# DeLiClu: Boosting Robustness, Completeness, Usability, and Efficiency of Hierarchical Clustering by a Closest Pair Ranking

Elke Achtert, Christian Böhm, and Peer Kröger

Institute for Computer Science, University of Munich, Germany
{achtert,boehm,kroegerp,}@dbs.ifi.lmu.de

**Abstract.** Hierarchical clustering algorithms, e.g. Single-Link or OPTICS compute the hierarchical clustering structure of data sets and visualize those structures by means of dendrograms and reachability plots. Both types of algorithms have their own drawbacks. Single-Link suffers from the well-known single-link effect and is not robust against noise objects. Furthermore, the interpretability of the resulting dendrogram deteriorates heavily with increasing database size. OPTICS overcomes these limitations by using a density estimator for data grouping and computing a reachability diagram which provides a clear presentation of the hierarchical clustering structure even for large data sets. However, it requires a non-intuitive parameter $\varepsilon$ that has significant impact on the performance of the algorithm and the accuracy of the results. In this paper, we propose a novel and efficient $k$-nearest neighbor join closest-pair ranking algorithm to overcome the problems of both worlds. Our density-link clustering algorithm uses a similar density estimator for data grouping, but does not require the $\varepsilon$ parameter of OPTICS and thus produces the optimal result w.r.t. accuracy. In addition, it provides a significant performance boosting over Single-Link and OPTICS. Our experiments show both, the improvement of accuracy as well as the efficiency acceleration of our method compared to Single-Link and OPTICS.

## 1 Introduction

Hierarchical clustering methods determine a complex, nested cluster structure which can be examined at different levels of generality or detail. The complex cluster structure can be visualized by concepts like dendrograms or reachability diagrams. The most well-known hierarchical clustering method is Single-Link [1] and its variants like Complete-Link and Average-Link [2]. Single-Link suffers from the so-called single-link effect which means that a single noise object bridging the gap between two actual clusters can hamper the algorithm in detecting the correct cluster structure. The time complexity of Single-Link and its variants is at least quadratic in the number of objects.

Another hierarchical clustering algorithm is OPTICS [3], which follows the idea of density-based clustering [4], i.e. clusters are regions of high data density separated by regions of lower density. OPTICS solves some of the problems of Single-Link but only to the expense of introducing new parameters *minPts* and $\varepsilon$. The latter is not very intuitive and critical for both, performance of the algorithm and accuracy of the result.

If $\varepsilon$ is chosen too low, fundamental information about the cluster structure is lost, if it is chosen too high the performance of the algorithm decreases dramatically.

In this paper, we introduce a novel hierarchical clustering algorithm *DeLiClu* (Density Linked Clustering) that combines the advantages of OPTICS and Single-Link by fading out their drawbacks. Our algorithm is based on a closest pair ranking (CPR). The objective of a CPR algorithm is: given two sets $R$ and $S$ of feature vectors, determine in a first step that pair of objects $(r, s) \in (R \times S)$ having minimum distance, in the next step the second pair, and so on. Well-known CPR algorithms like [5] operate on static data sets which are not subject to insertions or deletions after initialization of the ranking. Our new DeLiClu algorithm, however, needs a ranking algorithm where after each fetch operation for a new pair $(r, s)$ the object $s$ is deleted from $S$ and inserted into $R$. We show how the ranking algorithm can be modified to allow the required update operations without much additional overhead and how Single-Link can be implemented on top of a CPR. This allows the use of an index structure which makes the algorithm more efficient without introducing the parameter $\varepsilon$ like OPTICS does. Finally, we describe how the density-estimator of OPTICS can be integrated into our solution.

The rest of this paper is organized as follows: Sec. 2 discusses related work. In Sect 3 our novel algorithm is described. Sec. 4 presents an experimental evaluation. Sec. 5 concludes the paper.

