Clustering Objects on a Spatial Network *

Man Lung Yiu
Department of Computer Science
University of Hong Kong
mlyiu2@csis.hku.hk

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
nikos@csis.hku.hk

Abstract

Clustering is one of the most important analysis tasks in spatial databases. We study the problem of clustering objects, which lie on edges of a large weighted spatial network. The distance between two objects is defined by their shortest path distance over the network. Past algorithms are based on the Euclidean distance and cannot be applied for this setting. We propose variants of partitioning, density-based, and hierarchical methods. Their effectiveness and efficiency is evaluated for collections of objects which appear on real road networks. The results show that our methods can correctly identify clusters and they are scalable for large problems.

1. Introduction

Clustering is one of the most important analysis tasks. The goal is to divide a collection of objects into groups, such that the similarity between objects in the same group is high and objects from different groups are dissimilar. In spatial databases, objects are characterized by their position in the Euclidean space and, naturally, dissimilarity between two objects is defined by their Euclidean distance.

In many real applications, however, the accessibility of spatial objects is constrained by spatial (e.g., road) networks. It is therefore realistic to define the dissimilarity between objects by their *network* distance, instead of the Euclidean distance. The network distance between two objects p and q is defined by the length of the shortest path that reaches q from p and vice versa, assuming an undirected network graph. In addition, the distance between two nodes of the network graph may not be proportional to their Euclidean distance, as it can be characterized by weights, capturing cost due to various conditions (e.g., traffic, rough terrain, elevation, etc.).

Evaluation of spatial queries over spatial networks has already been studied by several researchers [17, 8, 16], recognizing the important role, which the network is playing in search. Secondary memory structures are designed towards efficient processing of queries over spatial networks. Nevertheless there is no previous work on clustering objects that lie on spatial networks. For instance, assume that we want to apply clustering on the set of restaurants that appear in a city map, considering the distance with respect to the city road network. The resulting clusters may identify areas, which can be of interest to touristic location-based service providers or restaurant chains which want to open a new branch in the city.

In this paper, we study the problem and propose several solutions that extend existing clustering paradigms to operate on network data. Our first technique falls into the class of partitioning clustering methods [12]. It randomly selects a set of k medoids and partitions the data into clusters using them. Clusters are refined by iteratively swapping medoids with random points as long as the clustering structure improves. Our second algorithm falls into the class of density-based methods [13], placing points in the same cluster if their distance is smaller than a threshold. Finally, we propose an algorithm that conforms to the hierarchical clustering paradigm. Our contributions can be summarized as follows:

- We define the problem of clustering objects according to their network distance. To our knowledge, this is the first work on this interesting and important problem.
- We propose algorithms that apply dominant clustering paradigms on the network-based clustering problem. Our algorithms are carefully designed to avoid unnecessary distance computations and redundant accesses of the network components.
- We provide an extensive and comprehensive experimental evaluation of our techniques, which demonstrates their effectiveness in the discovery of clusters and their efficiency and scalability for large problems.

The rest of the paper is organized as follows. Section 2 provides background and related work on generic clustering techniques and query processing methods for spatial network data. Section 3 formally defines the problem settings and discusses why existing methods are not appropriate for our problem. In Section 4, we describe a network data storage architecture and the proposed clustering algorithms. Section 5 experimentally evaluates their effectiveness and efficiency. The applicability of the techniques to network-based clustering variants is discussed in Section 6. Finally, Section 7 concludes with a discussion about future work.

*work supported by grants HKU 7380/02E and HKU 7149/03E from Hong Kong RGC

SIGMOD 2004, June 13–18, 2004, Paris, France.

2. Background and Related Work

A large number of clustering algorithms can be found in the literature [6]. The generic definition of clustering is usually refined depending on the type of data to be clustered and the clustering objective. In other words, different clustering paradigms use different definitions and evaluation criteria.

Partitioning methods divide the objects into k groups and iteratively exchange objects between them until the quality of the clusters does not further improve. k-means and k-medoids are representative methods from this class. In k-means algorithms, clusters are represented by a mean value (e.g., a Euclidean centroid of the points in it) and object exchanging stops if the average distance from objects to their cluster's mean value converges to a minimum value. k-medoids algorithms (e.g., PAM, CLARA [12], and CLARANS [15]) represent each cluster by an actual object in it. First, k medoids are chosen randomly from the dataset. An evaluation function sums the distance from all points to their nearest medoid. Then, a medoid is replaced by a random object and the change is committed only if it results to a smaller value of the evaluation function. A local optimum is reached, after a large sequence of unsuccessful replacements. This process is repeated for a number of initial random medoid-sets and the clusters are finalized according to the best local optimum found.

Another class of (agglomerative) *hierarchical* clustering techniques define the clusters in a bottom-up fashion, by first assuming that all objects are individual clusters and gradually merging the closest pair of clusters until a desired number of clusters remain. Several definitions for the distance between clusters exist; the *single-link* approach considers the minimum distance between objects from the two clusters. Others consider the maximum such distance (*complete-link*) or the distance between cluster representatives. *Divisive* hierarchical methods operate in a top-down fashion by iteratively splitting an initial global cluster that contains all objects. The cost of brute-force hierarchical methods is $O(N^2)$, where N is the number of objects, which is prohibitive for practical use. Moreover, they are sensitive to outliers (like partitioning methods). Algorithms like BIRCH [19] and CURE [5] were proposed to improve the scalability of agglomerative clustering and the quality of the discovered partitions. C2P [14] is another hierarchical algorithm similar to CURE, which employs closest pairs algorithms and uses a spatial index to improve scalability.

Density-based methods discover dense regions in space, where objects are close to each other and separate them from regions of low density. DBSCAN [13] is the most representative method in this class. First, DBSCAN selects a point p from the dataset. A range query, with center p and radius ϵ is applied to verify if the neighborhood of p contains at least a number $MinPts$ of points (i.e., it is dense). If so, these points are put in the same cluster as p and this process is iteratively applied again for the new points of the cluster. DBSCAN continues until the cluster cannot be further expanded; the whole dense region where p falls is discovered. The process is repeated for unvisited points until all clusters and outlier points have been discovered. A limitation of this approach (alleviated in [2]) is that it is hard to find appropriate values for ϵ and $MinPts$.

There is limited work on clustering the nodes of a weighted graph. An agglomerative hierarchical approach [7] treats each node as a cluster and then merges the clusters until one remains. The single-link variant of this method has complexity $O(|V|^2)$, whereas the complete-link variant comes with complexity $O(|V|^2 \log |V|)$. Both methods are not scalable for large networks. Another variant [18] applies divisive clustering on the minimum spanning tree of the graph, which can be computed in $O(|V| \log |V|)$ time. However, this method is very sensitive to outliers. CHAMELEON [10] is a general-purpose algorithm, which transforms the problem space into a weighted k-NN graph, where each object is connected with its k nearest neighbors. The weight of each edge reflects the similarity between the objects. A *graph-partitioning* technique [11] is then used to partition the k-NN graph into small, dense subgraphs. The subgraphs are hierarchically merged according to their *interconnectivity* (defined by the total weight of the edges connecting points from the two clusters) and their *closeness* (defined by the average distance between points from the two clusters). This algorithm can produce results of high quality, however, the computation of the measures used for clustering have quadratic cost, and the algorithm is not scalable to large problems.

There is also related work on query processing over spatial networks. The main focus of database research is how to organize large sparse networks (e.g., road networks) on disk, such that shortest path queries can be efficiently processed. The shortest path between two network nodes can be computed by Dijkstra's algorithm [4]. This technique starts from the source node n_s and organizes its adjacent nodes in a priority queue (i.e., heap) based on their distance from the source. The closest node n_c from n_s is iteratively removed from the heap and its adjacent nodes (with their distances from n_s via n_c) are added on the heap. The process continues until the destination node n_d (with the shortest path distance) is popped from the heap. An advantage of Dijkstra's algorithms is that each adjacency list is visited at most once. The worst-case complexity of this method is $O(|E| \log |E|)$, which is reduced to $O(|V| \log |V|)$ for planar graphs due to Euler's formula ($|E| \leq 3|V| - 6$). Most spatial network graphs are planar, or even if not planar they are sparse enough for this complexity to hold.

