$$

Given a query object O_q and a similarity range parameter ε, a similarity range query $simRange(O_q, \varepsilon)$ starts at the root node of an M-tree and recursively traverses the whole tree down to the leaf level, thereby pruning all sub-trees which certainly contain no result objects.

A description of $simRange$ in pseudocode and the recursive procedure $rangeSearch$ used to traverse the M-tree is given in figure 4.

The sub-tree of a routing object O_r can be pruned, if the absolute value of the distance of the routing object's parent object O_p to the query object O_q, $d(O_p, O_q)$, minus the distance between O_r and O_p is greater than the covering radius of O_r plus ε:

$$|d(O_p, O_q) - d(O_p, O_r)| > r(O_r) + \varepsilon$$

```
1   simRange(queryObject $O_q$, range $\varepsilon$) $\rightarrow$ ResultSet
2       $result$ = NIL;
3       rangeSearch($root$, $O_q$, $\varepsilon$);
4       return $result$;
```

```
1   rangeSearch(Node $N$, queryObject $O_q$, range $\varepsilon$)
2       $O_p$ := parent object of node $N$;
3       IF $N$ is not a leaf THEN
4           FOR EACH $O_r$ in $N$ DO
5               IF $|d(O_p, O_q) - d(O_r, O_p)| \leq r(O_r) + \varepsilon$
6               THEN
7                   compute $d(O_r, O_q)$;
8                   IF $d(O_r, O_q) <= r(O_r) + \varepsilon$ THEN
9                       rangeSearch($ptr(T(O_r)), O_q, \varepsilon$);
10                  END IF
11              END IF
12          END FOR
13      ELSE
14          FOR EACH $O_d$ in $N$ DO
15              IF $|d(O_p, O_q) - d(O_d, O_p)| \leq \varepsilon$ THEN
16                  compute $d(O_d, O_q)$;
17                  IF $d(O_d, O_q) \leq \varepsilon$ THEN
18                      add $O_d$ to result;
19                  END IF
20              END IF
21          END FOR
22      END IF
```

Figure 4: Pseudocode description of similarity range search on M-trees.

A proof for this is given in [6]. Thus, as the distance between O_p and O_q has already been computed when accessing a node N, sub-trees can be pruned without further distance computations (see line 5 of the algorithm in figure 4).

3.2 Positive Pruning

A hierarchical index structure, like the M-tree, is composed of directory nodes with routing objects O_r which represent all objects in their respective subtree $T(O_r)$. For all objects $O \in T(O_r)$, $d(O_q, O_r) \leq r(O_r)$ holds. Efficient processing of range queries on the original M-tree is based on the concept of ”negative pruning”. During the query processing, certain subtrees are excluded from the search based on the following formula: $d(O_q, O_r) > r(O_r) + \varepsilon$ (see line 7 of the algorithm in figure 4).

In this section we introduce the concept of ”positive pruning”. If a directory node is completely covered by the query range, we can report all objects on the leaf level of the M-tree without performing any cost intensive distance computations (cf. figure 5).

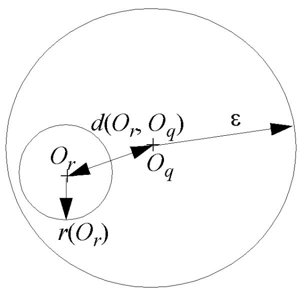

Figure 5: Positive Pruning for the M-tree.

LEMMA 1. *Let $O_q \in \mathcal{O}$ be a query object and $\varepsilon \in I\!R_0^+$ a query range. Furthermore, let O_r be a routing object in an M-tree with a covering radius $r(O_r)$ and a subtree $T(O_r)$. Then the following statement holds:*

$$d(O_r, O_q) + r(O_r) \le \varepsilon \Rightarrow \forall O \in T(O_r) : d(O, O_q) \le \varepsilon$$

PROOF. The following inequalities hold for all $O \in T(O_r)$ due to the triangle inequality and due to $d(O_r, O_q) + r(O_r) \le \varepsilon$:

$$\begin{aligned} d(O, O_q) &\le d(O, O_r) + d(O_r, O_q) \\ &\le r(O_r) + d(O_r, O_q) \le \varepsilon \end{aligned}$$