## 2 Related Work

*Hierarchical Clustering.* Hierarchical clustering algorithms produce a nested sequence of clusters, resulting in a binary tree-like representation, a so-called dendrogram. The root of the dendrogram represents one single cluster, containing the $n$ data points of the entire data set. Each of the $n$ leaves of the dendrogram corresponds to one single cluster which contains only one data point. Hierarchical clustering algorithms primarily differ in the way they determine the similarity between clusters. The most common method is the Single-Link method [1] which measures the similarity between two clusters by the similarity of the closest pair of data points belonging to different clusters. This approach suffers from the so-called single-link effect, i.e. if there is a chain of points between two clusters then the two clusters may not be separated. In the Complete-Link method the distance between two clusters is the maximum of all pairwise distances between the data points in the two clusters. Average-Link clustering merges in each step the pair of clusters having the smallest average pairwise distance of data points in the two clusters. A major drawback of the traditional hierarchical clustering methods is that dendrograms are not really suitable to display the full hierarchy for data sets of more than a few hundred compounds. Even for a small amount of data, a reasonable interpretation of the dendrogram is almost impossible due to its complexity. The single-link effect can also be seen in the figure: as an impact of the connection line between the two clusters Single-Link computes no clearly separated clusters.

OPTICS [3] is another hierarchical clustering algorithm, but uses the concept of density based clustering and thus reduces significantly the single-link effect. Additionally, OPTICS is specifically designed to be based on range queries which can be efficiently supported by index-based access structures. The density estimator used by

OPTICS consists of two values for each object, the core distance and the reachability distance w.r.t. parameters *minPts* $\in \mathbb{N}$ and $\varepsilon \in \mathbb{R}$. The clustering result can be displayed in a so-called reachability plot that is more appropriate for very large data sets than a dendrogram. A reachability plot consists of the reachability values on the y-axis of all objects plotted according to the cluster order on the x-axis. The "valleys" in the plot represent the clusters, since objects within a cluster have lower reachability distances than objects outside a cluster. Figure 1 shows examples of reachability plots with different parameter settings for $\varepsilon$ and *minPts*. The effect of *minPts* to the resulting cluster structure is depicted in the left part of Figure 1. The upper part shows a reachability plot resulting from an OPTICS run with *minPts* $= 2$ where no meaningful cluster structure has been detected. If the value of *minPts* is increased as in the lower part of the figure, the two clusters in the data set can be seen as valleys in the reachability plot. The second parameter $\varepsilon$ is much more difficult to determine but has a considerable impact on the efficiency and the accuracy of OPTICS. If $\varepsilon$ is chosen too small, fundamental information about the cluster structure will be lost. The right part of figure 1 shows this effect in the upper diagram where the information about clusters consisting of data points with reachability values greater than $\varepsilon = 12$ is no longer existent.

*Closest Pair Ranking.* The closest pair problem is a classical problem of computational geometry [6]. The intention is to find those two points from given data sets $R$ and $S$ whose mutual distance is the smallest. The CPR determines in the first step that pair of objects in $R \times S$ having the smallest distance, in the next step the second pair, etc. The number of pairs to be reported is *a priori* unknown. In the database context the CPR problem was introduced first in [5], calling it distance join. An incremental algorithm based on the R-Tree family is proposed. For each data set $R$ and $S$ a spatial index is constructed as input. The basic algorithm traverses the two index structures, starting at the root of the two trees. The visited pairs of nodes are kept in a priority queue sorted by their distances. If the first entry of the priority queue exists of a pair of data points, then the pair is reported as the next closest pair. Otherwise, the pair is expanded and all possible pairs formed by inspecting the children of the two nodes are inserted into the priority queue. The algorithm terminates if all closest pairs are reported or the query is stopped by the user. CPR algorithms operate on static data sets, i.e. they do not support insertions or deletions of objects after initializing the ranking query. Our new DeLiClu algorithm, however, needs shifting object $s$ from $S$ to $R$ after reporting pair $(r, s)$. In Section 3 we propose a solution for this special case.