CCAM [17] is a disk-based storage architecture that aims at the minimization of I/Os during shortest path computations. Network nodes with their *adjacency lists* (i.e., outcoming edges with their weights) are grouped into disk pages, based on their connectivity and how frequently they are accessed together; neighbor nodes are placed in the same page with high probability. Hierarchical indexes that group disk pages, storing their summaries together with materialized shortest path precomputations were proposed in [1, 9, 8]. Finally, evaluation of nearest neighbor queries, range queries, and distance joins using the network distance, for objects lying on spatial networks was recently studied in [16]. The algorithms proposed there are extensions of Dijkstra's shortest path that utilize Euclidean distance bounds to accelerate search.

3. Network-based Clustering

In this section, we formally define the problem space on which we apply clustering and the distance metric used in our settings. We then identify the peculiarities of the problem and discuss why existing clustering algorithms are inapplicable or inefficient for objects that lie on a network.

3.1 Definitions

DEFINITION 1. *A* **network** *is an undirected weighted graph $G = (V, E, W)$ where V is the set of vertices (i.e., nodes), E is the set of edges, and $W : E \rightarrow \mathbb{R}^+$ associates each edge to a positive real number. An* **object** *(i.e., point) is located on an edge $e \in E$ in the network. The position of the object in the network can be expressed by the triplet $\langle n_i, n_j, pos\rangle$ where $pos \in [0, W(e)]$ is the distance of the point from node n_i along the edge. To ensure the position of the object is expressed unambiguously by one triplet, we require that $n_i < n_j$ (assuming a total ordering of node labels).*

Figure 1 shows an example of a network. Nodes are denoted by squares and each node has a label. The lines represent edges and each edge is associated with a distance label. Objects are denoted by crosses. A point lies on exactly one edge.[1] For instance, p_2 lies on the edge (n_1, n_3) and it is 1.0 units away from n_1 along the edge. Therefore, its position can be expressed by $\langle n_1, n_3, 1.0\rangle$.

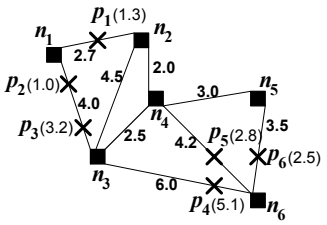


Figure 1: A network

DEFINITION 2. *Let p and q be two points, whose positions are $\langle n_a, n_b, pos_p\rangle$ and $\langle n_c, n_d, pos_q\rangle$, respectively. The* **direct distance** *$d_L(p, q)$ between points p and q is defined by $|pos_p - pos_q|$ if $n_a = n_c$ and $n_b = n_d$ (i.e., p and q lie on the same edge); otherwise, it is defined as ∞. Given a point p with position $\langle n_a, n_b, pos_p\rangle$, the* **direct distance** *$d_L(p, n_a)$ between p and n_a is pos_p. The direct distance $d_L(p, n_b)$ is defined by $W(n_a, n_b) - pos_p$.*

For instance, in Figure 1, $d_L(p_2, p_3) = 2.2$, since the points lie on the same edge. On the other hand, $d_L(p_2, p_1) = \infty$. The direct distance of a point from a network node is defined only when the point is lying on an edge adjacent to the node. In simple words, it is the network distance between the point and the node based on the weight of the corresponding edge. Thus, $d_L(p_1, n_1) = 1.3$ and $d_L(p_1, n_2) = 2.7 - 1.3 = 1.4$. The direct distance can be found in constant time. Notice that the direct distance of two points on the same edge is not necessarily the shortest distance between them. Finally, notice that the direct distance is symmetric, i.e., $d_L(p, q) = d_L(q, p)$ and $d_L(p, n_i) = d_L(n_i, p)$.

DEFINITION 3. *Given nodes n_i and n_j, the* **network distance** *$d(n_i, n_j)$ is defined as the distance of the shortest path from n_i to n_j and vice versa.*

The shortest path from n_i to n_j is the one for which the sum of weights on the edges is the minimum. In Figure 1, the network distance between n_2 and n_6 is 6.2.

[1]In real-life problems, some objects may not lie on edges of the network. In such cases, we assume that the object is represented by the position on the network which is most directly accessible from it, as in [16].

DEFINITION 4. *Given points p and q, where p lies on edge (n_a, n_b) and q lies on the edge (n_c, n_d), the* **network distance** *$d(p, q)$ is the distance of the shortest path from p to q. $d(p, q)$ is defined by $\min_{x\in\{a,b\},y\in\{c,d\}}(d_L(p, n_x)+d(n_x, n_y)+ d_L(n_y, q))$, if p and q lie on different edges; otherwise, $d(p, q)$ is the minimum of the previous quantity and $d_L(p, q)$.*

The network distance (like the direct distance) is symmetric. The inequality $d(p, s) \leq d(p, q) + d(q, s)$ holds for any p, q, s, because $d(p, s)$ is the shortest distance between p and s. Therefore, the network distance is a metric. Given a collection of N points that lie on a network, our objective is to group them into a set of clusters, according to some clustering criteria. Since different clustering paradigms (i.e., partitioning, density-based, hierarchical) use different cluster definitions and quality measures, we will elaborate them when we present our network-based clustering algorithms for each paradigm.

3.2 Application of existing methods

Straightforward application of the clustering methods reviewed in Section 2 is either infeasible or inefficient for our problem. The replacement of the Euclidean distance by the network distance increases the complexity, since now the distance between two arbitrary objects cannot be computed in constant time, but an expensive shortest path algorithm is required. Another problem is that we want not to cluster network nodes, but objects which lie on arbitrary locations on the graph edges. Thus, even the shortest path definition is adapted for this case (see Definition 4).

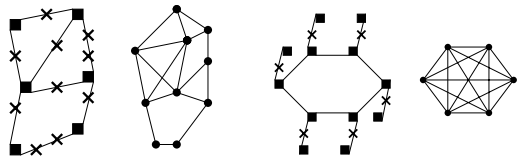
One possible method to alleviate the problem is to precompute the distance between every pair of network nodes and store it in a 2D matrix of size $O(|V|^2)$. Dijkstra's algorithm can be applied for each node to compute its shortest path distances to every other node. The total time complexity is $O(|V|^2 \log |V|)$ (for planar graphs). With this matrix, the distance between any points can be found in constant time using Definition 4. Then, existing clustering algorithms (e.g., k-medoids) could be used to cluster the points. Nevertheless the time complexity of this method is high for large graphs. In addition, this matrix could be prohibitively large to store. For instance, if the graph has $|V|$=100K nodes, we need to store $|V|(|V| - 1)/2 \cong 5 \times 10^9$ distances, which are expensive to manage and retrieve.[2]

Another problem which makes recent advanced clustering techniques inapplicable is that concepts like cluster centroids and summaries, used by several clustering methods (e.g., BIRCH, CURE, and C2P) cannot be defined in our context. Clearly, given a group of objects on a network, we cannot define their centroid using the network distance, since there may not be a unique network point whose average distance from the points in the group is the minimum. Even if there is such a point, finding it could be very expensive. In addition, we cannot select cluster representatives

[2]A similar method could be applied to compute the distance between every pair of *objects* by applying Dijkstra's algorithm for the points (not the network nodes). This approach would be beneficial if the number N of objects is much smaller than the number $|V|$ of network nodes. In this case, the distance matrix has size $O(N^2)$ and it can be computed in $O(N \cdot |V| \log |V|)$ time. However, in this paper we are interested in problems where both $|V|$ and N are large.

that are not actual points in the cluster (e.g., by "shrinking" medoids, as in CURE and C2P). On the other hand, it is possible to approximate a cluster using actual representative points from it (i.e., actual medoids).

Finally, a potential solution would be to transform the weighted graph G to a new graph G', where each node n_p in G' is an object p from the original network G and there is an edge (n_p, n_q) in G', if there is a path from p to q in G not passing via any other object s. The weight of this edge corresponds to the length of the (shortest) path between p and q. Figure 2a illustrates such a transformation; the original network is shown on the left and the transformed one on the right. Weights are omitted for simplicity. After the transformation, a graph-based clustering algorithm could be used for clustering. However, this method has several limitations. First, the transformation from G to G' is quite expensive requiring many shortest path computations. Second, the transformed graph may no longer be planar and it can contain complex components, increasing the complexity of distance computations. For instance the ring on the left of Figure 2b translates to a clique. In general, we expect that the transformation will increase the complexity of the graph making it harder to search, as exemplified in both cases of Figure 2. Finally, existing clustering methods for weighted graphs (e.g., [10]) do not scale well, as discussed in Section 2.



(a) a simple network (b) a ring network

Figure 2: Transforming of networks with objects to conventional weighted graphs

4. Network-based Clustering Techniques

In this section, we present a storage architecture for the network graph and the objects that lie on it. Then, we propose partitioning, density-based, and hierarchical algorithms that apply on this scheme to cluster the objects efficiently.