□

In the case of negative pruning, we skip the recursive tree traversal of a subtree $T(O_r)$, if the query range does not intersect the covering radius $r(O_r)$. In the case of positive pruning we skip all the distance calculations involved in the recursive tree traversal if the query range completely covers the covering radius $r(O_r)$. In this case we can report all objects stored in the corresponding leaf nodes of this subtree without performing any further distance computations. Figure 6 shows how this concept can be integrated into the original method $rangeSearch$ depicted in figure 4.

```
1     rangeSearch(Node $N$, queryObject $O_q$, range $\varepsilon$)
                         ⋮
7           compute $d(O_r, O_q)$;
7a          IF $d(O_r, O_q) + r(O_r) \le \varepsilon$ THEN
7b            report all objects in $T(O_r)$;
8           ELSE IF $d(O_r, O_q) <= r(O_r) + \varepsilon$ THEN
                         ⋮
```

Figure 6: Adaptation of similarity range search on M-trees for positive pruning.

This approach is very beneficial for accelerating density-based clustering on complex objects. DBSCAN for instance, only needs the information whether an object is contained in $simRange(O_q, \varepsilon) = \{O \in DB | d(O, O_q) \le \varepsilon\}$, but not the actual distance of this object to the query object O_q.

3.3 Combination of Filtering and Indexing

The M-tree reduces the number of distance calculations by partitioning the data space even if no filters are available. Unfortunately, the M-tree may suffer from the navigational cost related to the distance computations during the recursive tree traversal. On the other hand, the filtering approach heavily depends on the quality of the filters.

When combining both approaches, these two drawbacks are reduced. We use the filter distances to optimize the required number of exact object distance calculations needed to traverse the M-tree. Thereby, we do not save any I/O cost compared to the original M-tree, as the same nodes are traversed, but we save a lot of costly distance calculations necessary for the traversal. The filtering M-tree stores the objects along with their corresponding filter values within the M-tree. A similarity query based on the filtering M-tree always computes the filter distance values prior to the exact distance computations. If a filter distance value is already a sufficient criterion to prune branches of the M-tree, we can avoid the exact distance computation. If we have several filters, the filter distance computation always returns the maximum value of all filters.

The pruning quality of the filtering M-tree benefits from both the quality of the filters and the clustering properties of the index structure. In the following, we will show that the number of distance calculations used for range queries as well as for the creation of an M-tree can be optimized by using lower-bounding filters.

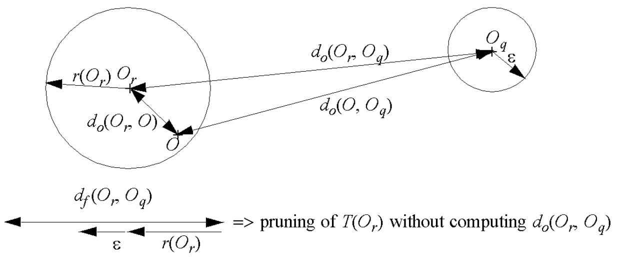

Figure 7: Similarity range query based on the filtering M-tree.

3.3.1 Range Queries

Similarity range queries are used to retrieve all objects from a database which are within a certain similarity range from the query object (cf. definition 1). By computing the filter distance prior to the exact distance we can save on many distance computations. Based on the following lemma, we can prune many subtrees without computing the exact distances between a query object O_q and a routing object O_r (cf. figure 7).

LEMMA 2. *Let $\mathcal{O}$ be a set of objects and $DB \subseteq \mathcal{O}$ a database. Furthermore, let $d_o, d_f : \mathcal{O} \times \mathcal{O} \mapsto I\!R_0^+$ be two distance functions, for which d_f lower bounds d_o, i.e. $\forall O_1, O_2 \in \mathcal{O} : d_f(O_1, O_2) \le d_o(O_1, O_2)$ holds. Let $O_q \in \mathcal{O}$, $\varepsilon \in I\!R_0^+$. For each routing object $O_r \in DB$ with covering radius $r(O_r) \in I\!R_0^+$ and subtree $T(O_r)$ the following statement holds:*

$$\begin{aligned} \forall O \in T(O_r) : (d_f(O_q, O_r) > r(O_r) + \varepsilon) \\ \Rightarrow d_o(O_q, O) > \varepsilon \end{aligned}$$