## 3 Density-Linked Clustering

Our new algorithm DeLiClu combines the advantages of Single-Link and OPTICS by fading out the drawbacks mentioned in Section 2. To achieve these requirements we introduce a density-smoothing factor *minPts* into hierarchical clustering and use as representation of the clustering result reachability plots like OPTICS. In contrast to OPTICS we avoid the introduction of the non-intuitive parameter $\varepsilon$ which is critical for both, performance of the algorithm and completeness of the result. In addition, we improve the performance over both algorithms by applying powerful database primitives such as the similarity join and a CPR, and by applying index structures for feature spaces.

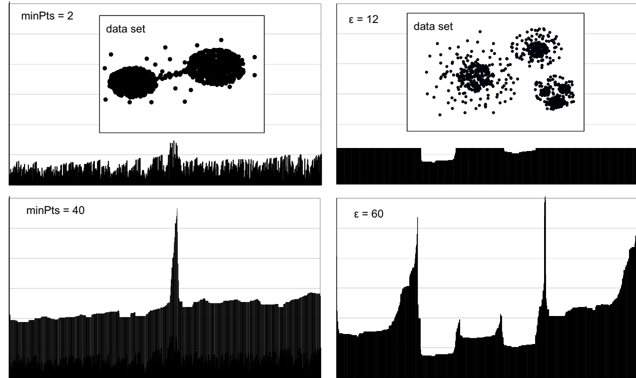



**Fig. 1.** Impact of parameters *minPts* and $\varepsilon$

### 3.1 General Idea of DeLiClu

Typical hierarchical clustering algorithms work as follows: They keep two separate sets of points, those points which have already been placed in the cluster structure and those which have not. In each step, one point of the latter set is selected and placed in the first set. The algorithm always selects that point which minimizes the distance to any of the points in the first set. Assume the algorithm has already done part of its work, and some of the points have already been placed in the cluster structure. What actually happens then is that the closest pair is selected between the set $R$ of those points which are already assigned to clusters and the set $S$ of the points which are not yet processed. This means, we can also reformulate the main loop of the general algorithm into:

determine the closest pair $(r, s) \in (R \times S)$;
migrate $s$ from $S$ into $R$;
append $s$ to cluster structure / reachability plot;

Note that we still have to render more precisely what exactly we mean by the notion *closest pair* because we have to integrate the density-based smoothing factor *minPts* into this notion. Additionally, since the reachability plot shows for each object its reachability distance we have to define a proper density distance for our DeLiClu algorithm. However, this will be done in Section 3.3 and until then, we simply mean the closest pair according to the Euclidean distance and assign each object with its closest pair or nearest neighbor distance to the reachability plot.

If the closest pair from $(R \times S)$ would be determined in each step from scratch, we would do a lot of unnecessary work. Instead, we like to save the status of processing from one call of the closest pair determination to the next one. But since we migrate object $s$ from $S$ to $R$ after the closest pair $(r, s)$ has been reported, we need a ranking algorithm which supports insertions or deletions after initialization. We show in the next section how the standard algorithm [5] can be extended to allow the required object migration during the ranking. The core of our DeLiClu clustering algorithm now is:

1. Let $R$ contain an arbitrary start object from data set $\mathcal{D}$;
2. Let $S$ be $\mathcal{D} \setminus R$;
3. Initialize the CPR over $(R \times S)$;
4. Take the next pair $(r, s)$ from the ranking;
5. Migrate $s$ from $S$ into $R$;
6. Append $s$ to the reachability plot;
7. Continue with step (4) until all points are handled;

The critical remaining aspects are the migration of point $s$ from $S$ into $R$ (step 5) and the introduction of the density-based smoothing factor *minPts* and a proper density distance definition.

### 3.2 Closest Pair Ranking with Object Migration

The original algorithm for CPR without object migration requires the two data sets to be stored in hierarchical index structures such as R-trees [7]. The algorithm uses a priority queue into which pairs of nodes and pairs of data objects can be inserted. The entries in the priority queue are ordered by ascending distances between the pair of objects (nodes, respectively) in the data space. Upon each request, the algorithm dequeues the top pair. If it is a pair of data objects, it is reported as the result of the request. Otherwise, the pair is expanded, i.e. for all pairs of child nodes the distances are determined and the pairs are inserted into the queue. Several strategies exist to decide which of the elements of a pair is expanded (left, right, or both). We assume here a symmetric expansion of both elements of the pair. Further, we assume that both indexes have exactly the same structure. Although the tree for $R$ initially contains only the arbitrarily chosen start element, we use a full copy of the directory of $S$ for convenience, because this method facilitates insertion of any element of $S$ into $R$. We simply use the same path as in the tree storing $S$. No complex insert and split algorithm has to be applied for insertion.