4.1 Disk-based storage of the network

The core module of a clustering algorithm on a graph is the shortest path computation between any pair of network nodes, since this is essential for determining the network distance. In real-life problems, the size of the network (as well as the number of objects to be clustered) can be large, therefore we need efficient disk-based representations for them.

We use a disk-based storage model that groups network nodes based on their connectivity and distance, as suggested in [17]. A graphical illustration of our files and indexes for the network of Figure 1, is shown in Figure 3. Our model allows efficient access of the adjacency lists and points. The adjacency list and the points are stored in two separate flat files. To facilitate efficient access, these flat files are then indexed by B$^+$–trees.

The point-IDs are assigned in such a way that for the points on the same edge, IDs are sequential and their position offsets are in ascending order. One clustering operation

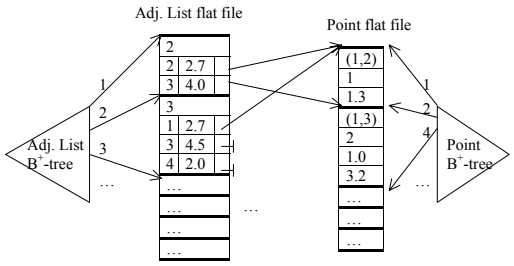


Figure 3: Disk-based storage representation

may ask for the edge a particular point lies on. To support this query efficiently, we store the points (position offsets) on the same edge in a *point group*. At the beginning of the point group, we also store the edge and number of points on the edge. After that, a sparse B$^+$–tree is built on this flat file, with keys the first point ID of each points group. For example, consider the network of Figure 1; p_2 and p_3 are in the same group (i.e., same edge) and their position offsets 1.0 and 3.2 are stored in the points flat file. The edge $(1, 3)$ and the number of points on the edge 2 are stored at the beginning of the point group. In a leaf node entry of the points B$^+$–tree, the key 2 points to the corresponding point group in the flat file.

The number of adjacent nodes of node n_i is stored at the beginning of its adjacency list. For each adjacent node n_j of the node n_i, we store the adjacent node ID, the weight of the edge (n_i, n_j) and a reference pointer to its point group (i.e., the ID of the first point in the group). If the edge does not have any point group, a $NULL$ pointer is stored. A B$^+$–tree is built on the adjacency list file. For this tree, the key is the node ID and the value is a pointer to its adjacency list. For the example of Figure 1, the adjacent nodes of n_1 are n_2 and n_3 so we store n_2 at the beginning of the adjacency list. After that, we store the detail information about edges $(1, 2)$ and $(1, 3)$ in the points file.

This storage model is simple and supports the required operations by our clustering algorithms. The adjacency list module is similar to that used by CCAM [17]. However, we also use the points file, which is absent in CCAM (CCAM indexes network nodes only). The model in [16] is essentially more complex than ours, since it stores the spatial location of the indexed objects in the same representation and indexes them using an R–tree. We do not use this model here. First, we assume no dependencies between Euclidean and network distance. Second, our algorithms need not access the objects and network nodes based on their spatial location. Nevertheless our algorithms can be applied on that representation, as well.

4.2 Partitioning-based clustering

In this section, we propose a partitioning algorithm for our problem. As discussed, the centroid for a group of objects that lie on a network cannot be defined or computed easily, thus, we explore the application of a k-medoids algorithm.

Initially, a set of k points (medoids) is randomly selected from the data. Each point is then assigned to the cluster corresponding to the nearest medoid reachable from it. While partitioning, an evaluation function is applied to compute the quality of the partition. Let m_i be a medoid and

C_i its corresponding cluster for $i \in [1,k]$. A cluster is good if the points in the cluster are near to the medoid. The evaluation function for a set of clusters is defined as $R(\{(C_i, m_i) : i \in [1,k]\}) = \sum_{i\in[1,k]} \sum_{p\in C_i} d(p, m_i)$. The lower the value of R, the better the corresponding partitioning is. After a partitioning is evaluated, a medoid is replaced by a random point from the dataset and the points are reassigned to clusters based on the new medoid-set. If the R value of the new partitioning is better than the one before the replacement, the change is committed. Otherwise, the change is rolled-back and a new medoid replacement is attempted. If for a given set of medoids, a number of changes does not lead to a better clustering, the process ends (a local optimum has been reached) and the medoids and clusters are finalized.

4.2.1 *Finding the nearest medoid for each node*

One of the main issues of k-medoids is how to assign the points to clusters fast. As discussed in Section 3.2, it is too expensive to precompute and store the distances between every pair of network nodes (or objects). Thus, the point assignment has to be performed dynamically for every medoid.

Let M be the set of medoids. Given a network node n_i, assume that m^{n_i} is the nearest medoid in M reachable from n_i. Given a point p, lying on edge (n_x, n_y), the distance from p to its closest medoid can be found by:

$$d(m^p, p) = \min \{ d(m^{n_x}, n_i) + d_L(n_x, p), \; d(m^{n_y}, n_y) + d_L(n_y, p), \; \min_{i\in[1,k]} d_L(m_i, p)\} \quad (1)$$

In other words, m^p is the closest medoid reachable via n_x, or the closest medoid reachable via n_y, or m^p lies on the same edge as p and its direct distance from p is the smallest compared to the previous two quantities. Since the direct distances can be computed efficiently, the problem is reduced to finding the distance from each node to its nearest medoid. The algorithm shown in Figure 4 performs exactly this. It *concurrently* traverses the network around all medoids assigning nodes to their nearest medoids. First, a priority queue Q is initialized. Each entry B of the queue has a node-ID $B.node$, a medoid-ID $B.med$, and the shortest path distance from $B.med$ to $B.node$. Initially, all nodes from the edges, where the medoids lie are enqueued. Then, the function **Concurrent_Expansion** is called, which applies Dijkstra's algorithm concurrently for all medoids. If a node has been dequeued before (line 3), we know that it has been already assigned to some medoid with a smaller distance than the current (guaranteed by the priority queue), and therefore we need not change the information about it or traverse the graph again from it. Otherwise, its unassigned neighbors n_z with their distance from the currently dequeued medoid are enqueued. At the end of the algorithm, all nodes have been tagged with information about their nearest medoids and their distances from them. The points are then scanned and assigned to clusters according to Equation 1, computing the evaluation function for the current partitioning at the same time.

```
Algorithm Medoid_Dist_Find(medoids M)
1    for each node n_i, assigned[i]:=false;
2    Q:=new priority queue;
3    for each medoid m_i ∈ M
4        let m_i lie on the edge (n_x, n_y);
5        create new queue entries B_x, B_y;
6        B_x.node:=n_x; B_x.med:=m_i; B_x.dist:=d_L(m_i, n_x);
7        B_y.node:=n_y; B_y.med:=m_i; B_y.dist:=d_L(m_i, n_y);
8        enqueue(Q, B_x); enqueue(Q, B_y);
9    Concurrent_Expansion(Q);

Function Concurrent_Expansion(queue Q)
1    while (notempty(Q))
2        B:=dequeue(Q);
3        if (not assigned[B.node]) then
4            m^{B.node}:=B.med;
5            d(m^{B.node}, B.node):=B.dist;
6            assigned[B.node]:=true;
7            for each adjacent node n_z of B.node
8                if (not assigned[n_z]) then
9                    create a new entry B';
10                   B'.node:=n_z; B'.med:=B.med;
11                   B'.dist:=B.dist + W(e(n_z, B.node));
12                   enqueue(Q, B');
```

Figure 4: Finding nearest medoid for each node

4.2.2 *Incremental replacement of node distances*

The k-medoids method only replaces one medoid at each iteration. The nearest medoid information of many nodes may be the same as that from the previous iteration. Instead of running the algorithm of Figure 4 for every iteration, we can incrementally update the nearest medoids of nodes using the algorithm of Figure 5. First, all nodes that were assigned to the replaced medoid *oldm* are unassigned. All neighbors of those nodes that belong to some other medoid are enqueued in a queue Q. Then, the nodes which define the edge where the new medoid *newm* lies are also enqueued. Finally a function **Concurrent_Expansion**(Q) is invoked. This function is the same as the one of Figure 4, except from two changes; lines 3 and 8 change to the following:

```
...
3 if (not assigned[B.node]) or (B.dist < d(m^{B.node}, B.node))) then
...
8 if (not assigned[n_z]) or
        (B.dist + W(e(n_z, B.node)) < d(m^{n_z}, n_z))) then
...
```

These changes capture the cases where the nearest medoid for a node has to be updated because it might be closer to the new medoid *newm*. In general, this method is better than finding the nearest medoid distance for each node from scratch every time there is a change. However, it needs to access information about the previous medoids of nodes. For each network node, information about its nearest medoid and distance from it can be stored and incrementally updated on the disk-based structure, if too large to store in memory. The performance savings of this method in comparison to running the one of Figure 4 for every iteration, are expected to increase with k, since the larger k is, the smaller the fraction of information that changes after a medoid replacement.