PROOF. As $\forall O_1, O_2 \in \mathcal{O} : d_f(O_1, O_2) \le d_o(O_1, O_2)$ holds, the following statement is true:

$$d_f(O_q, O_r) > r(O_r) + \varepsilon \Rightarrow d_o(O_q, O_r) > r(O_r) + \varepsilon$$

Based on the triangle inequality and our assumption that $d_o(O, O_r) \le r(O_r)$, we can prove the above lemma as follows:

$$\begin{aligned} d_f(O_q, O_r) &> r(O_r) + \varepsilon \\ \Rightarrow d_o(O_q, O_r) &> r(O_r) + \varepsilon \\ \Rightarrow d_o(O_q, O_r) - r(O_r) &> \varepsilon \\ \Rightarrow d_o(O_q, O_r) - d_o(O, O_r) &> \varepsilon \\ \Rightarrow d_o(O_q, O) &> \varepsilon \end{aligned}$$

□

Let us note that a similar optimization can be applied to the objects stored on the leaf level with the assumption that their 'covering radius' is 0. Figure 8 shows how this concept can be integrated into the original method $rangeSearch$ of figure 4.

3.3.2 Construction of an M-tree

Filters can also be used for accelerating the creation of an M-tree.

Insert. Figure 9 depicts the function *findSubTree* which decides which tree to follow during the recursive tree-traversal of the insert operation. The main idea is that we sort all objects according to the filter distance and then walk through this sorted list. Thereby, we first test those candidates which might not lead to an increase in the covering radius. If we detect a routing object for which no increase

```
1      rangeSearch(Node $N$, queryObject $O_q$, range $\varepsilon$)
                  ⋮
5            IF $|d(O_p,O_q) - d(O_r,O_p)| \leq r(O_r) + \varepsilon$
6            THEN
6a             compute $d_f(O_r,O_q)$
6b             IF $d_f(O_r,O_q) <= r(O_r) + \varepsilon$ THEN
7                compute $d(O_r,O_q)$;
8                IF $d(O_r,O_q) <= r(O_r) + \varepsilon$ THEN
                  ⋮
15           IF $|d(O_p,O_q) - d(O_d,O_p)| \leq \varepsilon$ THEN
15a            compute $d_f(O_d,O_q)$
15b            IF $d_f(O_d,O_q) \leq \varepsilon$ THEN
16               compute $d(O_d,O_q)$;
17               IF $d(O_d,O_q) \leq \varepsilon$ THEN
                  ⋮
```

Figure 8: Adaptation of similarity range search on M-trees for filtering.

is necessary, we postpone the reporting of this object. We first investigate all routing objects which are closer to the given query object and possibly also do not have to increase their covering radius. If several of those routing objects exist, we take the one closest to the inserted object. If no such routing object exists, we walk through the list until we have found the routing object which leads to a minimal increase of its covering radius. Let us note that this idea is closely related to the optimum multi-step k-nearest neighbor search algorithm [16] presented by Seidl and Kriegel.

Split. If a node overflow occurs due to an insertion, the node has to be split adequately. The 'ideal' split strategy should promote two new routing objects, such that for the resulting regions volume and overlap are minimized. Several different strategies for splitting a node are described in [6]. There the authors show that in most cases it is the best strategy to minimize the maximum of the resulting covering radii. This strategy, which is called *mM_Rad* is also the most complex in terms of distance computations. It considers all possible pairs of objects and after partitioning the set of entries, promotes the pair of objects for which the maximum of the two covering radii is minimal. Given a set of n entries and two routing objects, the generalized hyperplane decomposition is used to assign each of the n objects to one of the two routing objects. Although this leads to unbalanced splits, experimental results show that it is superior to techniques resulting in a balanced distribution.