Whenever a new element $s$ is inserted into the index storing the data set $R$, we have to determine a suitable path $P = (root, node_1, ..., node_h, s)$ from the root to a leaf node for this element (including the element itself). Comparing the nodes of this path with the nodes of the index for $S$, we observe that some node pairs might already have been inserted into the priority queue, others may not. Some of the pairs (e.g. $(root_R, root_S)$) might have even already been removed from the priority queue. We call such removed pairs *processed*. Processed pairs are a little bit problematic because they require catch-up work for migrated objects. Processed pairs can be easily found by traversing the tree $S$ top-down. A pair should be in the queue if the parent pair has already been processed (i.e. has a distance smaller than the current top element of the priority queue), but the pair itself has a distance higher than the top element.

After a pair of objects $(r, o)$ has been processed, the formerly not handled object $o$ is migrated from $S$ to the set of already processed objects $R$. The catch-up work which now has to be done consists of the insertion of all pairs of objects (nodes, respectively) $(o, s) \in R \times S$ into the priority queue for which the parent pair of nodes $(o.parent, s.parent)$ has already been processed. The complete recursive method is called reInsertExpanded and is shown in Figure 2. Initially, reInsertExpanded is called with the complete path of the migrated object $o$ in $R$ and the root node of $S$.

```
reInsertExpanded(Object[] path, Object o)

  if (path[0], o) is a pair of objects then
    insert the pair (path[0], o) into priority queue;
  if (path[0], o) is a pair of nodes and has not yet been expanded then
    insert the pair (path[0], o) into priority queue;
  if (path[0], o) is a pair of nodes and has already been expanded then
    determine all child nodes o_child of o;
    reInsertExpanded(tail(path), o_child);
```

**Fig. 2.** Algorithm reInsertExpanded

### 3.3 The Density Estimator MinPts

Until now, we have re-engineered the Single-Link method without applying any density estimator for enhancing the robustness. Our re-engineering has great impact on the performance of the algorithm because now a powerful database primitive is applied to accelerate the algorithm. We will show in Section 4 that the performance is significantly improved. But our new implementation also offers an easy way to integrate the idea of the density estimator *minPts* into the algorithm without using the difficult parameter $\varepsilon$ of OPTICS. To determine the reachability distance of an object shown in the reachability plot we consider additionally the $k$-nearest neighbor distance of the point where $k =$ *minPts*. We call this distance density distance and it is formally defined as follows:

**Definition 1 (density distance).** *Let $\mathcal{D}$ be a set of objects, $q \in \mathcal{D}$ and* DIST *be a distance function on objects in $\mathcal{D}$. For minPts $\in \mathbb{N}$, minPts $\leq |\mathcal{D}|$ let $r$ be the minPts-nearest neighbor of $q$ w.r.t.* DIST. *The* density distance *of an object $p \in \mathcal{D}$ relative from object $q$ w.r.t. minPts is defined as*

$$\text{DENDIST}_{minPts}(p, q) = \max\{\text{DIST}(q, r), \text{DIST}(q, p)\}.$$

The density distance of of an object $p$ relative from object $q$ is an asymmetric distance measure that takes the density around $p$ into account and is defined as the maximum value of the *minPts*-nearest neighbor distance of $p$ and the distance between $p$ and $q$. Obviously, the density distance of DeLiClu is equivalent to the reachability distance of OPTICS w.r.t. the same parameter *minPts* and parameter $\varepsilon = \infty$. Our algorithm DeLiClu can adopt the density-based smoothing factor *minPts* by ordering the priority queue using the density distance rather than the Euclidean distance. The rest of the algorithm remains unchanged. Obviously, this modification can be done without introducing the parameter $\varepsilon$. The cluster hierarchy is always determined completely, unlike in OPTICS. And in contrast to OPTICS a guaranteed complete cluster result is not payed with performance deterioration.