4.3 Density-based clustering

The convergence (and cost) of k-medoids depends on the choice of the medoids and cannot be estimated a priori. If

```
Algorithm Inc_Medoid_Update(medoids oldm, newm, M)
1   Q:=new priority queue;
2   for each n_i ∈ V such that m^{n_i} = oldm
3     m^{n_i}:=null; //reset nearest medoid info
5     for each adjacent node n_z of node n_i
6     if (m^{n_z} ∈ {M − oldm}) then
7       Create a new entry B;
8       B.node:=n_i; B.med:=m^{n_z};
9       B.dist:=d(m^{n_z}, n_z) + W(e(n_z, n_i));
10      enqueue(Q, B);
11  let newm lie on the edge (n_x, n_y);
12  create new entries B_x, B_y;
14  B_x.node:=n_x; B_x.med:=newm; B_x.dist:=d_L(newm, n_x);
15  B_y.node:=n_y; B_y.med:=newm; B_y.dist:=d_L(newm, n_y);
16    enqueue(Q, B_x); enqueue(Q, B_y);
17  Concurrent_Expansion(Q);
```

Figure 5: Incremental update of nearest medoids

the local optimum is far from the initial medoids assignment, many iterations are required and the cost of the algorithm is high. Another problem of k-medoids is that outliers are included in the clusters, since all points are essentially grouped. Outliers may degenerate the structure of a cluster and affect the evaluation function. Density based clustering algorithms [13] cluster points based on their local density and can handle outliers in low-density regions. In this section, we apply the density-based paradigm to cluster objects, located on a spatial network.

DBSCAN is a density-based method [13], which given a random point p, identifies the cluster, where p belongs, by applying an ϵ-range query around p and checking if there are at least $MinPts$ points in this range. If so, a new cluster for p is created containing the points in the range query. DBSCAN iteratively applies range queries for the new points in the cluster, until it cannot be expanded any further. We can directly apply this method on our network model. A main module of the algorithm finds the ϵ-neighborhood of a point p in the network. This can be done by expanding the network around p and assign points until the distance exceeds ϵ (a similar range search algorithm was proposed in [16]).

Essentially, a range search query has to be performed for every object p, which may be expensive if the number of points is large. In the next paragraph, we propose a fast technique that works for the case where $MinPts = 2$; i.e., the sufficient condition that two points are placed in the same cluster is that their distance is at most ϵ.

4.3.1 *An ϵ-Link algorithm*

Our density-based clustering algorithm iteratively picks unclustered points from the dataset and runs the **ϵ-Link** algorithm, shown in Figure 6, to discover the cluster which contains them. Clustering terminates if there are no more unassigned points. Given a point m, the idea is to browse the network using Dijkstra's algorithm, however, the shortest path for every node now changes dynamically as new points are assigned in the cluster. The algorithm starts by visiting the edge where m lies and traversing it in both directions, inserting into the current cluster C the visited points as long as their direct distance from the previous point is at most ϵ (lines 6–9). If the nodes n_x and n_y are within ϵ distance from the last clustered point in their direction, they are inserted into a queue Q together with their distances (lines 10–11).

Consider the example of Figure 7 and assume that $\epsilon = 5$. Assume also that we start from point m on the edge (n_1, n_2). This edge is traversed first from m in the direction towards n_1. The first point found there is p_1 having distance $6 > \epsilon$ from m, thus this and the next points (if any) in this direction are not inserted to C. Node n_1 is also not enqueued in Q, since its distance from the last clustered point (i.e., m) is greater than ϵ. Then the edge is traversed from m towards n_2. In this case, p_2 and p_3 are inserted to the cluster. Also n_2 is enqueued since its distance $d_L(n_2, p_3) = 4$ from the last clustered point (i.e., p_3) is smaller than ϵ.

```
Algorithm ε–Link(ε, point m, cluster C)
1   for each n_i ∈ V NNdist[n_i]:=∞;
2   Q:=new priority queue; C:={m}; clustered[m]:=true;
3   find edge (n_x, n_y) where m lies;
4   create new entries B_x, B_y;
5   p:=m; nextp:=next point on (n_x, n_y) from m to n_x;
6   while (nextp ≠ null) and (not clustered[nextp])
          and (d_L(p, nextp) ≤ ε)
7     p:=nextp; C:=C ∪ {p}; clustered[p]:=true;
8     nextp:=next point on (n_x, n_y) from m to n_x;
9   B_x.node:=n_x; B_x.dist:=d_L(p, n_x);
10  if (B_x.dist ≤ ε)) then enqueue(Q, B_x);
11  apply lines 6–11 for n_y as well;
12  while (notempty(Q))
13    B:=dequeue(Q);
14    if (B.dist < NNdist[B.node])) then
15      NNDist[B.node]:=B.dist;
16      for each node n_z adjacent to B.node
17        p:=B.node; nextp:=next pt from B.node to n_z;
18        if (nextp ≠ null) and (not clustered[nextp])
                and ((d_L(nextp, B.node) + B.dist) ≤ ε)
19          newd_{B.node}:=d_L(nextp, B.node);
20          newd_{n_z}:=d_L(nextp, n_z);
21          p:=nextp; C:=C ∪ {p}; clustered[p]:=true;
22          nextp:=next point from B.node to n_z;
23          while (nextp ≠ null)
                  and (not clustered[nextp])
                  and (d_L(p, nextp) ≤ ε)
24            newd_{n_z}:=d_L(nextp, n_z);
25            p:=nextp; C:=C ∪ {p};
26            clustered[p]:=true;
27            nextp:=next point from B.node to n_z;
28          if (NNDist[B.node] > newd_{B.node}) then
29            B_new.node:=B.node;
30            B_new.dist:=newd_{B.node};
31            enqueue(Q, B_new);
32        if (no points on edge (B.node, n_z)) then
33          newd_{n_z}:=B.dist + W(n_z, B.node)};
34          if (NNDist[n_z] > newd_{n_z})
                    and (newd_{n_z} ≤ ε) then
35            B_new.node:=n_z;
36            B_new.dist:=newd_{n_z};
37            enqueue(Q, B_new);
```

Figure 6: The ϵ–Link clustering algorithm

While Q is not empty, the first node (e.g., n_2 in Figure 7) is dequeued from it. The condition in line 14 ensures that the node's distance from C has changed since the last time it was dequeued. Only then the edges adjacent to it are visited and traversed to possibly insert points in the cluster. In our example, assume that (n_2, n_3) is the first such edge to be examined. The points on the edge (if any) are traversed (lines 18–27), checking whether there are at most ϵ away from the nearest point to the cluster so far (captured by B or the last clustered point on that edge). In our running example, p_4 will not be clustered since its known distance from the cluster is $d_L(B.node, p_4) + B.dist = 7$, where $B.node = n_2$ and $B.dist = 4$. In this case, no new node will be enqueued since we know that (i) the distance

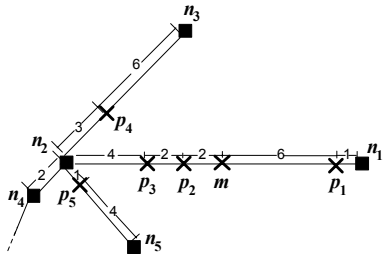


Figure 7: Example of ϵ-Link clustering

of n_2 to the cluster has not been changed and (ii) there are some points on (n_2, n_3), which are not clustered, thus n_3 cannot be closer than ϵ from the cluster. The next neighbor n_4 of n_2 is now visited. In this case, we can find no points on edge (n_2, n_4). However, we check (lines 33–34) the distance of n_4 from the cluster (via n_2); if this distance is smaller than ϵ we'll have to enqueue n_4, since some object may be reachable via it. Otherwise, like in our example, n_4 is not enqueued (its distance is 6). Next, we move to edge (n_2, n_5). In this case, the first point there (p_5) has distance $5 \leq \epsilon$ and therefore it is clustered. After this, (i) we enqueue n_2 again, since its distance from the cluster has decreased (it is now $d_L(n_2, p_5) = 1$) and (ii) we enqueue n_5, since its distance from the cluster is $d_L(n_5, p_5) = 4$. Next, n_2 is dequeued again (this time $B.dist = 1$) and its adjacent edges are re-visited. The reader will notice that now (i) p_4 is clustered (but n_3 is not enqueued) and (ii) n_4 is enqueued with distance 3. The algorithm continues by dequeuing n_4 and checking its neighbor edges until the cluster is discovered.