In figure 10, it is shown how the filter distances can be used to speed-up the split of an M-tree node. The main idea is that we generate a priority queue containing pairs of promoting objects based on the filter distances. We walk through this list and if we detect that the *mM_Rad* value based on the filters is higher than the best already found *mM_Rad* value based on exact distance computations, we can stop. Thus we do not necessarily have to test all $O(n^2)$ pairs of promoting objects. Again this approach is similar to [16]. Furthermore, if we test two actual promoting objects O_{p1} and O_{p2}, we have to assign an object O either to O_{p1} or to O_{p2}. This test can be accelerated by computing first the actual distance between O and the promoting object for which the filter distance is smaller. If the resulting exact distance is still smaller than the filter distance to the other promoting object, we can save on the second exact distance computation. Note, that we can easily alter the function *nodeSplit* in such a way that it returns the two resulting nodes instead of the promoting objects.

```
findSubTree(RoutingObject $O_R$,Object $O$)→RoutingObject
  ActResult = (nil, false, $\infty$); // (object, InCovRad, distance)
  FOR EACH $O_r$ in $T(O_R)$ DO
    compute $d_f(O_r,O)$;
  END FOR
  $C_1 = \{O_r | d_f(O_r,O) - r(O_r) < 0\}$;
  $C_2 = T(O_R) \backslash C_1$;
  Sort all $O_r \in C_1$ and $O'_r \in C_2$ ascending according to
  $d_f(O_r,O)$ resulting in a SortedList =
  $\langle O_{r_1}, O_{r_2}, \ldots, O_{r_{|C_1|}} \rangle \circ \langle O'_{r_1}, O'_{r_2}, \ldots, O'_{r_{|C_2|}} \rangle$;
  FOR EACH $O_r$ in SortedList DO
    IF ActResult.InCovRad AND $O_r \in C_2$ THEN
      return ActResult.object;
    END IF
    IF $d_f(O_r,O) >$ ActResult.distance THEN
      IF ActResult.InCovRad THEN
        return ActResult.object;
      ELSE
        compute $d_o(O,O_r)$;
        IF $d_o(O,O_r) - r(O_r) < 0$ THEN
          ActResult = ($O_r$, true, $d_o(O,O_r)$);
        ELSE
          IF $d_o(O,O_r) - r(O_r) < ActResult.distance$
          $-r(ActResult.object)$ THEN
            ActResult = ($O_r$, false, $d_o(O,O_r)$);
          END IF
        END IF
      END IF
    ELSE
      compute $d_o(O,O_r)$;
      IF ActResult.InCovRad THEN
        IF $((d_o(O,O_r) - r(O_r) < 0)$ AND $(d_o(O,O_r)$
        $< ActResult.distance))$ THEN
          ActResult = ($O_r$, true, $d_o(O,O_r)$);
        END IF
      ELSE
        IF $d_o(O,O_r) - r(O_r) < 0$ THEN
          ActResult = ($O_r$, true, $d_o(O,O_r)$);
        ELSE
          IF $d_o(O,O_r) - r(O_r) < ActResult.distance$
          $-r(ActResult.object)$ THEN
            ActResult = ($O_r$, false, $d_o(O,O_r)$);
          END IF
        END IF
      END IF
    END IF
  END FOR
  return ActResult.object;
```

Figure 9: Pseudocode description of function *findSubTree* for inserting an object into an M-tree.

3.4 Caching Distance Calculations

In this section, we present a further technique which helps to avoid costly distance computations for index construction and query processing.

3.4.1 Cache Based Construction

If we have to cope with distance computations which are more expensive than accessing secondary storage, we suggest to store the already processed distance computations to disk. Especially when splitting the same overflowing node repeatedly, accessing stored distance computation values can speed up the insertion process, since otherwise the same distances are computed several times.

```
nodeSplit (Node $N$) → PromotingObjects
  ActResult = ((nil, nil),∞); //((PromotingObject1, PromotingObject2), mMRad)
  FOR EACH pair of objects $(O_i, O_j)$ of node $N$ DO
    compute $d_f(O_i, O_j)$;
  END FOR
  Compute for each of those pairs $(O_i, O_j)$ the mMRad value mMRad_filter
  based on the filters;
  Sort the resulting mMRad_filter values ascending,
  resulting in a SortedList = $\langle (O_{a_1}, O_{b_1}, mMrad\_filter_1), \ldots,$
  $(O_{a_n}, O_{b_n}, mMRad\_filter_n) \rangle$;
  FOR EACH object $O_i$ of node $N$ DO
    IF $mMRad\_filter_i >$ ActResult.mMRad THEN
      return ActResult;
    END IF
    $mMRad_i$ = 0;
    Sort all objects $O_k$ of node $N$ descending according to
    $min(d_f(O_k, O_{a_i}), d_f(O_k, O_{b_i}))$;
    FOR EACH object $O_k$ of this sorted list DO
      IF $d_f(O_k, O_{a_i}) < d_f(O_k, O_{b_i})$ THEN
        compute $d_o(O_k, O_{a_i})$;
        IF $d_o(O_k, O_{a_i}) < d_f(O_k, O_{b_i})$ THEN
          $mMRad_i = max(mMRad_i, d_o(O_k, O_{a_i}))$;
        ELSE
          compute $d_o(O_k, O_{b_i})$;
          $mMRad_i =$
          $max(mMRad_i, min(d_o(O_k, O_{a_i}), d_o(O_k, O_{b_i})))$;
        END IF
      ELSE
        compute $d_o(O_k, O_{b_i})$;
        IF $d_o(O_k, O_{b_i}) < d_f(O_k, O_{a_i})$ THEN
          $mMRad_i = max(mMRad_i, d_o(O_k, O_{b_i}))$;
        ELSE
          compute $d_o(O_k, O_{a_i})$;
          $mMRad_i =$
          $max(mMRad_i, min(d_o(O_k, O_{a_i}), d_o(O_k, O_{b_i})))$;
        END IF
      END IF
      IF $mMRad_i >$ ActResult.mMRad THEN
        break;
      END IF
    END IF
    IF $mMRad_i <$ ActResult.mMRad THEN
      ActResult = $(O_{a_i}, O_{b_i}, mMRad_i)$;
    END IF
  END FOR
  return ActResult;
```