The $k$-nearest neighbor distance where $k =$ *minPts* can be determined for all points in a preprocessing step which applies a $k$-nearest neighbor join of the data set. Some methods have been proposed for this purpose [8, 9] but unfortunately none for the simple $R$-tree and its variants. Therefore, we apply a new algorithm which is described in the next section.

### 3.4 The $k$-NN Join on the R-tree

The $k$-nn join combines each of the points of $R$ with its $k$ nearest neighbors in $S$. Algorithms for the $k$-nn join have been reported in [8] and in [9]. The first algorithm is based on the MuX-index structure [10], the latter is on top of a grid order. Unfortunately, there is no $k$-nn join algorithm for the R-tree family. Thus, in the following we present a $k$-nn join algorithm based on the R-tree [7] and its variants, e.g. R*-tree[11].

Formally we define the $k$-nn join as follows:

**Definition 2** **($k$-nn join $R \ltimes S$).** *Let $R$ and $S$ be sets of objects, and* $\textsc{Dist}$ *be a distance function between objects in $R$ and $S$. $R \ltimes S$ is the smallest subset of $R \times S$ that contains for each point of $R$ at least k points of $S$ and for which the following condition holds:*

$$\forall (r, s) \in R \ltimes S, \forall (r, s') \in R \times S \setminus R \ltimes S : \textsc{Dist}(r, s) < \textsc{Dist}(r, s')$$

Essentially, the $k$-nn join combines each point of the data set $R$ with its $k$-nearest neighbors in the data set $R$. Each point of $R$ appears in the result set exactly $k$ times. Points of $S$ may appear once, more than once (if a point is among the $k$-nearest neighbors of several points in $R$) or not at all (if a point does not belong to the $k$-nearest neighbors of any point in $R$).

For the $k$-nn join $R \ltimes S$ based on the R-tree it is assumed that each data set $R$ and $S$ is stored in an index structure belonging to the R-tree family. The data set $R$ of which the nearest neighbors are searched for each point is denoted as the *outer point set*. Consequently, $S$ is the *inner point set*. The data pages of $R$ and $S$ are processed in two nested loops whereas each data page of the outer set $R$ is accessed exactly once. The outer loop iterates over all data pages $pr$ of the outer point set $R$ which are accessed in an arbitrary order. For each data page $pr$, the data pages $ps$ of the inner point set $S$ are sorted in ascending order to their distance to $pr$. For each point $r$ stored in the data page $pr$, a data structure for the $k$- nearest neighbor distances, short a $k$-nn distance list, is allocated. The distances of candidate points are maintained in these $k$-nn distance lists until they are either discarded and replaced by smaller distances of better candidate points or until they are confirmed to be the actual nearest neighbor distances of the corresponding point. A distance is confirmed if it is guaranteed that the database cannot contain any points being closer to the given object than this distance. The last distance value in the $k$-nn distance list belonging to a point $r$ is the (actual) $k$-nn distance of $r$: points and data pages beyond that distance need not to be considered. The pruning distance of a data page is the maximum (actual) $k$-nn distance of all points stored in this page. All data pages $ps \in S$ having a distance from a given data page $pr \in R$ that exceeds the pruning distance of the data page $pr$ can be safely neglected as join-partners of that data page $pr$. Thus, in the inner loop only those data pages $ps$ have to be considered having a distance to the current data page $pr$ less or equal than the pruning distance of $pr$. Analogous, all points $s$ of a data page $ps$ having a distance to a current point $r$ greater than the current $k$-nn distance of $r$ can be safely pruned and do not have to be taken into consideration as candidate points.