The user can specify an optional parameter *min_sup* so that clusters with fewer than *min_sup* points are declared as outliers. Observe that the algorithm does not necessarily traverse the whole network, but only the edges which contain the points or are within ϵ distance from some point. Its worst case complexity is at most one graph traversal (cost of each iteration by k-medoids). Therefore, this algorithm is much faster than k-medoids. Also, its cost is deterministic for a given problem and parameters, as opposed to k-medoids, which is a randomized algorithm. The memory requirements is the $NNdist$ array used to store the current distance of each network from the cluster, and a bitmap (*clustered*) that indicates whether a point has been clustered before. If $NNdist$ cannot fit in memory, we can store/update this information for each node into the disk-based structure described in Section 4.1. An appropriate value for ϵ may be hard to determine a priori. A possible way to solve this problem is to use a value determined by the user's experience, or by sampling on the network edges.

4.4 Hierarchical clustering

In this section, we propose a hierarchical algorithm, which starts by considering each point as a cluster, and iteratively merges the closest pair of clusters with a single graph traversal. We then discuss methods that make this algorithm scalable for large problems.

4.4.1 *A Single-Link algorithm*

Like most hierarchical methods, our algorithm initially assumes that each point is a cluster. It then iteratively merges the closest pair of clusters until one cluster remains. The user may opt to stop the algorithm after a desired number of k clusters have been discovered. Clusters are merged according to the *single-link* approach.

Figure 8 shows a pseudocode for the algorithm. During the first (*initialization*) phase (lines 3–21), all edges of the network are accessed and each point p on them (if any) is inserted to a separate cluster C_p. In addition, all pairs of directly connected clusters (i.e., consecutive points on the same edge) are inserted into a priority queue P together with their distance. Consider for example, edge (n_1, n_2) in the network of Figure 9. Points p_1, p_2, p_3 form clusters C_{p_1}, C_{p_2}, C_{p_3}, respectively, and entries $\langle C_{p_1}, C_{p_2}, 1\rangle$, $\langle C_{p_2}, C_{p_3}, 2\rangle$ are inserted into P. For every edge (n_x, n_y), let p be the point on the edge closest to n_x and q the point on the edge closest to n_y. Entries $\langle n_x, d_L(n_x, p), C_p\rangle$ and $\langle n_y, d_L(n_y, q), C_q\rangle$ are inserted into a hash table T organized on node-ID (lines 6 and 13).[3] For our example of Figure 9 and for edge (n_1, n_2), entries $\langle n_1, 5, C_{p_1}\rangle$, $\langle n_2, 4, C_{p_3}\rangle$, are inserted into T.

Then, for each node n_i in T we update information about its nearest adjacent cluster and enqueue it into a priority queue Q (lines 14–19). For our running example and for node n_2, $\langle n_2, 2, C_{p_5}\rangle$ will be enqueued in Q, since the nearest cluster to n_2 is C_{p_5}. Moreover, if there are more than one tuples for n_i in T, this means that n_i is adjacent to multiple clusters and we can compute the distances between these clusters via n_i. We insert into P all pairs consisting of the nearest adjacent cluster to n_i and any other cluster adjacent to n_i (lines 20–21). We need not add more pairs, since no pair of clusters can be merged unless it contains the nearest cluster. For instance, there are three entries for n_2 in T, since n_2 is adjacent to three clusters. Thus, we insert to P entries $\langle C_{p_5}, C_{p_3}, 6\rangle$ and $\langle C_{p_5}, C_{p_4}, 5\rangle$. We need not insert entry $\langle C_{p_3}, C_{p_4}, 7\rangle$, since C_{p_3} cannot be merged with C_{p_4} via n_2, unless C_{p_5} and C_{p_4} are merged first (they are the closest pair).

During the second *expansion* phase (lines 22–44), nodes are dequeued from Q and the distance to their nearest cluster is doubled and compared with the top element in P. If it is larger, this means that we can find no closer pair of clusters to merge than the top element of P. Thus, we merge clusters until it is possible to find a closest merging via the currently dequeued node $B.node$ from Q (lines 25–26). In our running example, the first dequeued element from Q is $\langle n_4, 1, C_{p_6}\rangle$ and the top element of P is $\langle C_{p_1}, C_{p_2}, 1\rangle$. Since $2 \times B.dist \geq 1$, we dequeue the top pair of clusters (C_{p_1}, C_{p_2}) from P and merge them. Similarly, we merge $\{C_{p_1}, C_{p_2}\}$ with C_{p_3} (their distance is 2 in the next top pair of P). The next top element in P is now $\langle C_{p_5}, C_{p_4}, 5\rangle$, which violates the while condition of line 25. Since the top node in Q (i.e., n_4) has not been dequeued before (we keep a bitmap array *expanded* to verify this), we expand the network from n_4 and we visit all its neighbors n_z.[4] There are three cases; in the first one, n_z has been reached from the same cluster before. In this case, if it can be reached faster via $B.node$,

[3] Using a hash table T could be avoided if for every node n_x we locate all adjacent points directly by accessing its adjacent edges. However, this causes many random accesses to the points file, whereas now we only perform a single scan on this file.

[4] If $B.node$ had been expanded before, then we know that $B.node$ has been visited before from a closer cluster, thus we ignore it.

```
Algorithm Single–Link(dendrogram C)
1  Q:=new priority queue; P:=new priority queue;
2  T:=new hash table with entries ⟨NID, dist, CID⟩
       and hash-key NID;
3  for each network edge (n_x, n_y) with points on it
4    p_s:=first point on (n_x, n_y);
5    C_{p_s}:={p_s}; //define new cluster;
6    insert into T row ⟨n_x, d_L(n_x, p_s), C_{p_s}⟩;
7    p:=p_s; nextp:=next point on (n_x, n_y) from p to n_y;
8    while (nextp ≠ null)
9      C_{nextp}:={nextp};
10     enqueue(P, ⟨C_p, C_{nextp}, d_L(p, nextp)⟩);
11     p:=nextp;
12     nextp:=next point on (n_x, n_y) from p to n_y;
13   insert into T row ⟨n_y, d_L(n_y, p), C_p⟩;
14 for each n_i ∈ V NNclus[i]:=null;
15 for each group of tuples T_i ∈ T with same NID, ordered by dist
16   t_{i1}:=first tuple in T_i;
17   NNclus[t_{i1}.NID]:=t_{i1}.CID;
18   NNdist[t_{i1}.NID]:=t_{i1}.dist;
19   enqueue(Q, ⟨t_{i1}.NID, t_{i1}.dist, t_{i1}.CID⟩);
20   for each tuple t_{ij} ∈ T_i, j > 1
21     enqueue(P, ⟨t_{i1}.CID, t_{ij}.CID, t_{i1}.dist + t_{ij}.dist⟩);
22 for each n_i ∈ V expanded[i]:=false;
23 while there are still > 1 clusters
24   B:=dequeue(Q);
25   while (2 × B.dist ≥ P.top.dist)
         and there are still > 1 clusters
26     dequeue and merge top pair of clusters in P;
27   if (not expanded[B.node])
         and there are still > 1 clusters then
28     expanded[B.node]:=true;
29     for each node n_z adjacent to B.node
30       if (NNclus[B.node] = NNclus[n_z]) then
31         if (NNdist[B.node] + W(B.node, n_z) < NNclus[n_z])then
32           NNdist[n_z]:=NNdist[B.node] + W(B.node, n_z);
33           enqueue(Q, ⟨n_z, NNdist[n_z], NNclus[n_z]⟩);
34       else if NNclus[n_z] ≠ null then
35         enqueue(P, ⟨NNclus[B.node], NNclus[n_z],
36         NNdist[B.node] + NNdist[n_z] + W(B.node, n_z)⟩);
37         if (not expanded[n_z]) and
               (NNdist[B.node] + W(B.node, n_z) < NNclus[n_z]) then
38           NNclus[n_z]:=NNclus[B.node];
39           NNdist[n_z]:=NNdist[B.node] + W(B.node, n_z);
40           enqueue(Q, ⟨n_z, NNdist[n_z], NNclus[n_z]⟩);
41       else // NNclus[n_z] = null
42         NNclus[n_z]:=NNclus[B.node];
43         NNdist[n_z]:=NNdist[B.node] + W(B.node, n_z);
44         enqueue(Q, ⟨n_z, NNdist[n_z], NNclus[n_z]⟩);
```

Figure 8: The Single–Link clustering algorithm

we update its distance from the cluster and enqueue it in Q (lines 32–33). In the second case, n_z has been last reached from another cluster. In this case, we have discovered a path connecting the two clusters. The cluster pair is enqueued in P and if n_z was not expanded before (i.e., it might be expanded in the future) and it can be reached faster via $B.node$ we update its distance from the cluster and enqueue it in Q (lines 35–40). In the last case, it is the first time n_z is visited by the algorithm. Its distance to the nearest cluster via $B.node$ is updated and n_z is inserted into Q (lines 42–44).