Figure 10: Pseudocode description of function *nodeSplit* for the M-tree.

3.4.2 Cache Based Range Queries

Efficient query processing of range queries also benefits from the idea of caching distance calculations. During the navigation through the M-tree directory, the same distance computations may have to be carried out several times. Although each object O is stored only once on the leaf level of the M-tree, it might be used several times as routing object. Furthermore, we often have the situation that distance calculations carried out on the directory level have to be repeated at the leaf level.

As shown in figure 4 a natural way to implement range queries is by means of recursion resulting in a depth-first search. We suggest to keep all distance computations in main memory which have been carried out on the way from the root to the actual node. After leaving the node, i.e. when exiting the recursive function, we delete all distance computations carried out at this node. This limits the actual main memory footprint to $O(h \cdot b)$ where h denotes the maximum height of a tree and b denotes the maximum number of stored elements in a node. Even in multi-user environments this rather small worst-case main memory footprint is tolerable. The necessary adaptations of the rangeSearch algorithm are drafted in figure 11.

```
1      rangeSearch(Node $N$, queryObject $O_q$, range $\varepsilon$)
                                        ⋮
6        distCache($N, O_r, O_q$);
                                        ⋮
16       distCache($N, O_d, O_q$);
                                        ⋮
22     END IF
22a    deleteCache($N$);
```

```
distCache(Node $N$, Object $O_1$, Object $O_2$) → float
  result = hashtable.lookup($O_1, O_2$);
  IF result = null then THEN
    result = compute d($O_1, O_2$);
    hashtable.add($N, O_1, O_2$,result);
  END IF
  return $result$;
```

```
deleteCache(Node $N$)
  hashtable.delete($N$);
```

Figure 11: Adaptation of similarity range search on M-trees for Caching.

4. EVALUATION

To show the efficiency of our approach, we chose the applications and data types described in section 2 and performed extensive experiments. All algorithms were implemented in Java 1.4 and the experiments were run on a workstation with a Xeon 1.7 GHz processor and 2 GB main memory under Linux. We implemented the M-tree as described in [6]. As in all cases the time for distance calculations was dominating the runtime of a range query, we only show the number of distance calculations and not the runtime.

4.1 CAD Vector Set Data

For the experiments with this data type, we used the similarity model presented in [10], where CAD objects were represented by a vector set consisting of 7 vectors in 6D. All experiments were carried out on a data set containing 5,000 CAD objects from an American aircraft producer. As distance measure between sets of feature vectors we used the minimal matching distances which can be computed in $O(k^3)$, where k denotes the cardinality of the point set, by means of the Kuhn-Munkres [12, 13] algorithm. As filter, we used the centroid filter introduced in [10].