### 3.5 Algorithm DeLiClu

The algorithm DeLiClu is given in Figure 3. In a preprocessing step, the $k$-nearest neighbor distance for all points is determined as described in Section 3.4. In the follow-

```
DeLiClu(SetOfObjects S)
  kNNJoin(S,S);
  copy the index storing S to the index storing R;
  s := start object ∈ S;
  write (s, ∞) to output;
  migrate s from S to R;
  add pair (S.root, R.root) to priority queue;
  while S ≠ ∅ do
    p:= minimum pair in priority queue;
    if p = (n_S, n_R) is a pair of nodes then
      insert all combinations of (n_S.children, n_r.children) into priority queue;
    else p = (s, r) is a pair of objects
      write (s, denDist(s, r)) to output;
      reInsertExpanded(path(s), root);
```

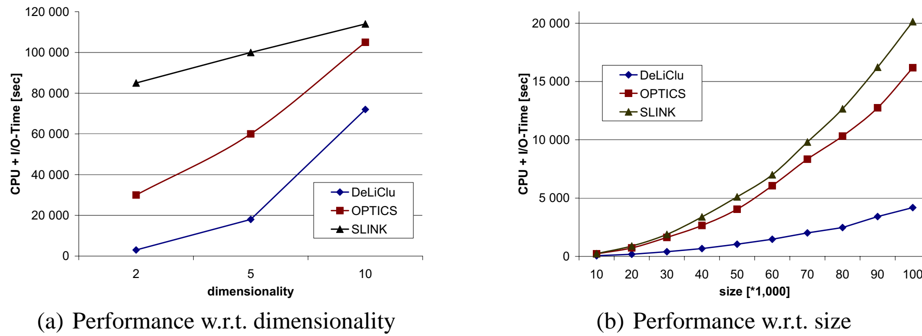
**Fig. 3.** Algorithm DeLiClu

ing $R$, denotes the set of objects already processed and $S$ indicates the set of objects which are still not yet handled. The algorithm starts with an arbitrary chosen start object $s \in S$, migrates $s$ from $S$ to $R$ and writes $s$ with a density distance of infinity to output. Note that migration of $s$ from $S$ to $R$ means, that $s$ is stored in the index structure of $R$ in the same path as in $S$. Thus, we do not need any complex insert or split algorithm upon object migration. The two index structures of $R$ and $S$ only need to have the same structure, i.e. the same directory and data nodes although the tree for $R$ initially contains no point.

The algorithm uses a priority queue into which pairs of nodes and pairs of data objects from $S \times R$ can be inserted. The entries in the priority queue are sorted in ascending order by the distance between the nodes of the pair or the density distance between the objects of the pair. The first pair inserted into the queue is the pair of nodes existing of the root of the index of $S$ and the root of the index of $R$. In each step, the top pair having minimum distance is dequeued from the priority queue. If it is a pair $(n_s, n_r)$ of nodes, the pair will be expanded, i.e. all combinations of the children of $n_s$ with the children of $n_r$ are inserted into the priority queue. Otherwise, if the top pair of the priority queue consists of a pair $(s, r)$ of data objects from $S \times R$, the not yet processed object $s \in S$ is written to output with the density distance $\text{DENDIST}_{minPts}(s, r)$. Afterwards, $s$ is migrated from $S$ to $R$. As described in Section 3.2, objects belonging to already expanded nodes of the path of $s$ have to be reinserted into the priority queue by invoking the algorithm *reinsertExpanded* (see Figure 2). The algorithm terminates if all objects are moved from $S$ to $R$.

## 4 Experimental Evaluation

All experiments have been performed on Linux workstations with two 64-bit 1.8 GHz CPU and 8 GB main memory. We used a disk with a transfer rate of 45 MB/s, a seek time of 4 ms and a latency delay of 2 ms. For either technique a LRU cache of about




(a) Performance w.r.t. dimensionality

(b) Performance w.r.t. size

**Fig. 4.** Performance analysis

50% of the data set size was allocated. The OPTICS algorithm was supported by an R-tree index structure. Unless otherwise specified, the *minPts* parameter of DeLiClu and OPTICS was set to 5. The $\varepsilon$-parameter of OPTICS was set to the optimal value w.r.t. accuracy. Performance is presented in terms of the elapsed time including I/O and CPU-time. Beside synthetic data sets, we used a data set containing 500,000 5D feature-vectors generated from the SEQUOIA benchmark and the El Nino data set from the UCI KDD data repository, containing about 800 9D data objects.