In our example, and for the neighbor n_3 of the currently dequeued node n_4 we find that n_3 was last reached by cluster C_{p_4}. Thus we add on the heap P the cluster-pair (C_{p_6}, C_{p_4}) with distance $1+2+1=4$. Since the distance from n_3 to its nearest cluster did not change, we needn't do anything else. The next node to be dequeued from Q is n_2 (with distance 2 from C_{p_5}), which causes the top pair (C_{p_6}, C_{p_4}) of P to be merged. The algorithm continues this way and terminates if all clusters are merged[5] (or if k clusters remain, given a parameter k). It correctly computes the dendrogram $\mathcal{C}$ of merged clusters due to the logic of the priority queues P and Q; no pair from P is merged if it is possible to find a closer pair via the top node of Q. Its correctness is not proven here, due to space constraints. For our example, the final dendrogram is shown on the right part of Figure 9.

[5]For efficient merging of clusters, we use the weighted-union heuristic of Union Find [4].

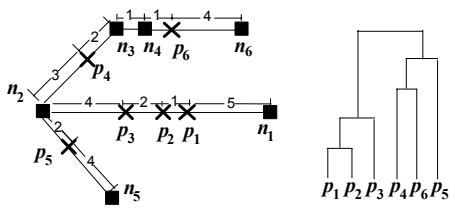


Figure 9: Example of hierarchical clustering

4.4.2 *Making Single-Link scalable*

The algorithm of Figure 8 initializes one cluster for each point in the dataset. Moreover it initializes a very large heap P of candidate cluster pairs to be merged. In large problems, the $O(N)$ information needed to be stored for the N clusters (e.g., cluster-IDs) and the size of heaps P and Q may exceed the memory capacity of the system. In order to reduce this information, we can use a simple heuristic. In the first phase, we immediately merge points on an edge, whose distance is at most δ, where δ is a user-defined threshold. In Figure 9, for instance, if $\delta = 2$, then p_1, p_2, and p_3 are immediately merged to a single cluster in the first phase of the algorithm. Moreover, at lines 15–21 of Figure 8, we immediately merge all cluster pairs in P with distance at most δ. In this way, the number of clusters to start with and the sizes of the queues significantly reduce. The price to pay is that we lose the first merges of the dendrogram, however, these are not usually important to the data analyst. An appropriate value of δ can be chosen by sampling on the dense edges of the network. Notice that the selection of δ is not as restrictive as the selection of ϵ in ϵ-Link, since dense clusters for distances $\epsilon > \delta$ will still be discovered by Single-Link.

5. Experimental Evaluation

In this section, we evaluate the effectiveness and efficiency of the proposed techniques. We implemented all clustering algorithms described in Sections 4.2, 4.3, and 4.4 in C++. Experiments were run on a PC with a Pentium 4 CPU of 2.3GHz. In all experiments, we used a memory buffer of 1Mb and the page size was set to 4Kb.

We used four real road networks, depicted in Figure 10. All networks are connected and for each of them the number of nodes and edges are in parentheses after the codename. The first dataset (NA) consists of main roads in North America obtained from `www.maproom.psu.edu/dcw/` and cleaned to form a connected network. The other three are road maps of San Francisco (SF), San Joaquin County (TG), and Oldenburg (OL), obtained from [3]. Since the original SF and TG networks were not connected, we extracted the largest connected component from them. The weights of the graph edges were set equal to the Euclidean distance of the connected nodes. This does not affect the generality of the methods and at the same time facilitates

evaluation of the results by visualization.

On the road networks, we generated data that simulate real world clusters. The density and spread of the clusters are controlled by two parameters; an initial separation distance s_{init} and a magnification factor $F > 1$. Let $|C_{final}|$ be the total number of points in a generated cluster. Initially, a random edge of the network and the first point of the cluster is generated on it. Then, the network is traversed using Dijkstra's algorithm. Whenever an edge is met for the first time, points are generated on it. Let $|C|$ be the current size of the cluster. The distance from a newly generated point to the previous one is randomly chosen in the range $[0.5s_{cur}, 1.5s_{cur}]$, where $s_{cur} = s_{init} + s_{init}(F-1)\frac{|C|}{|C_{final}|}$. Roughly speaking, the approximate distance s_{cur} between two consecutive generated points is initially s_{init} and increases as the network is expanded to reach $s_{init} \times F$ for the final point. This models the case where the cluster has a dense "core" and becomes sparser at its boundaries. In all generated pointsets of size N we experimented with, 99% points were evenly distributed to k equal-sized clusters and 1% were randomly generated outliers. F was set to 5.

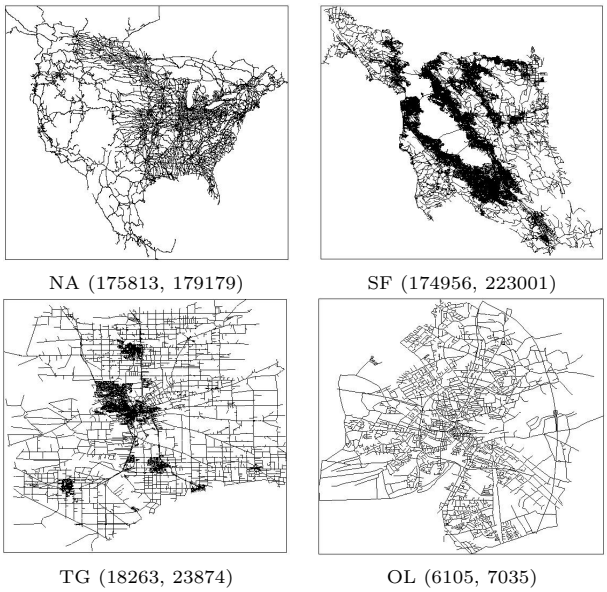


Figure 10: Four datasets used in experiments

5.1 Clustering effectiveness

In general, the proposed methods discover correctly clusters according to their definitions. Figure 11 shows a representative example of the discovered structures by the algorithms. Other tested datasets give similar results. In this experimental instance, we have generated 20,000 points on the OL network (this network is quite small and clusters can be easily spotted on its visualization). 19,800 points were evenly distributed to $k = 10$ clusters, whereas 1% of the data (i.e., 200 points) are outliers, randomly generated.

Figure 11a shows a typical result of the k-medoids algorithm, when given a initial set of random k medoids. In order to control the cost of the algorithm, we allowed only 15 unsuccessful swaps for termination to a local optimum. The result is quite different compared to the natural clusters, which can be easily spotted by the reader. Some original clusters are merged or split and every outlier is included in some cluster. We have also tested an "ideal" case (see Figure 11b), where the initial set of medoids were the first points of the generated clusters. Even in this case, the algorithm cannot discover all clusters exactly (observe that the leftmost cluster is split), suffering from the problems most single-representative methods have [6].