4.1.1 Creation of M-tree

The generation of the optimized M-tree was carried out without caching (cf. figure 12) and with caching (cf. figure 13). Without cashing, the number of necessary distance calculations is very high, due to the repeated splitting of nodes. Note that the number of distance calculations for one node split is quadratic w.r.t. the number of elements of this node. In this case, our $nodeSplit$ algorithm only needs $1/4$ of the distance calculations while still producing the same M-tree. If we apply caching, the overall number of required distance computations is much smaller as many distance computations necessary for splitting a node can be fetched from disk. In this case our $findSubTree$-function allows us to reduce the number of required distance calculations even further, i.e. the number of distance computations is bisected. To sum up, both optimizations, which are based on the exploitation of available filter information, allow us to build up an M-tree much more efficiently.

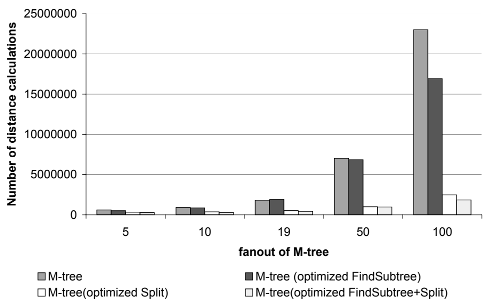

Figure 12: Creation without Caching Distance Calculations.

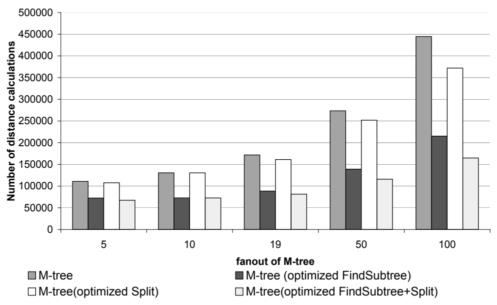

Figure 13: Creation with Caching Distance Calculations.

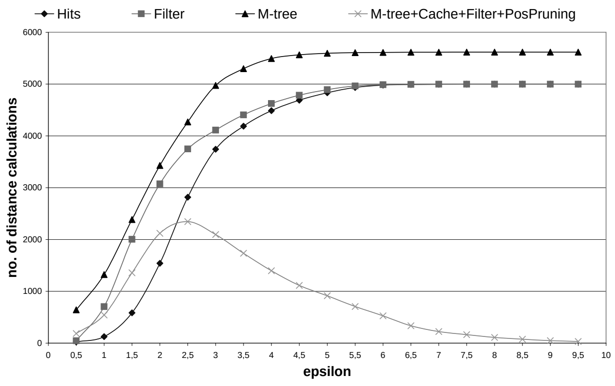

Figure 14: Comparison of our best technique to M-tree and filtering for vector set data .

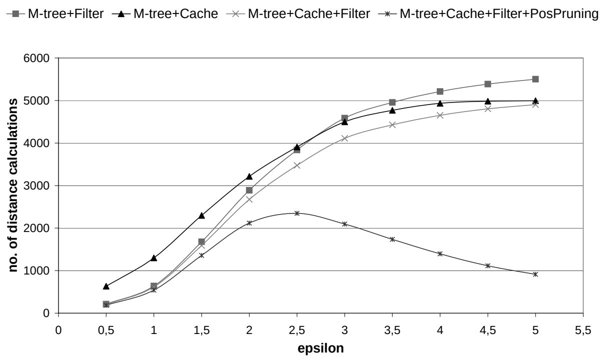

Figure 15: Comparison of our techniques for vector set data.

4.1.2 *Range Queries*

Figure 14 and 15 show in what way the different approaches for range query processing depend on the chosen ε-value. Figure 14 shows that for the investigated data set, the original M-tree is the worst access method for all ε-values. On the other hand, the pure filter performs very well. For this data set, reasonable ε-values for density-based clustering would be about 1 for DBSCAN and about 2 for OPTICS. In this parameter range, our approach clearly outperforms both the filter and especially the original M-tree.

In figure 15 one can see that for small ε-values, we benefit from the filtering M-tree, whereas for higher values we benefit from caching and positive pruning.

Furthermore, we clustered the data set using OPTICS [2] which forms the basis for the visual data mining tool presented in subsection 2.2. With a suitable parameter setting for OPTICS we achieved a speed-up of 16% compared to the centroid filter, 33% compared to the original M-tree and 104% compared to the sequential scan. Let us note, that the average cardinality of the result set of each range query was almost 2,000 which limits the best achievable speed-up to 150%.