*Performance speed-up.* We first compared the performance of the methods. As it can be seen in Figure 4(a) DeLiClu significantly outperforms OPTICS and SLINK w.r.t. the dimensionality of the database. In Figure 4(b), we can observe that DeLiClu also outperforms SLINK and OPTICS w.r.t. the number of data objects is. Obviously, the speed-up of DeLiClu grows significantly with increasing database size. Similar results can be made on the SEQUOIA benchmark (results are not shown due to space limitations). DeLiClu achieved a speed-up factor of more than 20 over OPTICS and a speed-up factor of more than 50 over SLINK.

*Improvement of accuracy.* The significant effect of parameter $\varepsilon$ on the results of the OPTICS algorithm is shown in Figure 5 (El Nino data). The left part of the figure shows a reachability plot resulting from the new algorithm DeLiClu, the middle part of the figure shows a reachability plot resulting from an OPTICS run with parameter $\varepsilon$ chosen too small. For this experiment, $\varepsilon$ was set to a value for which the runtime of OPTICS was approximately the same as for DeLiClu. Apparently, OPTICS lost a significant part of the whole cluster information due to the wrongly chosen $\varepsilon$. The interpretability of the dendrogram depicted in the right part of the figure is very weak in comparison with the reachability plot resulting from the DeLiClu algorithm. DeLiClu generates strongly separated clusters which cannot be seen in the dendrogram. Similar results have been achieved on the SEQUOIA benchmark.

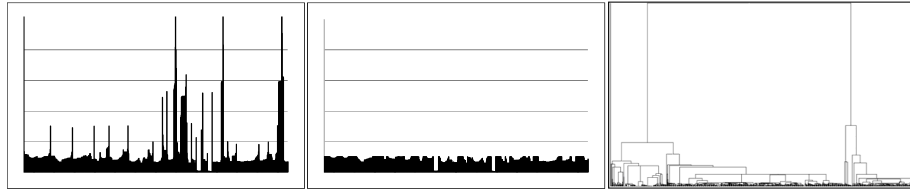


**Fig. 5.** Comparison of accuracy on real-world data set (El Nino data set)

## 5 Conclusions

We proposed the new algorithm DeLiClu based on a novel closest pair ranking algorithm that efficiently computes the hierarchical cluster structure. DeLiClu shows improved robustness over Single-Link w.r.t. noise and avoids the single-link effect by using a density estimator. In contrast to OPTICS it guarantees the complete determination of the cluster structure. It has an improved usability over OPTICS by avoiding the non-intuitive parameter $\varepsilon$. Our experimental evaluation showes that DeLiClu significantly outperforms Single-Link and OPTICS in terms of robustness, completeness, usability and efficiency.

## References


1. Sibson, R.: SLINK: An optimally efficient algorithm for the single-link cluster method. The Computer Journal **16** (1973)
2. Jain, A.K., Dubes, R.C.: Algorithms for Clustering Data. Prentice Hall (1988)
3. Ankerst, M., Breunig, M.M., Kriegel, H.P., Sander, J.: OPTICS: Ordering points to identify the clustering structure. In: Proc. SIGMOD. (1999)
4. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. KDD. (1996)
5. Hjaltason, G.R., Samet, H.: Incremental distance join algorithms for spatial databases. In: Proc. SIGMOD. (1998)
6. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer Verlag (1985)
7. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: Proc. SIGMOD. (1984)
8. Böhm, C., Krebs, F.: The k-nearest neighbor join: Turbo charging the KDD process. KAIS **6** (2004)
9. Xia, C., Lu, H., Ooi, B.C., Hu, J.: GORDER: An efficient method for KNN join processing. In: Proc. VLDB. (2004)
10. Böhm, C., Kriegel, H.P.: A cost model and index archtecture for the similarity join. In: Proc. ICDE. (2001)
11. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: Proc. SIGMOD. (1990)