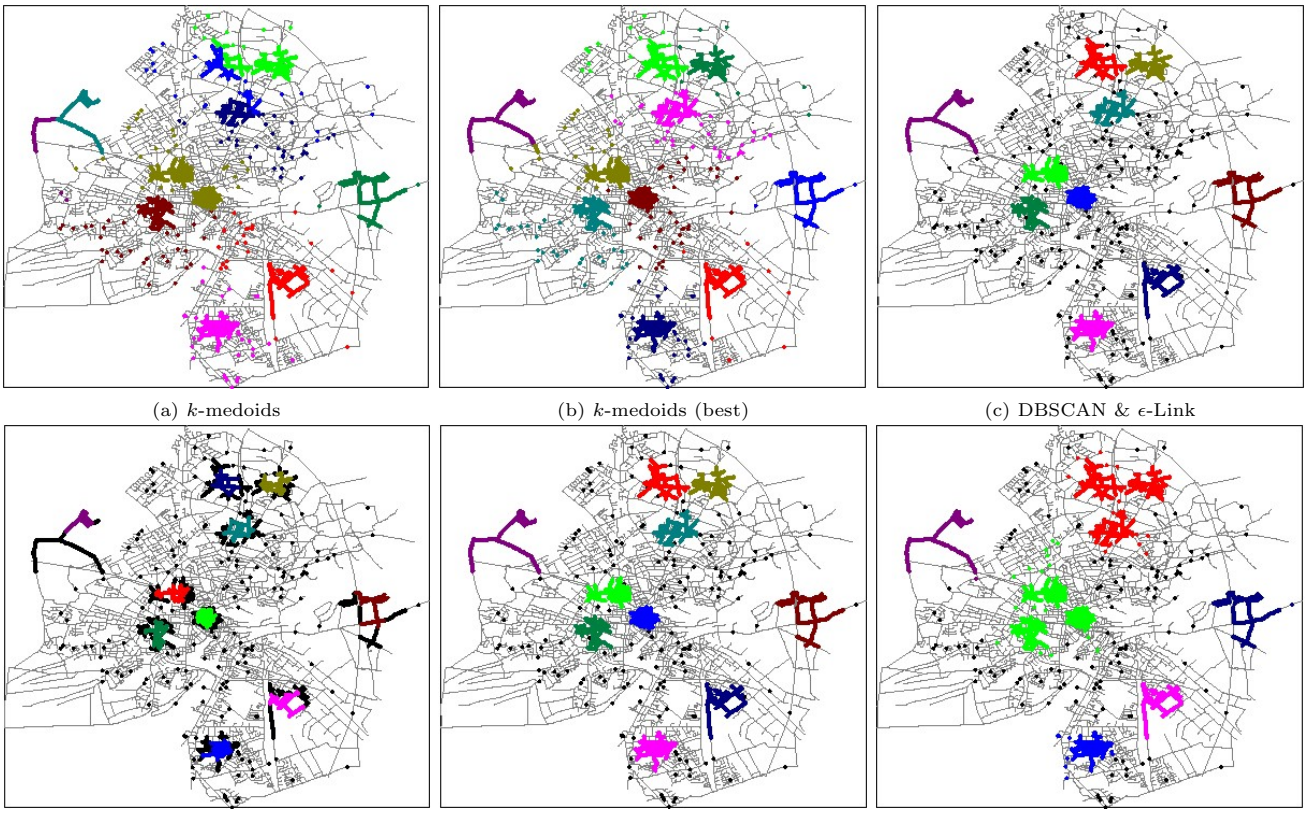
On the other hand, the density-based algorithms (the adaptation of DBSCAN and our ϵ-Link method) discover correctly the clusters, if given the appropriate parameter values ϵ and $MinPts$. Figure 11c shows the result of both algorithms, when $\epsilon = 1.5 \times s_{init} \times F$ and $MinPts = 2$. Essentially, the results of the algorithms are identical and correct. DBSCAN can also discover the clusters for larger values of ϵ and $MinPts$, however, at higher cost.

The hierarchical method (Single-Link) can discover the clusters, but it needs more time due to the expensive initial phase (every point is a cluster and the size of P is $O(N)$). Figure 11d plots the result of this method when the scalability heuristic is used with a small $\delta = s_{init} \times F$. In this case, the number of clusters to start with is one order of magnitude smaller than N and so are the sizes of the heaps P and Q. Notice that we only plot with colors large clusters of > 100 points. At the stage of Figure 11e, the 10 large clusters have been discovered. Figure 11f shows a latter set of 6 large clusters in the hierarchy, where two triplets of original clusters close to each other have been merged. If the user decides to terminate the algorithm at some point, clusters with a single of few points are treated as outliers by the algorithm. An interesting property of Single-Link is that, according to its definition, it discovers exactly the same clusters as ϵ-Link, at the point where the next popped element from P has distance $> \epsilon$. In other words, a version of Single-Link that stops merging when the next distance exceeds a parameter ϵ produces identical results as ϵ-Link.

5.2 Efficiency and Scalability

We tested the efficiency and scalability of the algorithms under various problem settings. In the first experiment, we compare the effects of incremental medoid replacement on the run-time of k-medoids. We generated 500K points in k clusters on the SF network. Figure 12 shows the average speedup achieved by the incremental medoid replacement, over the naive assignment of points to clusters from scratch, after some medoid replacement. As expected, the speedup increases with k, since the number of network nodes (and points) that are re-located to another cluster becomes smaller.

Table 1 shows the time spent by k-medoids to converge to a local optimum given a random set of k initial medoids. The table includes the number of iterations, as well as the execution time of each iteration (initial and incremental) of k-medoids for four datasets generated on networks NA, SF, TG, and OL, distributed to $k = 10$ clusters. The number of points are 500K, 500K, 50K, and 20K, respectively (roughly three times the number of network nodes). Observe that in all cases an incremental iteration takes roughly four times less than the first iteration, an improvement consistent with the speedup of Figure 12 for $k = 10$. We can also see that the algorithm converges quite fast; the local optimum is found after 4–8 iterations plus 15 unsuccessful medoids replacements are attempted from them. Of course, the overall cost



(a) k-medoids (b) k-medoids (best) (c) DBSCAN & ϵ-Link

(d) Single-Link, after the initial phase (e) Single-Link, after reaching ϵ (f) Single-Link, when finding 6 large clusters

Figure 11: Visualization of clustering results

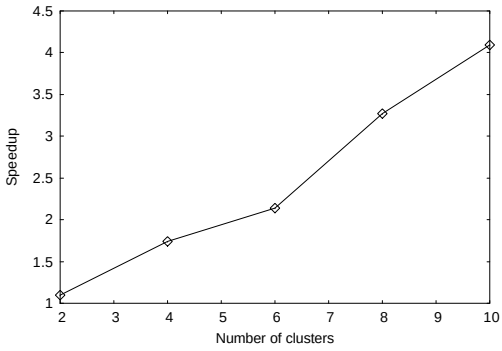


Figure 12: Speedup of incremental replacement

Table 1: Execution cost of k-medoids (sec)

	# iterations	first one	next ones
NA	21	20.6	5.8
SF	19	23	5.6
TG	22	1.4	0.39
OL	23	0.47	0.11

of the algorithm depends on the number of local optima which are evaluated until the clusters are finalized to the best one.

We measured the execution cost of k-medoids, DBSCAN, ϵ-Link, and the hierarchical Single-Link algorithm, using the datasets of the previous experiment. Table 2 shows the costs of the four methods. For k-medoids, we include the cost of finding only one local optimum; even in this case the algorithm is much slower than the other methods (usually producing much worse results, too). For DBSCAN, we set $MinPts = 2$ and used the same ϵ as ϵ-Link uses, i.e., the minimum value that discovers all clusters correctly, based on their generation. Notice that with these values DBSCAN has the lowest cost than any other combination of values that can successfully discover the clusters. ϵ-Link with the same (optimal) ϵ can discover the clusters much faster, due to its more systematic way of visiting the network for each cluster. This demonstrates that straightforward application of existing clustering methods on the problem are much more expensive than our specialized network-based techniques. For Single-Link, we included the cost of discovering the whole dendrogram which requires a traversal of the whole graph. Moreover, we used the scalability heuristic and set $\delta = 0.7 \times \epsilon$. With this setting, heaps P and Q are small enough to be easily accommodated in memory. Single-Link is slower than ϵ-Link because it essentially traverses the whole graph in order to compute the complete dendrogram.

Next, we test the scalability of the methods to the number N of points and the size $|V|$ of the network. We first

Table 2: Execution cost of the algorithms (sec)

	k-medoids	DBSCAN	ϵ-Link	Single-Link
NA	136.2	74.1	18.3	24.6
SF	124.1	62.4	11.6	25.9
TG	9.6	4.03	0.33	1.45
OL	2.8	0.92	0.12	0.48

generated N=100K, 200K, 500K, 1000K points on the SF network, grouped in $k = 10$ clusters, except from 1% of outliers. Figure 13 shows the execution cost of the four algorithms as a function of the number of points N. Observe that all algorithms are scalable with N, however, there are differences in the cost increase rate. The costs of DBSCAN and ϵ-Link are directly proportional to N. This is due to the fact that the points file is accessed randomly and sometimes multiple times. With the increase of N, more edges become populated and randomly accessing the points on them has severe impact on performance. On the other hand, the costs of k-medoids and Single-Link increase very slowly, appearing to depend mainly on the size of the network. This can be explained by the fact that both algorithms traverse the whole network, but the accesses of the points file are limited and sequential; k-medoids scans and assigns the points once for every iteration, whereas Single-Link scans the points file only once during its first step, using the heaps and traversing the network onwards.