4.2 Image Data

Image data are a good example for multi-represented complex objects. A lot of different similarity models exist for image data, each having its own advantages and disadvantages. Using for example text descriptions of images, one is able to cluster all images related to a certain topic, but these images need not look alike. Using color histograms instead, the images are clustered according to the distribution of color in the image. The approach for clustering multi-represented objects presented in [8] is able to get the best out of all these different types of representations. We present some experiments for image data represented as trees or graphs, where the efficiency of range query processing is especially important.

4.2.1 *Tree Structured Image Data*

Images can be described as segmentation trees. Thereby, an image is first divided into segments of similar color, then a tree is created from those segments by iteratively applying a region growing algorithm which merges neighboring segments if their colors are sufficiently alike. As similarity measure for the resulting trees, we used the degree-2 edit distance and implemented the filter refinement architecture as described in [9]. We used a sample set of 10,000 color TV-Images. For the experiments we chose reasonable epsilon values for the multi-represented clustering algorithm.

Figure 16 shows that we achieve a significant speed-up compared to the original M-tree. As can be seen we also outperform the pure filtering approach.

4.2.2 *Graph Structured Image Data*

To extract graphs from the images, they were segmented with a region growing technique and neighboring segments were connected by edges to represent the neighboring relationship. We used the edge matching distance and the image data set as described in [11]. The filter presented in this paper is almost optimal, i.e. the number of unnecessary distance calculations during query process-

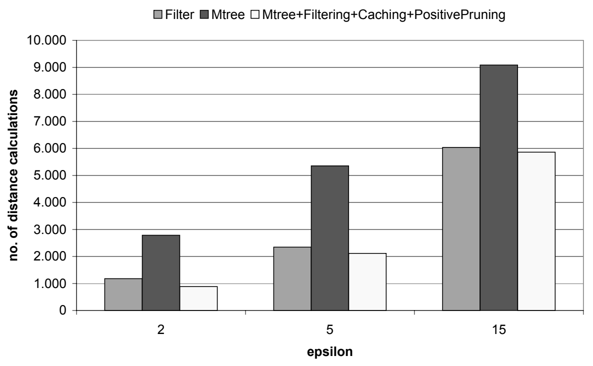

Figure 16: Comparison of our best technique to M-tree and filtering for tree structured data.

ing is very low. Even in this case our techniques is as good as the filter.

To show the robustness of our approach against the filter selectivity, we reduced it in a stepwise process. We weighted the original filter distances with constant factors to decrease the filter selectivity. Figure 17 shows that independently of the filter selectivity, our approach outperforms the original M-Tree by a factor of almost 2 and is at least as good as the pure filtering approach.

5. CONCLUSIONS

5.1 Summary

In this paper, we showed that there exist a lot of interesting application areas for density-based clustering of complex objects. Density-based clustering is based on similarity range queries where the similarity measures used for complex objects are often computationally very complex which makes them unusable for large databases. To overcome the efficiency problems, metric index structures or multi-step query processing are applied. We combined and extended these approaches to achieve the best from two worlds. More precisely, we presented three improvements for metric index structures, i.e. positive pruning, the combination of filtering and indexing and caching. In a broad experimental evaluation based on real world data sets, we showed that a significant speed-up for similarity range queries is achieved with our approach. By means of our new techniques, application areas like visually mining through cluster hierarchies of complex objects or clustering of complex multi-represented objects can be extended to larger databases.

5.2 Potentials for Future Work

In this paragraph, we shortly describe how the introduced optimized M-tree can be used for effectively and efficiently navigating through massive data sets. In Section 2.2.3, a data mining tool was sketched, called BOSS [4]. BOSS is based on the density-based hierarchical clustering algorithm OPTICS and on suitable cluster recognition and representation algorithms. Density-based clustering algorithms are able to detect arbitrarily shaped clusters, which are advantageous in application areas as for instance trend detection in spatial databases. On the other hand, in the area of similarity search clusters of spherical shapes are often more desirable. The optimized M-tree cannot only be used for computing a hierarchical density-based clustering efficiently, but it can also be utilized as a new data mining tool helping the user in his time-consuming task to find similar objects. Each directory node of an M-tree consists of objects representing all elements stored in the corresponding spherical subtrees. Thus, the tree itself can be regarded as a hierarchical clustering which, additionally, efficiently supports all kinds of similarity queries, e.g. ε-range queries. Furthermore, the optimizations introduced in this paper allow to build up an optimized M-tree much more efficiently than carrying out a complete hierarchical density-based clustering. In order to increase the quality, i.e. to minimize the overlap between subtrees of the optimized M-tree, we carry out update operations similar to update operations on Slim-trees [17]], i.e. we propose to use a variant of the slim-down algorithm trying to keep the tree tight. The quality of the resulting dynamic browsing tool could be measured by means of numerical values reflecting the degree of overlapping nodes (cf. the fat-factor and the bloat-factor presented in [17]). In our future work, we want to elaborate the trade-off between quality and efficiency of a new dynamic data-mining browsing tool which is based on the optimized M-tree as introduced in this paper.