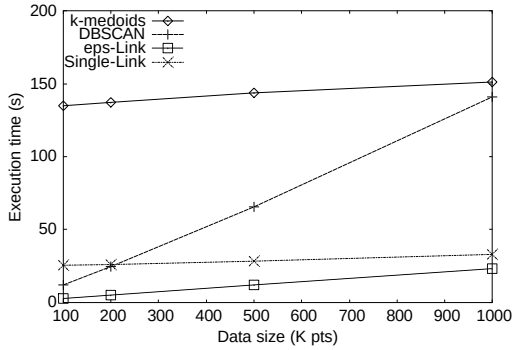


Figure 13: Scalability with N

In the next experiment, we extracted connected components of SF, consisting of 10%, 20%, and 50% nodes. We also used the original network (100%). On each network, we generated 200K points in $k = 10$ clusters, including 1% of outliers, as usual. Figure 14 shows the costs of the algorithms as a function of the network size. Again, all methods are scalable. In this case, the costs of k-medoids and Single-Link increase proportionally to $|V|$, since the methods traverse the whole network. On the other hand, the part of the network traversed by the density-based algorithms increases slowly, since the number of edges that contain points (and need to be visited) does not change significantly with the network size. Nevertheless we observed that Single-Link has similar cost to ϵ-Link, if we stop the algorithm when the merge distance exceeds ϵ.

In summary, assuming a planar network, the costs of DBSCAN and ϵ-Link are linear to N and affected only by

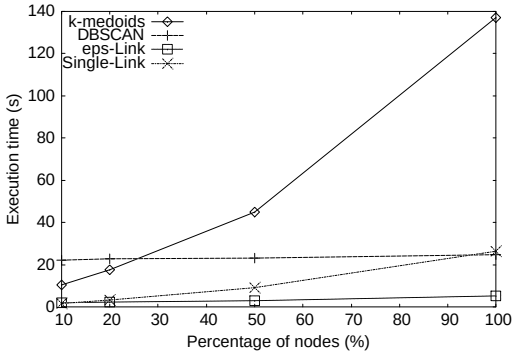


Figure 14: Scalability with $|V|$

the number of nodes $|V'|$ which are spanned by the clusters. On the other hand, k-medoids (one iteration) and Single-Link traverse the whole network, thus their cost is $O(|V| \log |V| + N)$. The worst-case complexity of all methods (assuming that the clusters span the whole network) is $O(|V| \log |V| + N)$.

5.3 Discovery of interesting clusters

If N is large, Single-Link produces a huge hierarchy of clusters which can be hard to analyze manually. Figure 15 plots the merge distance of the last 249 cluster pairs, popped from heap P, while Single-Link clusters the Oldenburg dataset of Section 5.1. We can spot three merge instances, where the distance difference between consecutive merges changes significantly (see the arrows on the figure). These merges correspond to interesting levels of clustering. For instance, the first one has the sharpest distance change and occurs when the merge distance has reached ϵ (see Figure 11e), i.e., when the original clusters have been discovered. We can detect these sharp changes manually from this graph and trace back the (incremental) history of merges to recover the interesting clustering level.

It would also be nice if Single-Link could automatically detect these interesting levels. For this task, the algorithm can maintain a vector $\{d_1 - d_0, d_2 - d_1, \ldots, d_K - d_{K-1}\}$ with differences between the distances of the last K consecutive merges and incrementally compute their average $d_{avg} = \frac{1}{K} \sum_{i=1}^{K} (d_i - d_{i-1})$. If the distance of the next merge is significantly larger than d_{avg}, Single-Link can hint the user for an interesting level in the merge hierarchy. In this way, multiple interesting levels of different resolution can be identified. For instance, if a sparse cluster contains multiple dense ones, the two levels would be identified at a single pass of Single-Link.

6. Discussion

Network-based clustering is useful for a variety of data analysis tasks and is not limited to the problem definition we have studied in this paper. The proposed algorithms are general enough to be applied for several interesting variants.

Our model allows clustering on networks, where arbitrary types of weights can be assigned on the edges. For instance, the weight on an edge connecting two network nodes could be their Euclidean distance, the time to travel from one node to another, the cost (price) of traversing the edge,

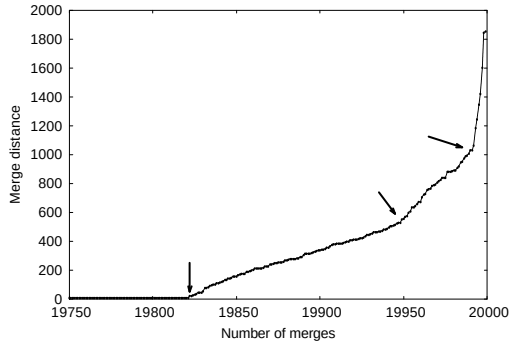


Figure 15: Merge distance during Single-Link

etc. Depending on the measure used, clustering may return different results, providing multiple clustering layers to the data analyst. Apart from this, it is possible to combine different weight measures with an aggregate function. An advanced problem is the discovery of time-dependent clusters in a model, where edge weights vary with time. For example, traffic on a road segment depends on the time of the day and/or the date of the week. Based on this model, we can derive clusters whose content is time-parameterized.

Another application is the discovery of clusters across different networks (e.g., a road network and a river/canal network) by combining both of them. For this, we can define *transition edges* that connect pairs of points from the networks (e.g., piers). *Transition weights* are assigned on them to model the cost of transition. In this way, shortest path distances between objects from different original networks can be defined in the combined network and discovered clusters may contain objects lying on both graphs.

7. Conclusion

In this paper, we have studied the problem of clustering objects which lie on a large spatial network. We presented efficient algorithms of three different paradigms, namely partitioning, density-based, and hierarchical. The algorithms avoid computing the distances between every pair of points, by exploiting the properties of the network.

A qualitative comparison between the methods shows that k-medoids is not effective; due to the randomized nature of search, it fails to discover the best partitioning within limited time. Moreover, it may split clusters and include outliers in them. On the other hand, ϵ-Link (and DBSCAN) can correctly discover the clusters if they are given the appropriate parameters. Finally, Single-Link can also correctly discover the clusters at a node of the merging dendrogram.

A cost comparison between the methods shows that k-medoids is the most expensive one due to the multiple times it requires to traverse the graph. ϵ-Link is very efficient because it accesses only the part of the graph that contains points. However, the points are accessed randomly (as opposed to k-medoids and Single-Link). DBSCAN has similar properties to ϵ-Link, but it is much slower because the graph is traversed less systematically (a range query is issued for every point in the dataset). Single-Link is fast, given the fact that it traverses the whole graph. Moreover, Single-Link does not rely on parameters like ϵ and can discover clusters at multiple hierarchies.

In Section 5.3, we described a method that can assist the automatic discovery of interesting clusters during Single-Link. In the future, we plan to further investigate the applicability of this method. We will also develop hierarchical algorithms that consider distances between multiple points from the merged clusters (e.g., representatives). Finally, we plan to study the extensions of the problem, discussed in Section 6.

References

[1] R. Agrawal and H. Jagadish. Algorithms for searching massive graphs. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):225–238, 1994.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering points to identify the clustering structure. In *ACM SIGMOD*, 1999.

[3] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[5] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *ACM SIGMOD*, 1998.

[6] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[7] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[8] N. Jing, Y. W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, 1998.

[9] S. Jung and S. Pramanik. HiTi graph model of topographical roadmaps in navigation systems. In *ICDE*, 1996.

[10] G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[12] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, Inc, 1990.

[13] E. Martin, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM SIGKDD*, 1996.

[14] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos. C2P: Clustering based on closest pairs. In *VLDB*, 2001.

[15] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, 1994.

[16] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.

[17] S. Shekhar and D. Liu. CCAM: A connectivity-clustered access method for networks and network computations. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):102–119, 1997.

[18] C. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 20:68–86, 1971.

[19] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *ACM SIGMOD*, 1996.