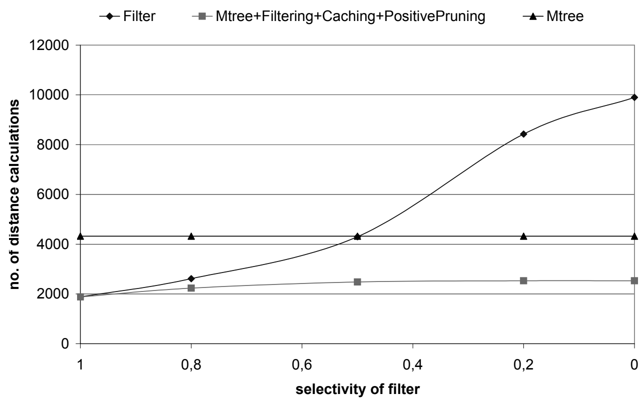

Figure 17: Comparison of our techniques for graph data.

6. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, 1993.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. “OPTICS: Ordering Points to Identify the Clustering Structure”. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD’99), Philadelphia, PA*, pages 49–60, 1999.

[3] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. “The Protein Data Bank”. *Nucleic Acids Research*, 28:235–242, 2000.

[4] S. Brecheisen, H.-P. Kriegel, P. Kröger, and M. Pfeifle. Visually mining through cluster hierarchies. In *Proc. SIAM Int. Conf. on Data Mining (SDM’04), Orlando, FL*, 2004.

[5] E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB’97, Proc. of 23rd International Conference on Very Large Databases, August 25-29, 1997, Athens, Greece*, pages 426–435, 1997.

[7] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD’96), Portland, OR*, pages 291–316. AAAI Press, 1996.

[8] K. Kailing, H.-P. Kriegel, A. Pryakhin, and M. Schubert. Clustering multi-represented objects with noise. In *Proc. 8th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD’04), Sydney, Australia*, 2004.

[9] K. Kailing, H.-P. Kriegel, S. Schönauer, and T. Seidl. Efficient similarity search for hierachical data in large databases. In *Proc. 9th Int. Conf. on Extending Database Technology (EDBT 2004)*, 2004.

[10] H.-P. Kriegel, S. Brecheisen, P. Kröger, M. Pfeifle, and M. Schubert. Using sets of feature vectors for similarity search on voxelized cad objects. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD’03), San Diego, CA*, pages 587–598, 2003.

[11] H.-P. Kriegel and S. Schönauer. Similarity search in structured data. In *Proc. 5th International Conference, DaWaK 2003, Prague, Czech Republic*, pages 309–319, 2003.

[12] H. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[13] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the SIAM*, 6:32–38, 1957.

[14] A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proc. 5th Int. Workshop on the Web and Databases (WebDB 2002), Madison, Wisconsin, USA*, pages 61–66, 2002.

[15] T. B. Sebastian, P. N. Klein, and B. B. Kimia. Recognition of shapes by editing shock graphs. In *Proc. 8th Int. Conf. on Computer Vision (ICCV’01), Vancouver, BC, Canada*, volume 1, pages 755–762, 2001.

[16] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proc. ACM SIGMOD Int. Conf. on Managment of Data*, pages 154–165, 1998.

[17] C. J. Traina, A. Traina, B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Proc. 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000*, pages 51–65, 2000.

[18] J. T. L. Wang, K. Zhang, G. Chang, and D. Shasha. Finding approximate patterns in undirected acyclic graphs. *Pattern Recognition*, 35(2):473–483, 2002.

[19] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 7(1):43–57, 1996.